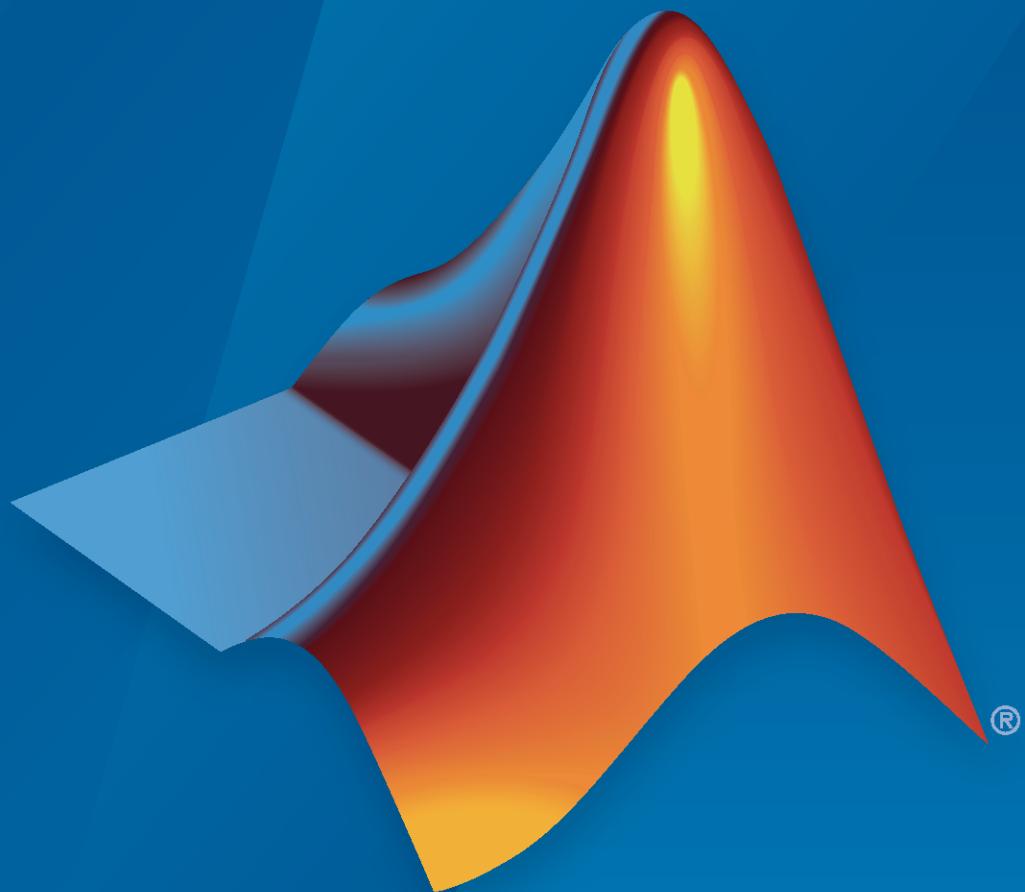


# Communications Toolbox™ Support Package for USRP™ Radio

## User's Guide



# MATLAB® & SIMULINK®

R2020b

 MathWorks®

# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)  
Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

*Communications Toolbox™ Support Package for USRP™ Radio User's Guide*

© COPYRIGHT 2014–2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

**FEDERAL ACQUISITION:** This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

September 2014	Online only	Revised for Version 14.2.0 (R2014b)
March 2015	Online only	Revised for Version 15.1.0 (R2015a)
April 2015	Online only	Revised for Version 15.1.1 (R2015a)
September 2015	Online only	Revised for Version 15.2.0 (R2015b)
March 2016	Online only	Revised for Version 16.1.0 (R2016a)
September 2016	Online only	Revised for Version 16.2.0 (R2016b)
October 2016	Online only	Revised for Version 16.2.1 (R2016b)
March 2017	Online only	Revised for Version 17.1.0 (R2017a)
April 2017	Online only	Revised for Version 17.1.1 (R2017a)
September 2017	Online only	Revised for Version 17.2.0 (R2017b)
March 2018	Online only	Revised for Version 18.1.0 (R2018a)
September 2018	Online only	Revised for Version 18.2.0 (R2018b)
March 2019	Online only	Revised for Version 19.1.0 (R2019a)
September 2019	Online only	Revised for Version 19.2.0 (R2019b)
March 2020	Online only	Revised for Version 20.1.0 (R2020a)
April 2020	Online only	Revised for Version 20.1.1 (R2020a)
September 2020	Online only	Revised for Version 20.2.0 (R2020b)

## Getting Started with Communications Toolbox Support Package for USRP Radio

1

<b>Supported Hardware and Required Software</b> .....	<b>1-2</b>
Supported Hardware .....	1-2
Supported Daughter Cards .....	1-2
Firmware .....	1-3
MathWorks Products .....	1-3
Recommended Products .....	1-3
<b>Install Communications Toolbox Support Package for USRP Radio</b> .....	<b>1-4</b>
<b>Guided USRP Radio Support Package Hardware Setup</b> .....	<b>1-5</b>
<b>Configure Host Computer for USB-Based Radio Connection</b> .....	<b>1-7</b>
Install USB Driver for Windows .....	1-7
Install USB Driver for Linux .....	1-8
Install USB Driver for Macintosh .....	1-10
<b>Configure Host Computer for Ethernet-Based Radio Connection</b> .....	<b>1-13</b>
Host Computer Ethernet Options .....	1-13
Configure Ethernet Connection Using Installer .....	1-13
<b>Configure Network Interface Using Installer with No Radio Connected</b> .....	<b>1-19</b>
<b>Test Radio Connection</b> .....	<b>1-21</b>
<b>Verify MATLAB Connection to USRP Radio</b> .....	<b>1-22</b>
<b>Check Radio Firmware</b> .....	<b>1-23</b>
<b>Manual USRP Radio Support Package Hardware Setup</b> .....	<b>1-24</b>
<b>Configure Ethernet Connection Manually on Windows 10</b> .....	<b>1-26</b>
Configure Ethernet Connection Via Windows Network Connections App .....	1-26
Configure Ethernet Connection Via Windows Command Prompt .....	1-30
Configure FastSendDatagramThreshold Registry Key .....	1-32
<b>Configure Ethernet Connection Manually on Linux</b> .....	<b>1-34</b>
Configure Ethernet Connection Via Linux Command Prompt .....	1-34
<b>Configure Ethernet Connection Manually on Mac</b> .....	<b>1-39</b>
Configure Ethernet Connection Via Mac System Preferences .....	1-39

Configure Ethernet Connection Via Mac Terminal Window .....	<b>1-40</b>
<b>Verify Hardware Connection .....</b>	<b>1-44</b>
Check Ethernet Configuration .....	1-44
Check Host-to-Radio Connection .....	1-44
Check Subnet Values on Host and Radio .....	1-46
<b>Using One Ethernet Port .....</b>	<b>1-49</b>
<b>Make Changes Persistent on Linux .....</b>	<b>1-50</b>
<b>Linux Systems with No prlimit Command .....</b>	<b>1-51</b>
<b>Resolving Ethernet Subnet Conflict .....</b>	<b>1-52</b>
<b>USRP Radio Firmware Update .....</b>	<b>1-53</b>
Why Download New Firmware? .....	1-53
Updating USRP Radio Firmware .....	1-53
<b>What to Do After Installation .....</b>	<b>1-55</b>
<b>Uninstall Support Packages .....</b>	<b>1-56</b>
<b>Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection .....</b>	<b>1-57</b>
Host Computer Ethernet Options .....	1-57
Configure Ethernet Connection Using Installer .....	1-57
<b>Configure Network Interface Using Installer with No USRP N3xx Radio Connected .....</b>	<b>1-73</b>

## Common Problems and Fixes

2

<b>Common Problems and Fixes .....</b>	<b>2-2</b>
Cannot Log into Computer .....	2-2
Ethernet Subnet Conflict .....	2-2
Ping command times out .....	2-3
Firmware is incompatible with host build .....	2-3
USRP radio is busy .....	2-3
USRP radio is not responding .....	2-4
Unexpected number of samples in burst reception .....	2-5
Buffer could not be resized .....	2-6
UHD driver cannot set thread priorities .....	2-6
Function findsdru throws error .....	2-6
Getting overruns or underruns .....	2-8
Slow Response for SDR System Objects and Blocks .....	2-8
No devices found .....	2-9
USB 2.0 not fast enough with Bus Series radios .....	2-9
8-bit Transport in Streaming Mode Causes libuhd error .....	2-10
Burst Mode Failure .....	2-10

Find USRP Devices Connected To Computer .....	3-2
---	-----

**Radio Management**

<b>Check Radio Connection</b> .....	4-2
Block Connection .....	4-2
System Object Connection .....	4-2
<b>Run in Offline Mode</b> .....	4-4
<b>Query and Set IP Addresses</b> .....	4-5
Block Parameter IP Address .....	4-5
System Object Property IP Address .....	4-5
Set IP Address with setsdruiip Function .....	4-5
<b>Radio Configuration</b> .....	4-7
<b>Single Channel Input and Output Operations</b> .....	4-8
Perform SISO Operations with SDRu System Objects .....	4-8
Perform SISO Operations with SDRu Blocks .....	4-9
<b>Multiple Channel Input and Output Operations</b> .....	4-14
About MIMO Mode .....	4-14
Perform MIMO Operations with SDRu System Objects .....	4-14
Perform MIMO Operations with SDRu Blocks .....	4-16
Perform MIMO Operations Bundling Multiple Radios .....	4-17
<b>Detect Underruns and Overruns</b> .....	4-22
Detect Lost Samples Using SDRu Transmitter Block .....	4-22
Detect Lost Samples Using SDRu Receiver Block .....	4-22
Detect SDRu Transmitter System Object Underruns .....	4-23
Detect SDRu Receiver System Object Overruns .....	4-24
<b>Data Frame Lengths</b> .....	4-26
Set Frame Length in SDRu Receiver Block .....	4-26
Set Frame Length in SDRu Receiver System Object .....	4-26
<b>Apply Conditional Execution</b> .....	4-27
Using Conditional Execution with SDRu Receiver Block .....	4-27
Using Conditional Execution with SDRu Receiver System Object .....	4-27
<b>Change Transport Data Rate</b> .....	4-29
Set Transport Data Type in Simulink .....	4-29
Set Transport Data Type in MATLAB .....	4-29
<b>Transmit and Receive Using External Clock</b> .....	4-34
Specify External Clock in SDRu Blocks .....	4-34

Specify External Clock in SDRu System Objects .....	<b>4-34</b>
<b>Desired Vs. Actual Parameter Values</b> .....	<b>4-37</b>
Desired vs. Actual in the SDRu Blocks .....	<b>4-37</b>
Desired vs. Actual in the SDRu System Objects .....	<b>4-38</b>
<b>Supported Data Types</b> .....	<b>4-40</b>

## Performance Optimization

**5**

<b>Burst-Mode Buffering</b> .....	<b>5-2</b>
What is Burst Mode? .....	<b>5-2</b>
Determining If You Need Burst Mode .....	<b>5-2</b>
Using Burst Mode .....	<b>5-2</b>
<b>Model Performance Optimization</b> .....	<b>5-4</b>
Acceleration .....	<b>5-4</b>
Model Tuning .....	<b>5-4</b>
Simulink Code Generation .....	<b>5-4</b>
<b>MATLAB Performance Improvements</b> .....	<b>5-6</b>
Vector-Based Processing .....	<b>5-6</b>
MATLAB Code Generation .....	<b>5-6</b>
<b>Computing Environment Optimization</b> .....	<b>5-7</b>

## FPGA Targeting

**6**

<b>FPGA Targeting Overview</b> .....	<b>6-2</b>
About FPGA Targeting .....	<b>6-2</b>
<b>Target FPGA with Custom Bitstream</b> .....	<b>6-3</b>
Create Algorithm .....	<b>6-3</b>
Set Tool Path .....	<b>6-4</b>
Create FPGA Files with HDL Workflow Advisor .....	<b>6-4</b>
Burn Custom FPGA File .....	<b>6-5</b>
Verify FPGA Implementation .....	<b>6-5</b>

## **Communications Toolbox Support Package for USRP Radio Examples**

Packetized Modem with Data Link Layer .....	10-2
Family Radio Service (FRS) Full-Duplex Transceiver with USRP® Hardware .....	10-14
FM Receiver with USRP® Hardware .....	10-20
Frequency Offset Calibration Receiver with USRP® Hardware .....	10-24
Frequency Offset Calibration Transmitter with USRP® Hardware .....	10-29
FRS/GMRS Walkie-Talkie Receiver with USRP® Hardware .....	10-34
FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware .....	10-40
IEEE® 802.11™ WLAN - OFDM Beacon Receiver with USRP® Hardware .....	10-46
LTE Cell Search, MIB and SIB1 Recovery with Two Antennas .....	10-51
LTE SIB1 Transmission over Two Antennas .....	10-67
Multi-User Transmit Beamforming with USRP® Hardware .....	10-71
QPSK Receiver with USRP® Hardware .....	10-80
QPSK Transmitter with USRP® Hardware .....	10-85
FM Receiver with USRP® Hardware .....	10-89
Frequency Offset Calibration with USRP® Hardware .....	10-91

<b>FRS/GMRS Walkie-Talkie Receiver with USRP® Hardware . . . . .</b>	<b>10-96</b>
<b>FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware . . . . .</b>	<b>10-100</b>
<b>QPSK Receiver with USRP® Hardware . . . . .</b>	<b>10-104</b>
<b>QPSK Transmitter with USRP® Hardware . . . . .</b>	<b>10-109</b>

# Getting Started with Communications Toolbox Support Package for USRP Radio

---

- “Supported Hardware and Required Software” on page 1-2
- “Install Communications Toolbox Support Package for USRP Radio” on page 1-4
- “Guided USRP Radio Support Package Hardware Setup” on page 1-5
- “Configure Host Computer for USB-Based Radio Connection” on page 1-7
- “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13
- “Configure Network Interface Using Installer with No Radio Connected” on page 1-19
- “Test Radio Connection” on page 1-21
- “Verify MATLAB Connection to USRP Radio” on page 1-22
- “Check Radio Firmware” on page 1-23
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24
- “Configure Ethernet Connection Manually on Windows 10” on page 1-26
- “Configure Ethernet Connection Manually on Linux” on page 1-34
- “Configure Ethernet Connection Manually on Mac” on page 1-39
- “Verify Hardware Connection” on page 1-44
- “Using One Ethernet Port” on page 1-49
- “Make Changes Persistent on Linux” on page 1-50
- “Linux Systems with No prlimit Command” on page 1-51
- “Resolving Ethernet Subnet Conflict” on page 1-52
- “USRP Radio Firmware Update” on page 1-53
- “What to Do After Installation” on page 1-55
- “Uninstall Support Packages” on page 1-56
- “Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection” on page 1-57
- “Configure Network Interface Using Installer with No USRP N3xx Radio Connected” on page 1-73

# Supported Hardware and Required Software

## Supported Hardware

The Communications Toolbox Support Package for USRP Radio supports Ettus Research™ devices listed in this table and has been tested on devices using “Supported Daughter Cards” on page 1-2. Support might also extend to other UHD™-based radios and daughterboards from Ettus Research.

National Instruments™ radios equivalent to the listed Ettus Research radios are also supported.

Radio Series	Comment
USRP2™	—
USRP Networked Series (N210 and N200)	Collectively referred to in this documentation as N2xx.
USRP Networked Series (N300*, N310**, N320*, and N321*)	Collectively referred to in this documentation as N3xx.
USRP Bus Series (B200 and B210*)	Collectively referred to in this documentation as B-series.
USRP X Series (X310* and X300*)	Collectively referred to in this documentation as X3xx.  X3xx radios with the TwinRX daughterboard support up to four-channel reception with phase synchronization present between all channels.
* SISO and 2x2 MIMO modes are supported.	
** 4x4 MIMO mode is supported.	

## Supported Daughter Cards

These daughter cards are used with USRP2, N2xx, N3xx, and X3xx series radios. Support might also extend to other UHD-based radios and daughterboards from Ettus Research.

- DBSRX
- DBSRX2
- LFRX
- LFTX
- SBX
- TVRX
- TVRX2
- WBX
- XCVR2450
- TwinRX is compatible with X3xx series radios only.

## Firmware

The blocks and System objects use a specific version of the UHD driver. The FPGA binaries on your USRP radio must match the specific version of the UHD drivers supported by the current release.

## MathWorks Products

The following MathWorks products are required for using Communications Toolbox Support Package for USRP Radio.

- MATLAB®
- Communications Toolbox
- DSP System Toolbox™
- Signal Processing Toolbox™
- Communications Toolbox Support Package for USRP Radio

## Recommended Products

- Simulink®
- HDL Coder™

## See Also

### Related Examples

- “Install Communications Toolbox Support Package for USRP Radio” on page 1-4

## Install Communications Toolbox Support Package for USRP Radio

- 1** On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Get Hardware Support Packages**.
- 2** In Add-On Explorer, browse or search for the Communications Toolbox Support Package for USRP Radio.
- 3** Select the support package, and then click **Install**.
- 4** The support package installer prompts you while it installs drivers needed for the USRP Radio software.

To review the installer steps before beginning the installation, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5. Use the installer for the Communications Toolbox Support Package for USRP Radio installation and to configure your host-to-radio connection. After installing the software package, if you choose to manually configure your host-to-radio connection, follow the instructions at “Manual USRP Radio Support Package Hardware Setup” on page 1-24.

To get the support package and begin the installation, see “Get and Manage Add-Ons”.

### See Also

### Related Examples

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

# Guided USRP Radio Support Package Hardware Setup

To use Communications Toolbox Support Package for USRP Radio features, you must establish communication between the host computer and the radio hardware. For details on establishing and verifying the connection between the host computer and the radio, see the related topics. Each operating system (OS) has instructions specific to that OS. Choose the instructions for the OS on the host computer.

- The host computer can be a desktop or a laptop.
- For USB-based radios (Ettus Research Bus Series radios - B200, B200mini, B200mini-i, B205mini-i, or B210), the host computer must have at least one available USB port to connect to the radio.
- For Ethernet-based radios (Ettus Research Networked or X Series radios - N200, N210, USRP2, X300, or X310), the host computer must contain at least one dedicated Gigabit network interface card (NIC) for connecting to the USRP radio.
  - When connecting to an Ethernet-based radio, a second NIC is recommended for the host computer to remain simultaneously connected to the radio and a network (or the Internet). Alternatively, you can use one NIC for the radio connection and WiFi to connect to a network. If the host computer has no WiFi and only one NIC, see “Using One Ethernet Port” on page 1-49.
  - Directly connect your USRP radio to the NIC on the host computer with an Ethernet cable.
  - USRP radios have a factory default IP address of 192.168.10.2. This value is used for the radio IP address throughout the setup instructions. If your radio has a different IP address, modify accordingly.
- For Ethernet-based USRP N3xx radios (Ettus Research Networked Series radios - N300, N310, N320, or N321), the host computer must contain at least one dedicated 1 or 10 Gigabit network interface card (NIC) for connecting to the USRP radio.
  - Make sure you have at least one microSD card reader and one writable microSD card. The SD card should have at least 16GB storage. If the host computer does not have an integrated card reader, use an external USB SD card reader.
  - When connecting to an Ethernet-based USRP N3xx radio, a second NIC is recommended for the host computer to remain simultaneously connected to the radio and a network (or the Internet). Alternatively, you can use one NIC for the radio connection and WiFi to connect to a network. If the host computer has no WiFi and only one NIC, see “Using One Ethernet Port” on page 1-49.
  - Directly connect your USRP radio to the NIC on the host computer with an Ethernet cable.
  - USRP radios have a factory default IP address of 192.168.10.2. This value is used for the radio IP address throughout the setup instructions. If your radio has a different IP address, modify accordingly.

To install the Communications Toolbox Support Package for USRP Radio software and configure the host computer, use the recommended installer by following the “Install Communications Toolbox Support Package for USRP Radio” on page 1-4 directions.

After completing the Communications Toolbox Support Package for USRP Radio installation, the installer guides you in setting up the connection between the host and the radio.

## See Also

### More About

- “Configure Host Computer for USB-Based Radio Connection” on page 1-7
- “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13
- “Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection” on page 1-57
- “Configure Network Interface Using Installer with No Radio Connected” on page 1-19
- “Configure Network Interface Using Installer with No USRP N3xx Radio Connected” on page 1-73
- “Test Radio Connection” on page 1-21
- “Verify MATLAB Connection to USRP Radio” on page 1-22

# Configure Host Computer for USB-Based Radio Connection

## In this section...

["Install USB Driver for Windows" on page 1-7](#)

["Install USB Driver for Linux" on page 1-8](#)

["Install USB Driver for Macintosh" on page 1-10](#)

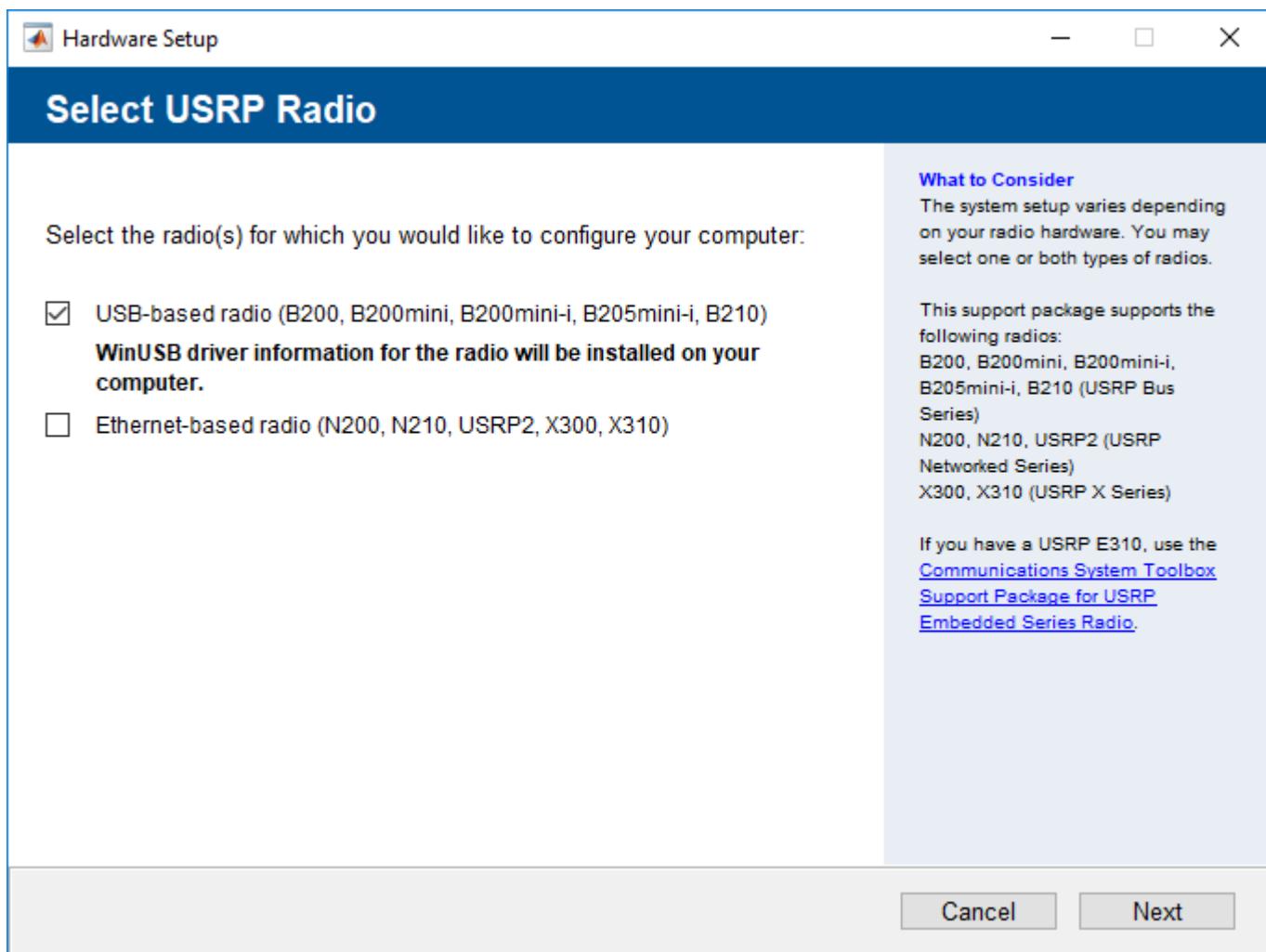
These sections guide you through the instructions to configure a host computer for an USB-based radio connection using the installer for Windows®, Linux®, and Macintosh computers. Refer to the operating system instructions of interest to you. To run the installer, see “Install Communications Toolbox Support Package for USRP Radio” on page 1-4.

**Note** If you are using an Ethernet-based radio connection, see “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13.

## Install USB Driver for Windows

The installer guides you through the installation and update of the WinUSB driver information.

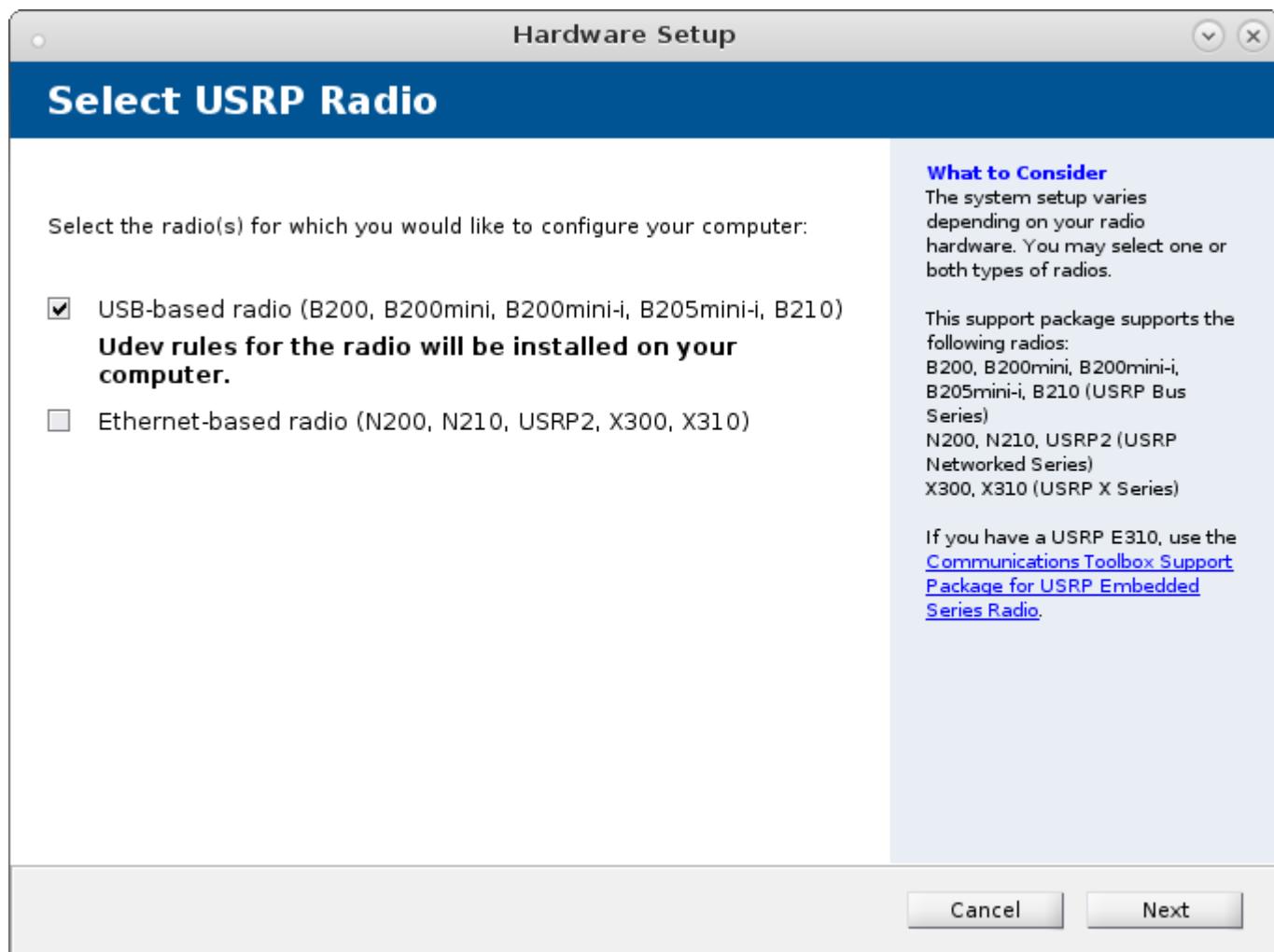
- 1 On the first Hardware Setup window for the installer, select **USB-based radio**, and then click **Next**.



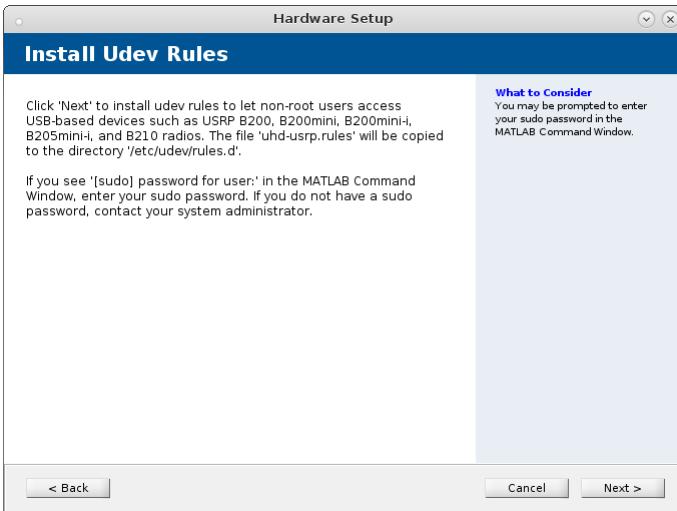
- 2 During the WinUSB driver information update, if you see a dialog box asking **Do you want to allow this app to make changes to your device?**, click **Yes**.
- 3 Connect a USB-based radio to an available USB port on the host computer. When you click **Next**, Windows installs the device driver automatically. If you get a Windows message telling you the driver could not be installed, try disconnecting the radio from the USB port of the host computer. When you reconnect the radio to any USB port, Windows checks the WinUSB driver and reinstalls the driver, if necessary.
- 4 After enabling USB-based devices, you can confirm the host-to-radio communication link by testing the radio connection. For details, see "Test Radio Connection" on page 1-21.

## Install USB Driver for Linux

- 1 On the first Hardware Setup window for the installer, select **USB-based radio**, and then click **Next**.



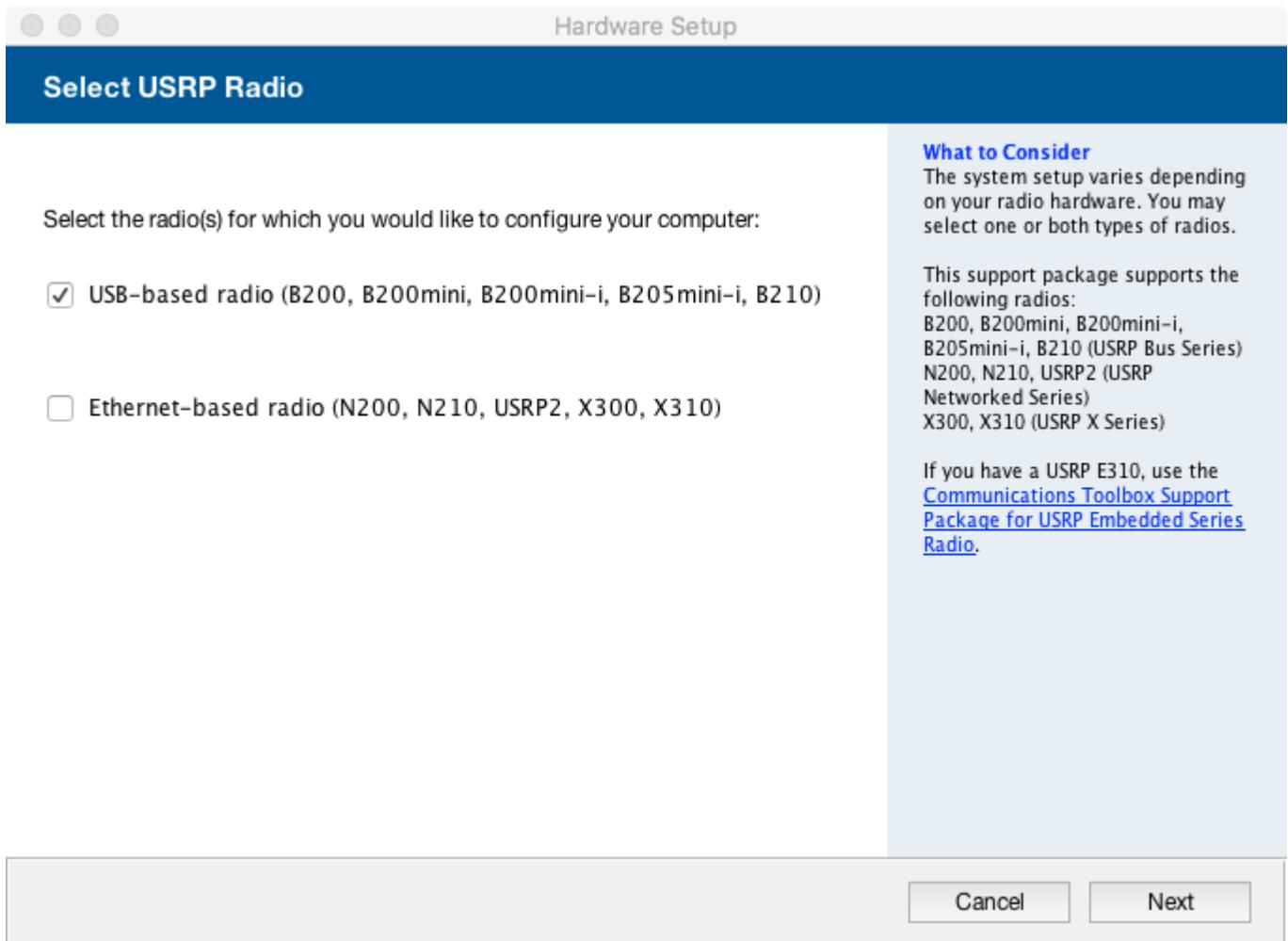
- 2 Nonroot accounts on Linux computer systems require `sudo` privileges to enable access to USB-based devices.
- 3 Udev rules must be installed or updated to permit access to USB-based devices by nonroot accounts. Click **Next** to continue.



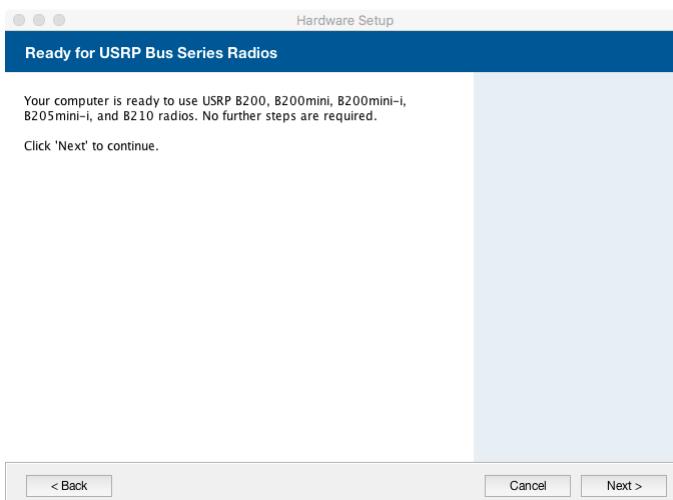
- 4 After enabling USB-based devices, you can confirm the host-to-radio communication link by testing the radio connection. For details, see "Test Radio Connection" on page 1-21.

## Install USB Driver for Macintosh

- 1 On the first Hardware Setup window for the installer, select **USB-based radio**, and then click **Next**.



- 2 The USB drivers are installed. Click **Next** to continue.



- 3 After enabling USB-based devices, you can confirm the host-to-radio communication link by testing the radio connection. For details, see "Test Radio Connection" on page 1-21.

## See Also

### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5

# Configure Host Computer for Ethernet-Based Radio Connection

## In this section...

["Host Computer Ethernet Options" on page 1-13](#)

["Configure Ethernet Connection Using Installer" on page 1-13](#)

These sections guide you through the instructions to configure a host computer for an Ethernet-based radio (N200, N210, X300, USRP2, X310) connection using the installer. To run the installer, see "Install Communications Toolbox Support Package for USRP Radio" on page 1-4.

**Note** If you are using a USB-based radio connection, see "Configure Host Computer for USB-Based Radio Connection" on page 1-7.

## Host Computer Ethernet Options

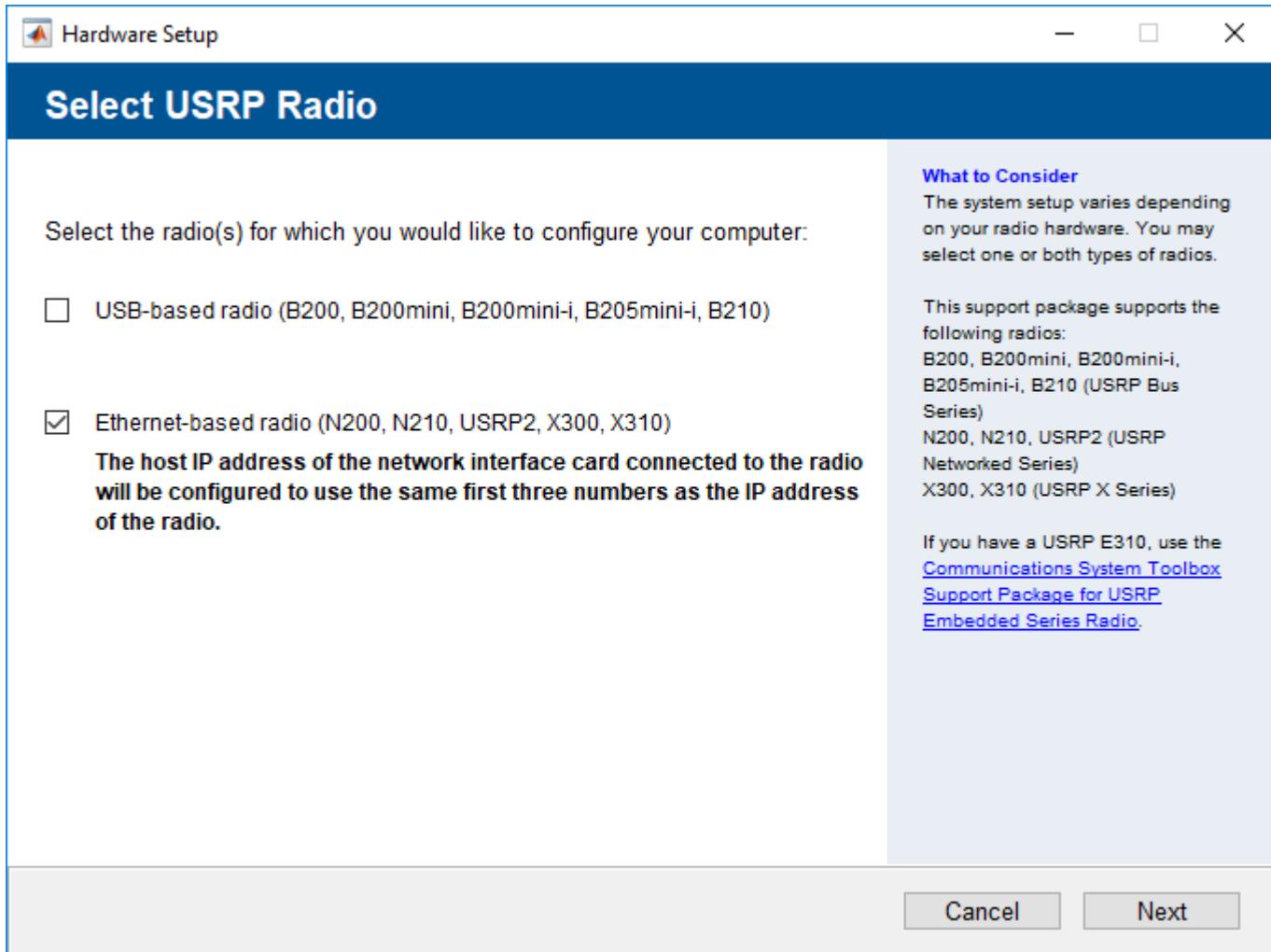
The Ethernet connection is often referred to as a network connection. You can use either an integrated network interface card (NIC) with a Gigabit Ethernet cable or a USB 3.0 Gigabit Ethernet adapter dongle. This connection is necessary for transmitting data, such as an FPGA or firmware image, from the host computer to the radio hardware. It is also necessary for sending and receiving signals to and from the radio hardware.

To have simultaneous internet access in the absence of a wireless connection, the host computer must have two Ethernet connections. If the host computer has only one Ethernet connection available, see "Using One Ethernet Port" on page 1-49.

## Configure Ethernet Connection Using Installer

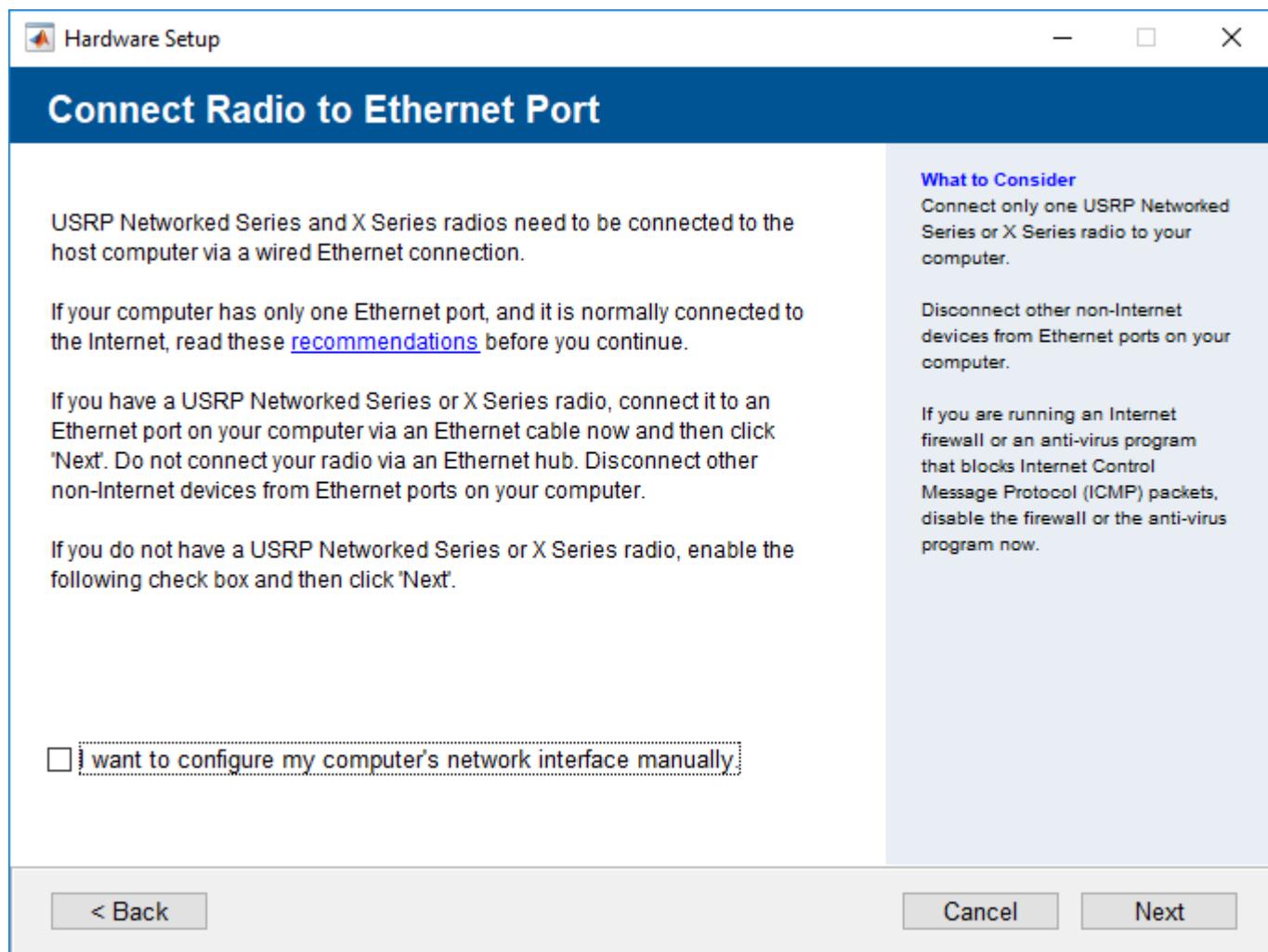
Unless otherwise noted, the installer instructions for configuring the Ethernet connection between the host and radio is the same for Windows, Linux, and Macintosh computers.

- 1 To have the installer guide you through the setup of the host-to-radio Ethernet connection, select **Ethernet-based radio**, and then click **Next**.

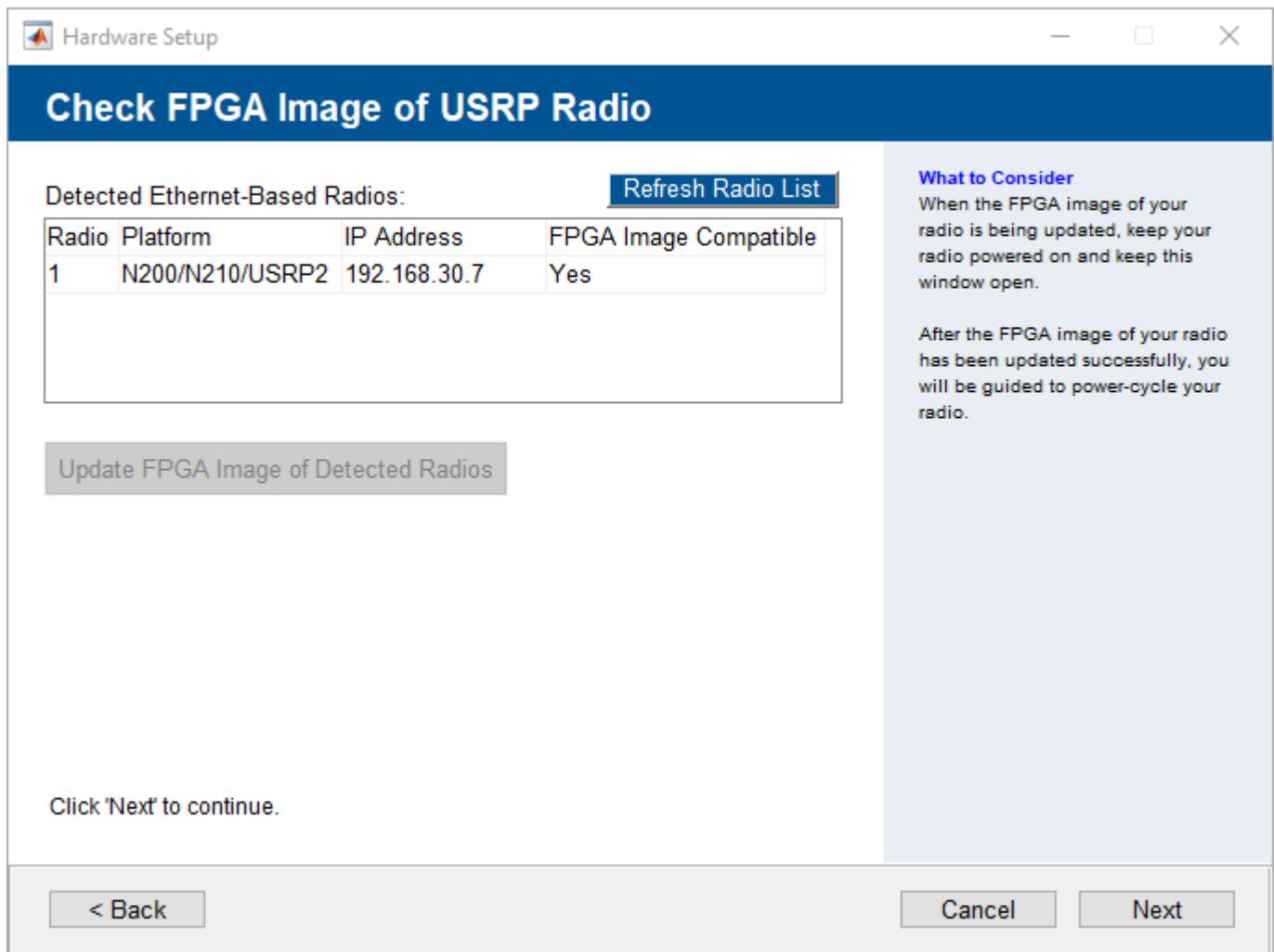


- 2** To have the installer configure the host-to-radio Ethernet connection, click **Next** on this screen. If your computer has only one NIC, see “Using One Ethernet Port” on page 1-49.

If you do not have a radio connected to the host computer, see “Configure Network Interface Using Installer with No Radio Connected” on page 1-19.



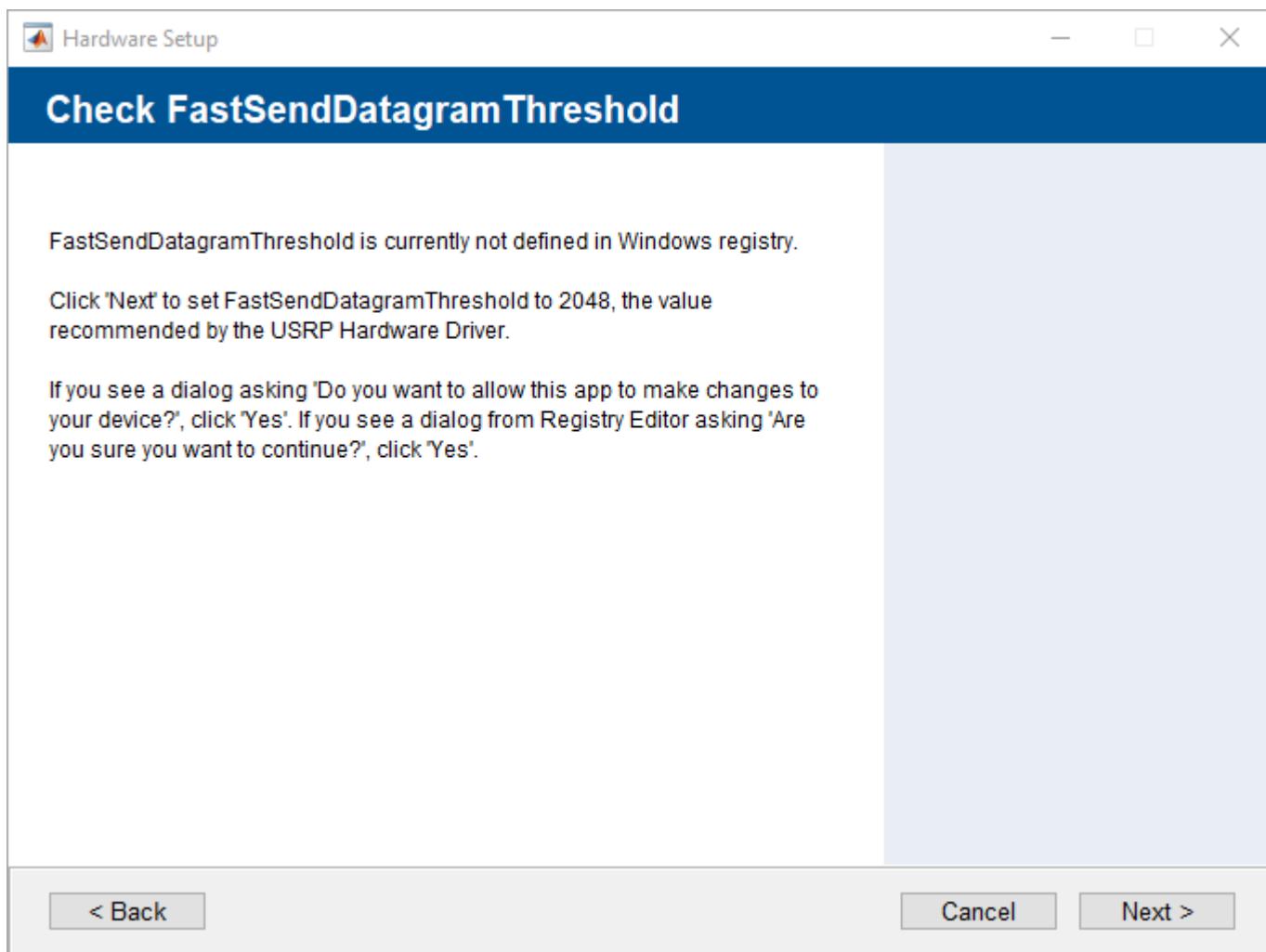
- 3 If the FPGA image of your radio is not compatible with the software version being installed, click **Update FPGA Image of Detected Radios** to update the radio FPGA image. After updating the image or to skip updating the image, click **Next** to continue.



**4 This step applies for Windows only.**

If `FastSendDataGramThreshold` is not defined in the Windows registry, the installer prompts you to set it to the value recommended by the USRP hardware driver from Ettus Research. To set `FastSendDataGramThreshold` to the recommended value, click **Next**.

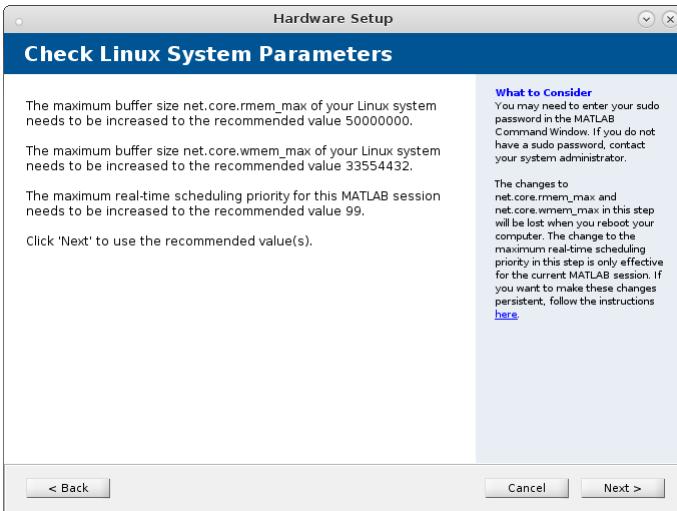
If you see a dialog box asking **Do you want to allow this app to make changes to your device?**, click **Yes** to continue. If you see a dialog box from the Registry Editor asking **Do you want to allow this app to make changes to your device?**, click **Yes** to continue.



**5 This step applies for Linux only.**

Ettus Research recommends setting the maximum buffer sizes for `net.core.rmem_max`, `net.core.wmem_max`, and real-time scheduling as indicated. To use these recommended values, click **Next**.

These changes are not persistent. To retain these settings in your account, see “Make Changes Persistent on Linux” on page 1-50.



- 6 After enabling your Ethernet-based device, you can confirm the host-to-radio communication link by testing the radio connection. For details, see “Test Radio Connection” on page 1-21.

## See Also

## More About

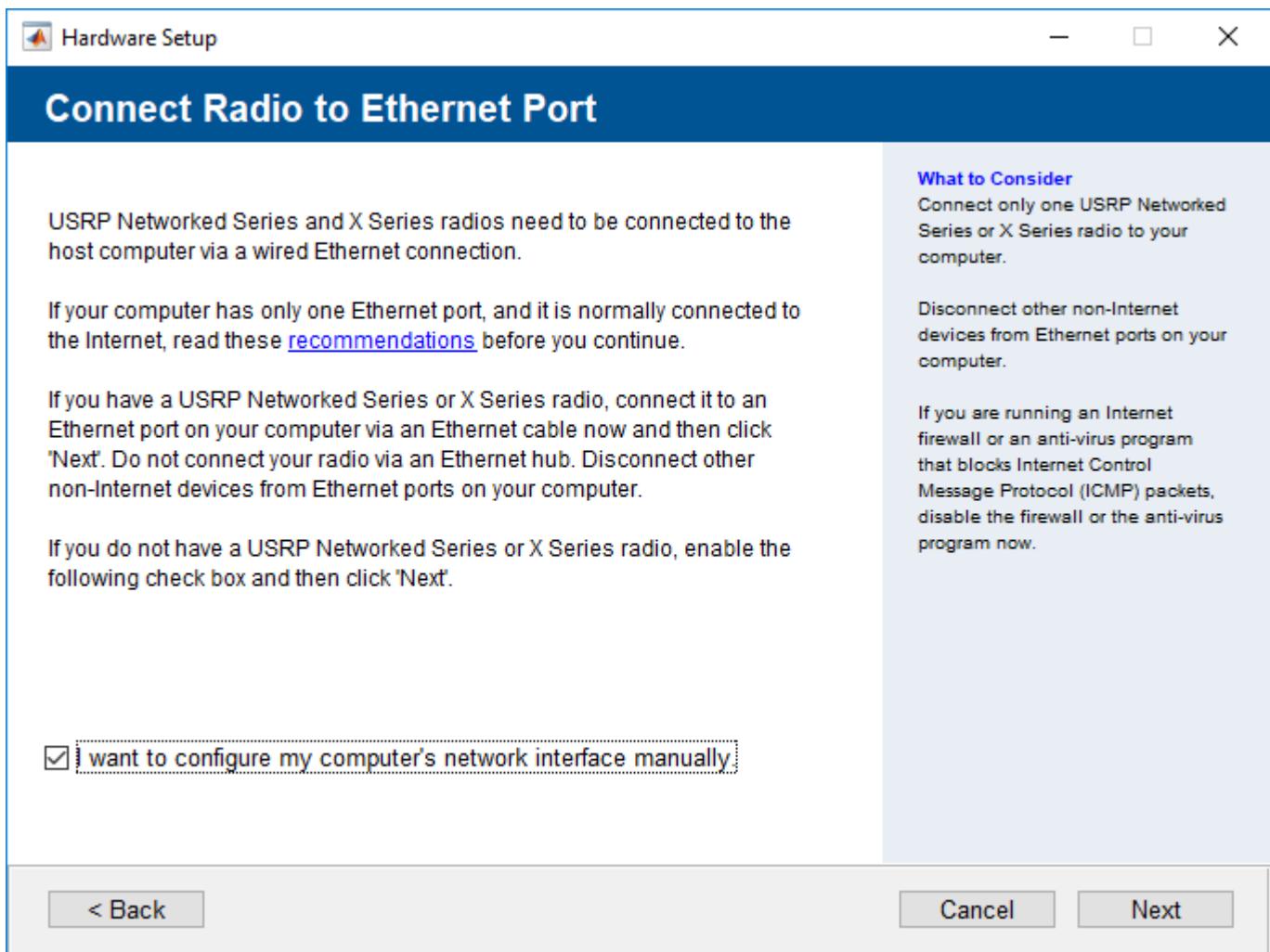
- “Guided USRP Radio Support Package Hardware Setup” on page 1-5

## Configure Network Interface Using Installer with No Radio Connected

These instructions describe installer steps to configure the host computer network interface with no radio connected. While making changes to a network interface, if you see a dialog box asking **Do you want to allow this app to make changes to your device?**, click **Yes** to continue.

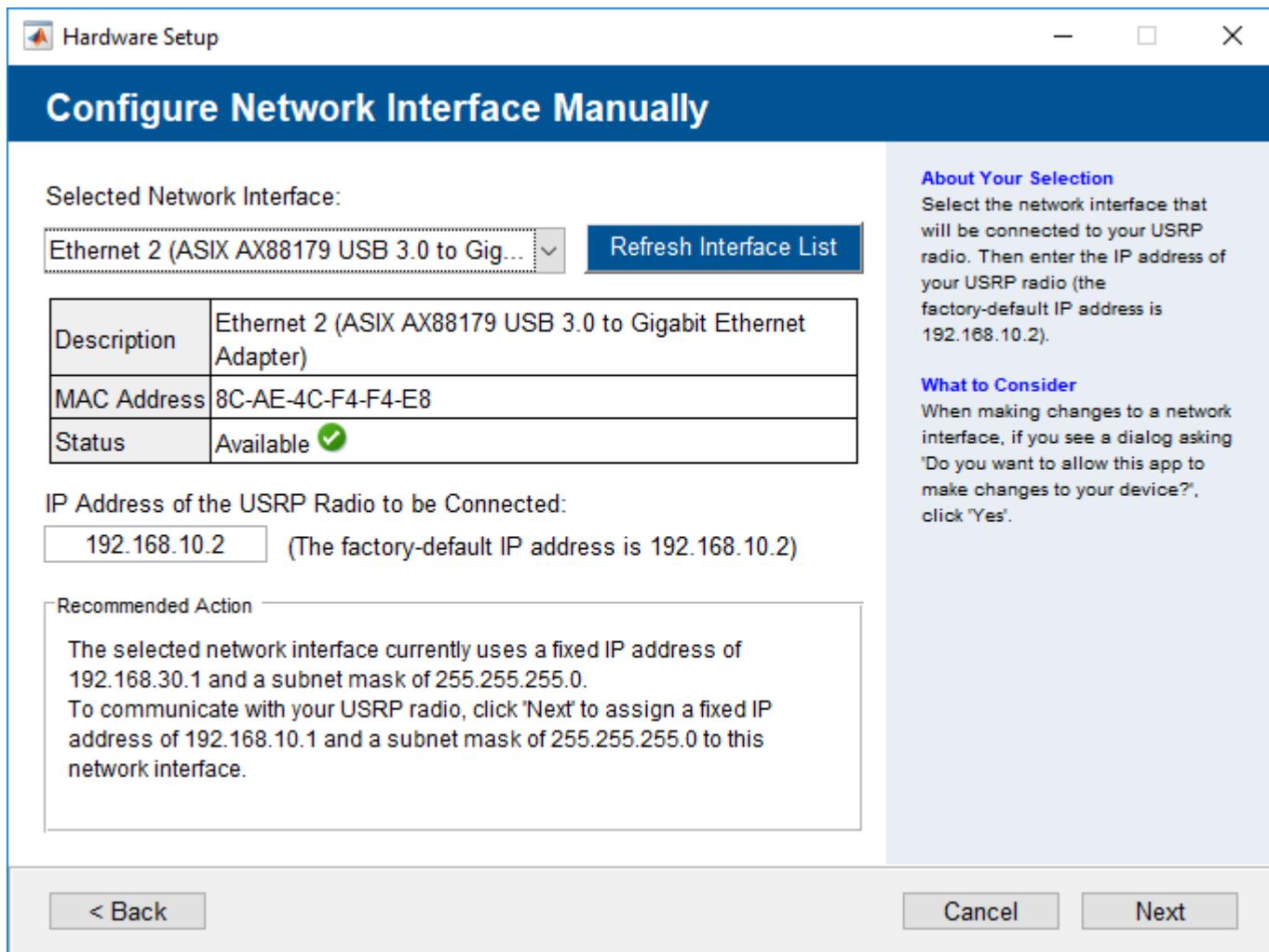
- 1 If you do not have an Ethernet-based radio connected to the host computer, select **I want to configure my computer's network interface manually**, before selecting **Next**. If your computer has only one NIC, see “Using One Ethernet Port” on page 1-49.

Selecting **I want to configure my computer's network interface manually** enables you to configure the host computer without connecting an Ethernet-based radio.



- 2 Select the network interface you want to configure, confirm the **Status** indicates Available, and then enter the IP address of your radio.

Click **Next** to continue.



- 3** Proceed to the FPGA image screen in the “Configure Ethernet Connection Using Installer” on page 1-13 instructions.

## See Also

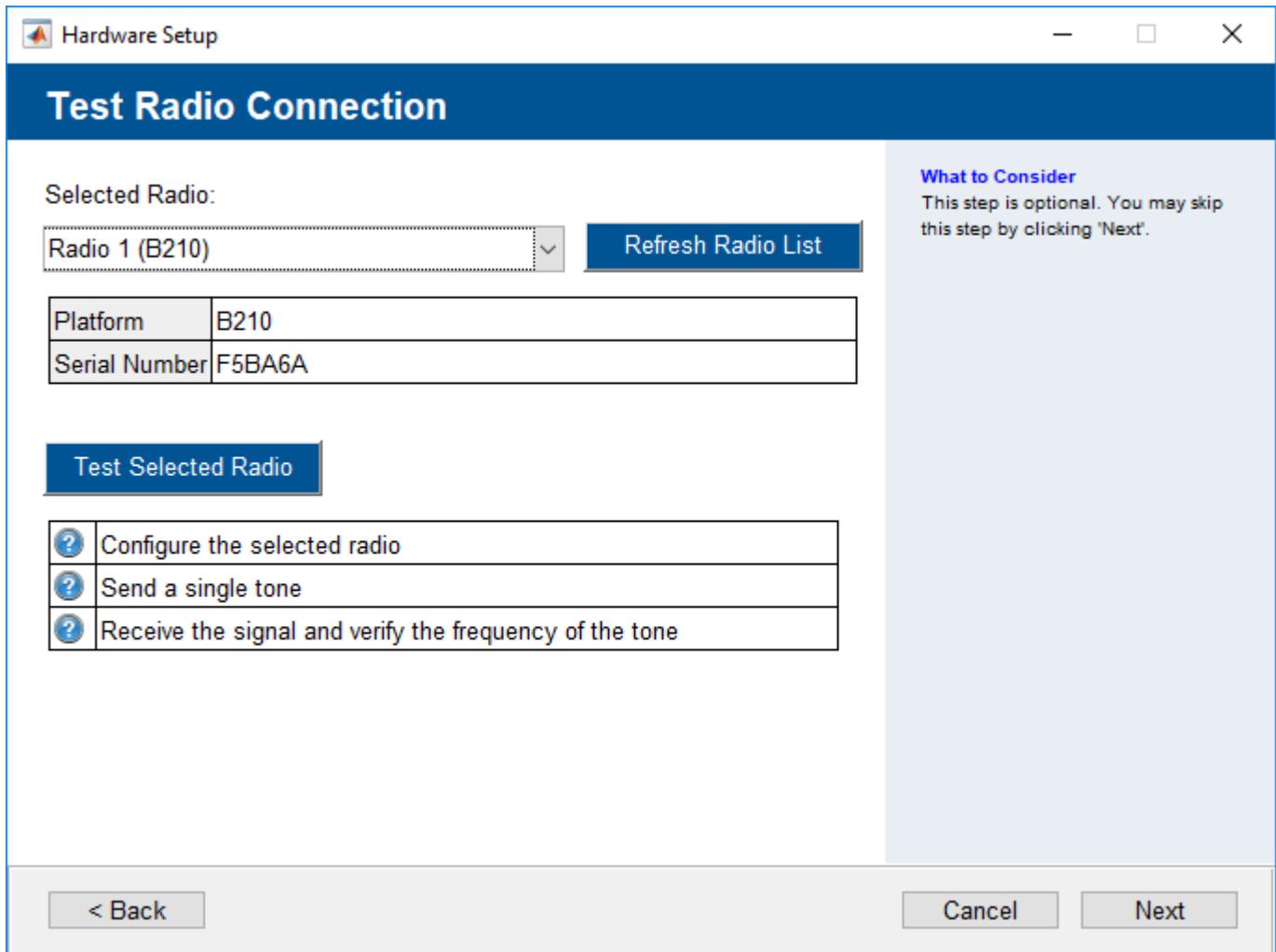
### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5

## Test Radio Connection

The installer performs the same radio connection test on Windows, Linux, and Macintosh computers. To run the installer, see “Install Communications Toolbox Support Package for USRP Radio” on page 1-4.

- 1 After setting up the host-to-radio connection, you can test the radio connection. Confirm **Selected Radio**, and then click **Test Selected Radio** button.



- 2 After testing the radio connection, verify that MATLAB can communicate with your radio using the Communications Toolbox Support Package for USRP Radio. For details, see “Verify MATLAB Connection to USRP Radio” on page 1-22.

## See Also

### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5

## Verify MATLAB Connection to USRP Radio

Follow these instructions to verify that MATLAB can communicate with your radio using the support package.

If you get a successful status, it means that MATLAB can communicate with the radio and the radio is ready to be used. In MATLAB, find and report the status of all USRP radios connected to the host computer by using the `findsdr` function.

```
radios = findsdr
```

The output variable, `radios`, is a structure that contains information on the USRP radios connected to the host computer. If the function finds one or more radios, it returns an array of structures.

Depending on the connection and radio type, the fields of the structure or array of structures are populated similar to this example.

- Ethernet-connected N3xx radios:

```
Platform: 'N310'  
IPAddress: '192.168.10.2'  
SerialNum: '315A36C'  
Status: 'Success'
```

- Ethernet-connected radios:

```
Platform: 'X310'  
IPAddress: '192.168.10.2'  
SerialNum: 'F4BF0D'  
Status: 'Success'
```

```
Platform: 'N200/N210/USRP2'  
IPAddress: '192.168.20.2'  
SerialNum: '873'  
Status: 'Success'
```

- USB-connected radio:

```
Platform: 'B200'  
IPAddress: ''  
SerialNum: 'ECR04ZDBT'  
Status: 'Success'
```

The function returns a successful status for a radio if MATLAB can communicate with that radio. Once MATLAB can communicate with a radio, it is ready to be used with features in the Communications Toolbox Support Package for USRP Radio.

If the `findsdr` function cannot find a radio, MATLAB returns an empty `IPAddress` or an empty `SerialNum` field, or a status other than `Success`. For possible causes and solution, see “Common Problems and Fixes” on page 2-2.

## See Also

### More About

- “Install Communications Toolbox Support Package for USRP Radio” on page 1-4

## Check Radio Firmware

The UHD firmware on the USRP radio hardware must match the UHD firmware on the host computer. You can find the latest supported UHD firmware version in the current Release Notes. To download the UHD firmware for Ethernet-based USRP N3xx radios, see “Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection” on page 1-57. Alternatively you can download the UHD firmware from <https://files.ettus.com/binaries/cache/n3xx/meta-ettus-v3.15.0.0/>.

To check the UHD firmware version loaded on your radio, use the `getSDRDriverVersion` function.

If your radio requires a UHD firmware upgrade, see “USRP Radio Firmware Update” on page 1-53.

## See Also

### More About

- “Install Communications Toolbox Support Package for USRP Radio” on page 1-4

## Manual USRP Radio Support Package Hardware Setup

To use Communications Toolbox Support Package for USRP Radio features, you must establish communication between the host computer and the radio hardware. For details on establishing and verifying the connection between the host computer and the radio, see the related topics. Each operating system (OS) has instructions specific to that OS. Choose the instructions for the OS on the host computer.

- The host computer can be a desktop or a laptop.
- For USB-based radios (Ettus Research Bus Series radios - B200, B200mini, B200mini-i, B205mini-i, or B210), the host computer must have at least one available USB port to connect to the radio.
- For Ethernet-based radios (Ettus Research Networked or X Series radios - N200, N210, USRP2, X300, or X310), the host computer must contain at least one dedicated Gigabit network interface card (NIC) for connecting to the USRP radio.
  - When connecting to an Ethernet-based radio, a second NIC is recommended for the host computer to remain simultaneously connected to the radio and a network (or the Internet). Alternatively, you can use one NIC for the radio connection and WiFi to connect to a network. If the host computer has no WiFi and only one NIC, see “Using One Ethernet Port” on page 1-49.
  - Directly connect your USRP radio to the NIC on the host computer with an Ethernet cable.
  - USRP radios have a factory default IP address of 192.168.10.2. This value is used for the radio IP address throughout the setup instructions. If your radio has a different IP address, modify accordingly.
- For Ethernet-based USRP N3xx radios (Ettus Research Networked Series radios - N300, N310, N320, or N321), the host computer must contain at least one dedicated 1 or 10 Gigabit network interface card (NIC) for connecting to the USRP radio.
  - Make sure you have at least one microSD card reader and one writable microSD card. The SD card should have at least 16GB storage. If the host computer does not have an integrated card reader, use an external USB SD card reader.
  - When connecting to an Ethernet-based USRP N3xx radio, a second NIC is recommended for the host computer to remain simultaneously connected to the radio and a network (or the Internet). Alternatively, you can use one NIC for the radio connection and WiFi to connect to a network. If the host computer has no WiFi and only one NIC, see “Using One Ethernet Port” on page 1-49.
  - Directly connect your USRP radio to the NIC on the host computer with an Ethernet cable.
  - USRP radios have a factory default IP address of 192.168.10.2. This value is used for the radio IP address throughout the setup instructions. If your radio has a different IP address, modify accordingly.

To install the Communications Toolbox Support Package for USRP Radio software and configure the host computer, use the recommended installer by following the “Install Communications Toolbox Support Package for USRP Radio” on page 1-4 directions.

After completing the Communications Toolbox Support Package for USRP Radio installation, if you choose to manually set up the connection between the host and the radio, see the related topics included here.

## See Also

### More About

- “Configure Ethernet Connection Manually on Windows 10” on page 1-26
- “Configure Ethernet Connection Manually on Linux” on page 1-34
- “Configure Ethernet Connection Manually on Mac” on page 1-39
- “Verify Hardware Connection” on page 1-44
- “Check Radio Firmware” on page 1-23

## Configure Ethernet Connection Manually on Windows 10

### In this section...

"Configure Ethernet Connection Via Windows Network Connections App" on page 1-26

"Configure Ethernet Connection Via Windows Command Prompt" on page 1-30

"Configure FastSendDatagramThreshold Registry Key" on page 1-32

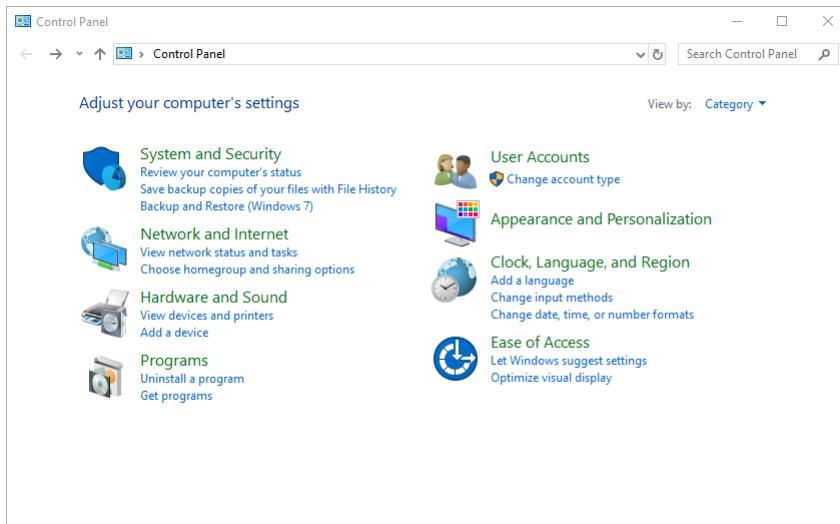
After running the Hardware Setup steps of the installer, if the radio is still not detected, you can attempt to configure the network interface manually via the Windows Network Connections App or the command prompt.

### Configure Ethernet Connection Via Windows Network Connections App

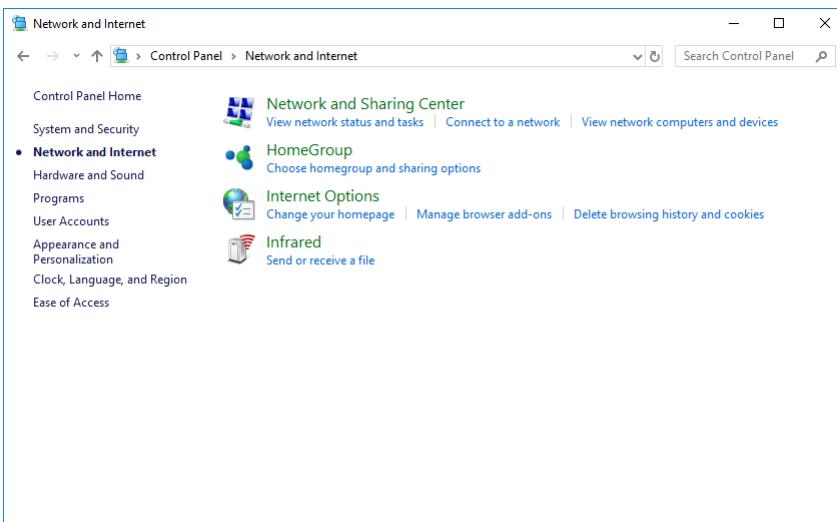
**1** Open the **Control Panel** from the Windows icon in the lower left corner of your monitor.

In the Control Panel window, make sure **View by** is set to **Category**.

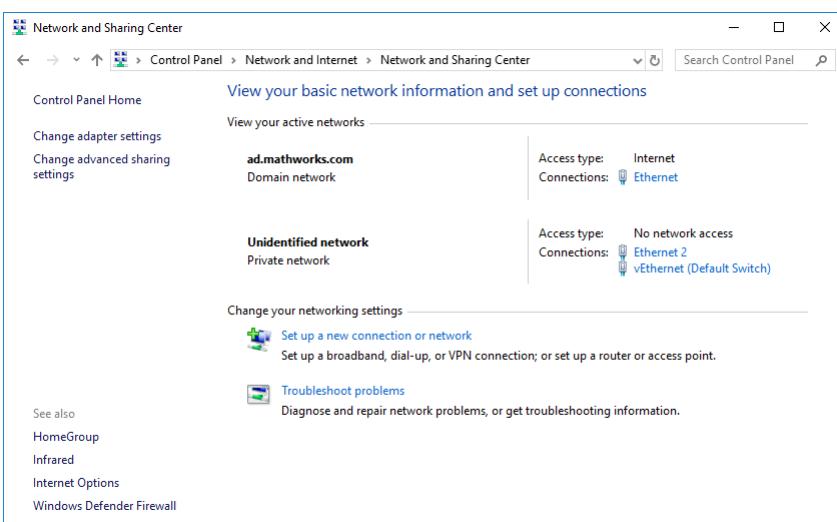
**2** Click **Network and Internet**.



**3** Click **Network and Sharing Center**.



- Click Change adapter settings in the left pane.

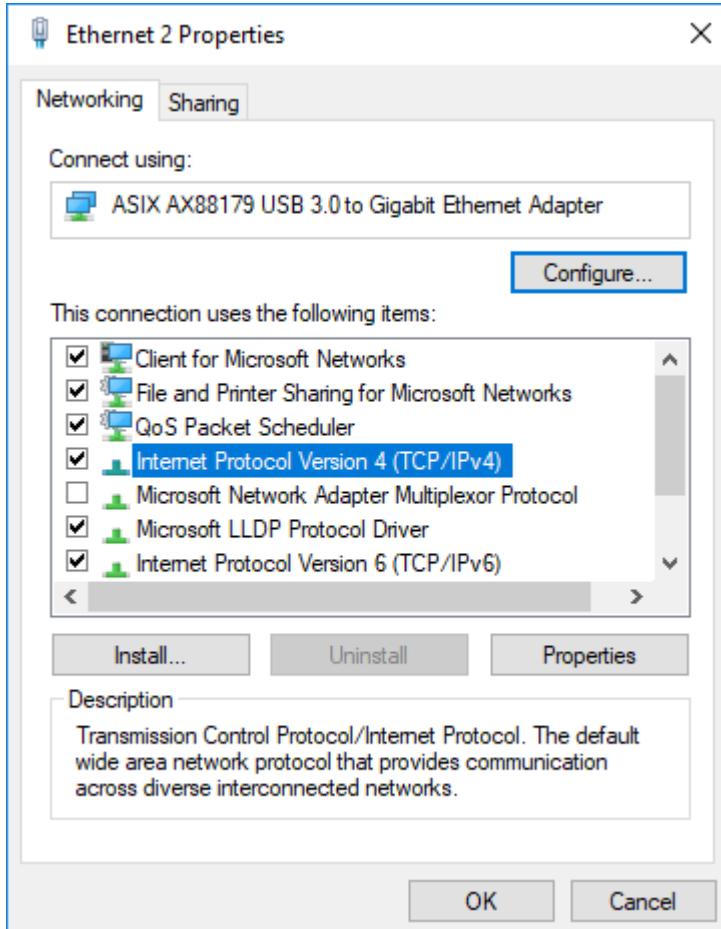


- Sort the adapters by clicking the More options button, and then selecting Details. Double-click the adapter to configure to open its properties.

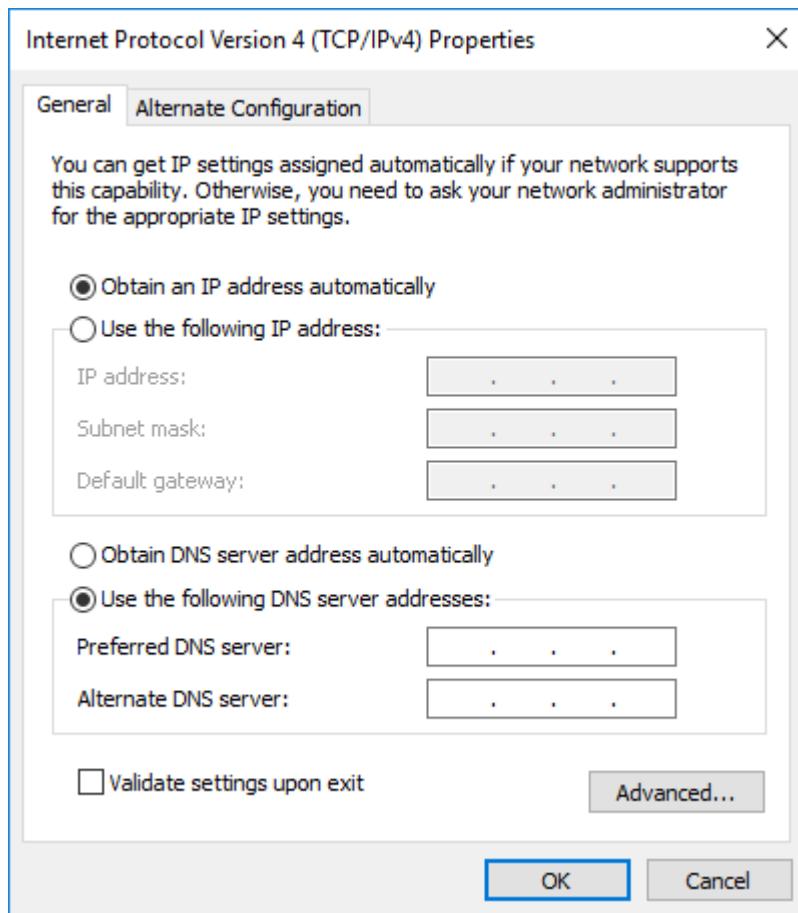
Name	Status	Device Name	Connectivity	Network Category	Owner	Type	Phone # or Host Address
Ethernet	ad.mathworks.com	Intel(R) Ethernet Connection (2) I218-LM	Internet access	Domain network	System	LAN or High-Speed Internet	
Ethernet 2	Unidentified network	ASIX AX88179 USB 3.0 to Gigabit Ethernet Adapter	No network access	Private network	System	LAN or High-Speed Internet	
vEthernet (Default Switch)	Unidentified network	Hyper-V Virtual Ethernet Adapter	No network access	Private network	System	LAN or High-Speed Internet	

On this computer, the adapter with the name **Ethernet** indicates connectivity to the Internet, but the connectivity of **Ethernet 2** indicates No network access. Double-click the **Ethernet 2** adapter to open its properties.

- 6** On the **Networking** tab, clear **Clients for Microsoft Networks** and **File and Printer Sharing for Microsoft Networks**. These services can cause intermittent connection problems with USRP radios. To configure the IP address, double-click **Internet Protocol Version 4 (TCP/IPv4)**.

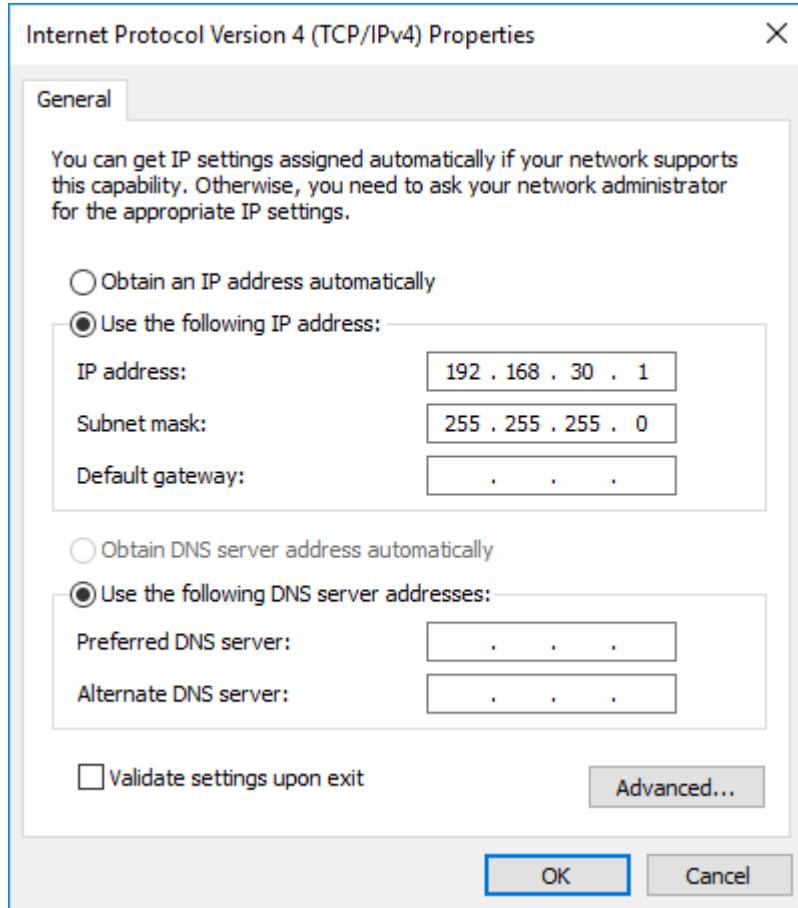


- 7** On the **General** tab, the default setting is typically set to **Obtain an IP address automatically**.



Select **Use the following IP Address**. Set the host **IP address** to 192.168.10.*X*, where *X* is any integer from 1 to 255 *except* 2. Leave the **Subnet mask** set to its default (255.255.255.0).

If your radio is on another subnet, meaning the first three octets of the IP address field are not 192.168.30, then enter the IP address values of your radio for the first three octets. See “Check Subnet Values on Host and Radio” on page 1-46.



- 8** Click **OK**.
- 9** The host computer configuration is complete. Continue to “Verify Hardware Connection” on page 1-44.

## Configure Ethernet Connection Via Windows Command Prompt

Use these commands to determine the IP address of the radio. After determining the IP address for the radio, update the network interface IP address, and ping the radio to verify that the host-to-radio connection is working.

- 1** View the list of network interfaces, by entering this command at the Windows command prompt.

```
C:\>getmac /V /FO CSV
```

For the host computer used in this example, this is the output.

```
"Connection Name","Network Adapter","Physical Address","Transport Name"  
"Ethernet","Intel(R) Ethernet Connection (2) I218-LM","6C-0B-84-A9-7F-FE",  
"\Device\Tcpip_{24EC4E6B-0552-4055-995C-3A404E3FE21F}"  
"vEthernet (Default Switch)","Hyper-V Virtual Ethernet Adapter","02-15-03-00-A9-6A",  
"\Device\Tcpip_{9BE08276-A58E-43EF-B928-0B215F3169A3}"  
"Ethernet 2","ASIX AX88179 USB 3.0 to Gigabit Ethernet Adapter","8C-AE-4C-F4-F4-E8",  
"Media disconnected"
```

- 2** Determine which Ethernet adapter is used for the internet connection by pinging the IP address associated with each *Connection Name* returned. When you identify the connection name of the

network adapter connected to the Internet, avoid selecting that network adapter for the radio. For more information, see "Host Computer Ethernet Options" on page 1-13.

The host computer in this example uses the "Ethernet" connection name for the Internet connection. Run the `netsh` command with this syntax to get the IP address of "Ethernet". Then `ping` that IP address to a website. Since the host computer uses this network adapter for its internet connection, reconfiguring this adapter to connect to the radio would jeopardize your ability to connect to the Internet. See "Using One Ethernet Port" on page 1-49.

```
C:\>netsh interface ip show address "Ethernet"

Configuration for interface "Ethernet"
  DHCP enabled: Yes
  IP Address: 172.21.18.178
  Subnet Prefix: 172.21.18.0/24 (mask 255.255.255.0)
  Default Gateway: 172.21.18.1
  Gateway Metric: 0
  InterfaceMetric: 25

C:\>ping -S "172.21.18.178" www.mathworks.com

Pinging ms-www.mathworks.com [144.212.244.17] from 172.21.18.178 with 32 bytes of data:
Reply from 144.212.244.17: bytes=32 time<1ms TTL=251
```

- 3 Next we run the `netsh` command specifying the "Ethernet 2" connection name. DHCP is typically enabled for an adapter that is not statically configured. Since the host computer does not use this network adapter for its internet connection, reconfiguring this adapter to connect to the radio, does not jeopardize your ability to connect to the Internet. That makes it a good adapter to use for the host-to-radio connection.

```
C:\>netsh interface ip show address "Ethernet 2"

Configuration for interface "Ethernet 2"
  DHCP enabled: Yes
  InterfaceMetric: 5

4 Connect the radio to this USB-Ethernet adapter. Wait for a few seconds, and then check the setting again. The response for a successful host-to-radio connection resembles the output shown here.

C:\>netsh interface ip show address "Ethernet 2"

Configuration for interface "Ethernet 2"
  DHCP enabled: Yes
  IP Address: 169.254.55.226
  Subnet Prefix: 169.254.0.0/16 (mask 255.255.0.0)
  InterfaceMetric: 25
```

The IP address 169.254.55.226 is a private IP address automatically assigned by Windows to the Ethernet adapter. To discover the IP address of your radio, ping the broadcast address 169.254.255.255. This address corresponds to the subnet prefix 169.254.0.0. The response for a successful ping resembles the output shown here.

```
C:\>ping -S 169.254.55.226 169.254.255.255

Pinging 169.254.255.255 from 169.254.55.226 with 32 bytes of data:
Reply from 192.168.30.7: bytes=32 time=1ms TTL=32
```

```
Reply from 192.168.30.7: bytes=32 time=1ms TTL=32
Reply from 192.168.30.7: bytes=32 time=2ms TTL=32
Reply from 192.168.30.7: bytes=32 time=1ms TTL=32

Ping statistics for 169.254.255.255:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
  Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 2ms, Average = 1ms
```

- 5 Based on the discovered radio IP address 192.168.30.7, assign an IP address to the Ethernet adapter from the same subnet, such as 192.168.30.1. Use a subnet mask of 255.255.255.0.

```
C:\>netsh interface ip set address "Ethernet 2" static 192.168.30.1 255.255.255.0
```

Verify that the updated IP address for the Ethernet adapter.

```
netsh interface ip show address "Ethernet 2"
```

```
Configuration for interface "Ethernet 2"
  DHCP enabled:      No
  IP Address:        192.168.30.1
  Subnet Prefix:     192.168.30.0/24 (mask 255.255.255.0)
  InterfaceMetric:   25
```

Verify that the host computer can ping the IP address assigned to the radio.

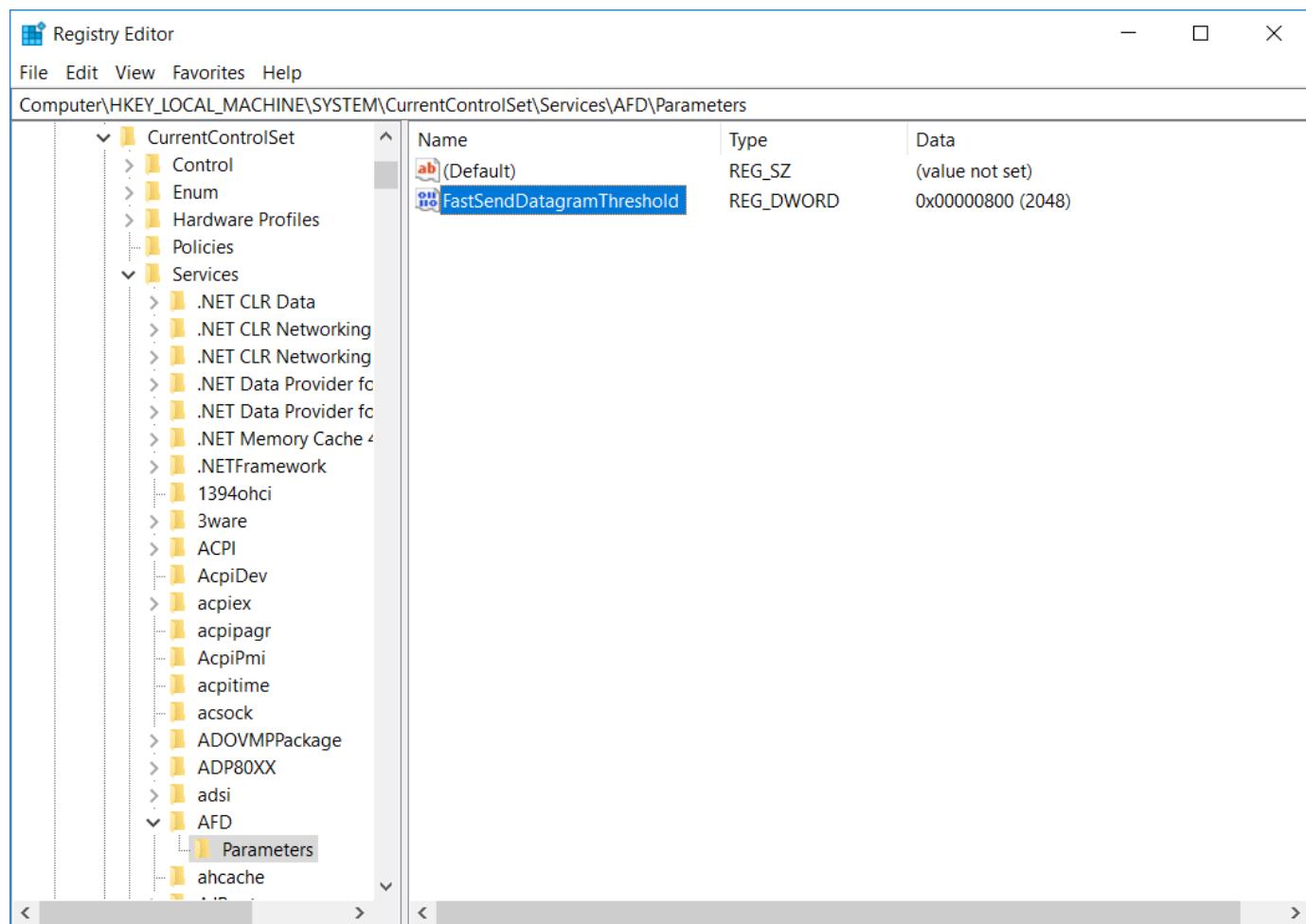
```
C:\>ping 192.168.30.7
```

```
Pinging 192.168.30.7 with 32 bytes of data:
Reply from 192.168.30.7: bytes=32 time=1ms TTL=32

Ping statistics for 192.168.30.7:
  Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
  Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 1ms, Average = 1ms
```

## Configure **FastSendDatagramThreshold** Registry Key

Ettus Research recommends configuring the *FastSendDatagramThreshold* with the value 2048 (decimal). Define or reconfigure the *FastSendDatagramThreshold* registry key in the Windows Registry Editor under HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\services\AFD\Parameters.



## See Also

### More About

- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

## Configure Ethernet Connection Manually on Linux

After running the Hardware Setup steps of the installer, if the radio is still not detected, you can attempt to configure the network interface manually by running commands in a terminal window.

### Configure Ethernet Connection Via Linux Command Prompt

To configure the network interface card (NIC) for your USRP radio via the Linux command prompt, use these instructions.

- 1 Set the correct host computer network interface IP address. USRP radios have a factory default IP address of 192.168.10.2. Leave the subnet mask set to its default (255.255.255.0).

If the USRP radio IP address is the default value of 192.168.10.2, run this shell command to set these values.

```
%sudo ifconfig ethX 192.168.10.Y netmask 255.255.255.0
```

`ethX` is the name of the host computer Ethernet port (usually `eth0`, `eth1`, etc.), and `Y` is any integer from 0 through 255 excluding 2. You might be required to enter a password to use the `sudo` command.

If your radio is on another subnet, meaning the first three octets of the IP address field are not 192.168.30, then enter the IP address values of your radio for the first three octets. See “Check Subnet Values on Host and Radio” on page 1-46.

This example shows this workflow in detail. Use these commands to determine the IP address of the radio. After determining the IP address for the radio, update the network interface IP address, and ping the radio to verify that the host-to-radio connection is working.

- a Run the `ipconfig` command to view the list of network interfaces. On Linux, some of the network interfaces shown by `ifconfig` might correspond to virtual network interfaces for virtual machines.

```
$ ifconfig

docker0    Link encap:Ethernet HWaddr 02:42:0a:d0:14:e4
           inet addr:192.168.0.10 Bcast:0.0.0.0 Mask:255.255.255.0
             UP BROADCAST MULTICAST MTU:1500 Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

eth0        Link encap:Ethernet HWaddr 6c:0b:84:65:8b:ab
           inet addr:172.21.152.206 Bcast:172.21.152.255 Mask:255.255.255.0
             inet6 addr: fe80::6e0b:84ff:fe65:8bab/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
             RX packets:285108152 errors:0 dropped:0 overruns:0 frame:0
             TX packets:153065910 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:240450151147 (223.9 GiB) TX bytes:44073818567 (41.0 GiB)
             Interrupt:20 Memory:fb100000-fb120000

eth1        Link encap:Ethernet HWaddr 8c:ae:4c:f4:f4:e5
           UP BROADCAST MULTICAST MTU:1500 Metric:1
             RX packets:6091 errors:0 dropped:0 overruns:0 frame:0
             TX packets:9345 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:1000
             RX bytes:459870 (449.0 KiB) TX bytes:1285142 (1.2 MiB)

lo          Link encap:Local Loopback
           inet addr:127.0.0.1 Mask:255.0.0.0
```

```

inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:6431593 errors:0 dropped:0 overruns:0 frame:0
TX packets:6431593 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:8072498091 (7.5 GiB) TX bytes:8072498091 (7.5 GiB)

vmnet1      Link encap:Ethernet HWaddr 00:50:56:c0:00:01
inet addr:172.16.37.1 Bcast:172.16.37.255 Mask:255.255.255.0
inet6 addr: fe80::250:56ff:fe01:64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:24326 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

vmnet8      Link encap:Ethernet HWaddr 00:50:56:c0:00:08
inet addr:192.168.138.1 Bcast:192.168.138.255 Mask:255.255.255.0
inet6 addr: fe80::250:56ff:fe08:64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:24326 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

- b** In most Linux distributions, you can find the physical Ethernet interfaces by checking the presence of symbolic links. For example, run `ls -l` to view the symbolic links assigned to `/sys/class/net/* device/driver`.

```
$ ls -l /sys/class/net/*	device/driver
lrwxrwxrwx 1 root root 0 Apr 23 16:41 /sys/class/net/eth0/device/driver
-> ../../bus/pci/drivers/e1000e/
lrwxrwxrwx 1 root root 0 Apr 24 14:31 /sys/class/net/eth1/device/driver
-> ../../../../../../bus/usb/drivers/ax88179_178a/
```

- c** Run `modinfo` to get more information about the device drivers `e1000e` and `ax88179_178a`.

```
$ modinfo -d e1000e
Intel(R) PRO/1000 Network Driver

$ modinfo -d ax88179_178a
ASIX AX88179/178A based USB 3.0/2.0 Gigabit Ethernet Devices
```

- d** In this case, `eth0` and `eth1` are real physical Ethernet interfaces. Ping these Ethernet interfaces using the broadcast address `255.255.255.255` to discover which interface the radio is attached to and to get the IP address of the radio.

```
$ ping -I eth1 -c 5 -b 255.255.255.255
WARNING: pinging broadcast address
ping: Warning: source address might be selected on device other than eth1.
PING 255.255.255.255 (255.255.255.255) from 172.21.152.206 eth1: 56(84) bytes of data.
64 bytes from 192.168.30.7: icmp_seq=1 ttl=32 time=1.15 ms
64 bytes from 192.168.30.7: icmp_seq=2 ttl=32 time=1.22 ms
64 bytes from 192.168.30.7: icmp_seq=3 ttl=32 time=1.16 ms
64 bytes from 192.168.30.7: icmp_seq=4 ttl=32 time=1.15 ms
64 bytes from 192.168.30.7: icmp_seq=5 ttl=32 time=1.20 ms

--- 255.255.255.255 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.154/1.181/1.227/0.028 ms
```

- e** In Ubuntu and some other Linux systems, you might need to assign a static IP address and a subnet mask to a network interface before you can discover the IP address of the radio. Use the `sudo` command to assign a static IP address.

View the network settings before and after running the `sudo` command.

```
$ ifconfig eth1
```

```
eth1      Link encap:Ethernet HWaddr 8c:ae:4c:f4:f4:e5
          UP BROADCAST MULTICAST MTU:1500 Metric:1
          RX packets:6091 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9345 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:459870 (449.0 KiB) TX bytes:1285142 (1.2 MiB)

$ sudo ifconfig eth1 192.168.10.1 netmask 255.255.0.0

$ ifconfig eth1

eth1      Link encap:Ethernet HWaddr 8c:ae:4c:f4:f4:e5
          inet addr:192.168.20.1 Bcast:192.168.255.255 Mask:255.255.0.0
          inet6 addr: fe80::8cae:4cff:fe4:f4e5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:10129 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13724 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:763398 (745.5 KiB) TX bytes:1733921 (1.6 MiB)
```

Ping eth1 at the broadcast address 192.168.255.255 to discover the IP address of the radio.

```
$ ping -I eth1 -c 5 -b 192.168.255.255
```

```
WARNING: pinging broadcast address
PING 192.168.255.255 (192.168.255.255) from 192.168.10.1 eth1: 56(84) bytes of data.
64 bytes from 192.168.30.7: icmp_seq=1 ttl=32 time=1.18 ms
64 bytes from 192.168.30.7: icmp_seq=2 ttl=32 time=1.17 ms
64 bytes from 192.168.30.7: icmp_seq=3 ttl=32 time=1.17 ms
64 bytes from 192.168.30.7: icmp_seq=4 ttl=32 time=1.22 ms
64 bytes from 192.168.30.7: icmp_seq=5 ttl=32 time=1.15 ms

--- 192.168.255.255 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 1.155/1.181/1.224/0.031 ms
```

- f If the static IP address of the network interface is the same as the IP address of the radio, ping does not reveal the IP address of the radio. Assign a different static IP address to the network interface, and then try the broadcast ping again.

For this example the radio and network interface both are assigned the IP address 192.168.10.1. Send a broadcast ping from eth1.

```
$ ping -I eth1 -c 5 -b 192.168.255.255
```

```
WARNING: pinging broadcast address
PING 192.168.255.255 (192.168.255.255) from 192.168.10.1 eth1: 56(84) bytes of data.

--- 192.168.255.255 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4031ms
```

Use the sudo command to update the IP address of the network interface to 192.168.20.1 with subnet mask 255.255.0.0, and then try the broadcast ping again.

```
$ sudo ifconfig eth1 192.168.20.1 netmask 255.255.0.0
$ ifconfig eth1
```

```
eth1      Link encap:Ethernet HWaddr 8c:ae:4c:f4:f4:e5
          inet addr:192.168.20.1 Bcast:192.168.255.255 Mask:255.255.0.0
          inet6 addr: fe80::8cae:4cff:fe4:f4e5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:10129 errors:0 dropped:0 overruns:0 frame:0
          TX packets:13724 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:763398 (745.5 KiB) TX bytes:1733921 (1.6 MiB)
```

```
$ ping -I eth1 -c 5 -b 192.168.255.255
```

```
WARNING: pinging broadcast address
PING 192.168.255.255 (192.168.255.255) from 192.168.20.1 eth1: 56(84) bytes of data.
```

```

64 bytes from 192.168.10.1: icmp_seq=1 ttl=32 time=1.18 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=32 time=1.19 ms
64 bytes from 192.168.10.1: icmp_seq=3 ttl=32 time=1.17 ms
64 bytes from 192.168.10.1: icmp_seq=4 ttl=32 time=1.27 ms
64 bytes from 192.168.10.1: icmp_seq=5 ttl=32 time=1.16 ms

--- 192.168.255.255 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 1.164/1.197/1.272/0.058 ms

```

- g** Choose a static IP address in the same subnet of your radio (192.168.30.1 in this example), and use a subnet mask 255.255.255.0. Use the `sudo` command to update the network interface IP address, use the `ipconfig` command to check the configuration, and ping the radio at 192.168.30.7 to verify the host-to-radio connection. In this example the ping was successful.

```

$ sudo ifconfig eth1 192.168.30.1 netmask 255.255.255.0
$ ifconfig eth1

eth1      Link encap:Ethernet HWaddr 8c:ae:4c:f4:f4:e5
          inet addr:192.168.30.1 Bcast:192.168.30.255 Mask:255.255.255.0
          inet6 addr: fe80::8cae:4cff:fe4:f4e5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:6123 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9463 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:462584 (451.7 KiB) TX bytes:1303558 (1.2 MiB)

ping -c 5 192.168.30.7

PING 192.168.30.7 (192.168.30.7) 56(84) bytes of data.
64 bytes from 192.168.30.7: icmp_seq=1 ttl=32 time=1.11 ms
64 bytes from 192.168.30.7: icmp_seq=2 ttl=32 time=1.25 ms
64 bytes from 192.168.30.7: icmp_seq=3 ttl=32 time=1.20 ms
64 bytes from 192.168.30.7: icmp_seq=4 ttl=32 time=1.19 ms
64 bytes from 192.168.30.7: icmp_seq=5 ttl=32 time=1.15 ms

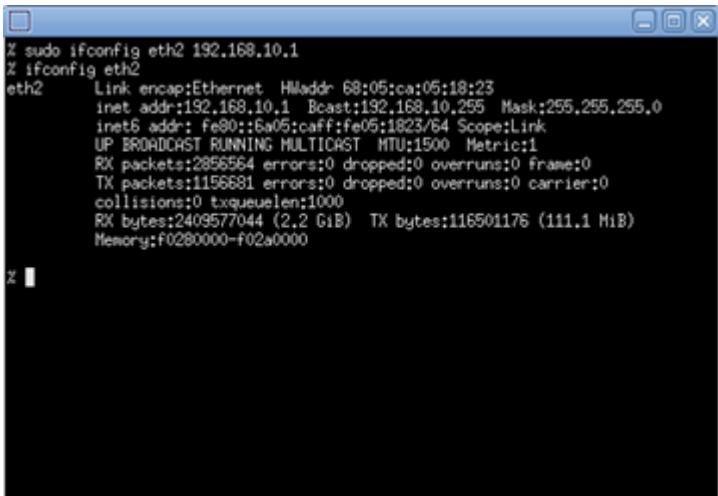
--- 192.168.30.7 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.119/1.183/1.251/0.063 ms

```

- 2** Run these shell commands to configure the network connection, and then check that the changes took effect.

```
%sudo ifconfig eth X 192.168.Y.Z
ifconfig ethX
```

For 192.168.Y.Z use the actual IP address to be assigned to the host computer, and for ethX use the name of the host Ethernet port (usually eth0, eth1, etc.).



A screenshot of a terminal window titled 'Terminal'. The window contains the following text:

```
x sudo ifconfig eth2 192.168.10.1
x ifconfig eth2
eth2      Link encap:Ethernet  HWaddr 68:05:ca:06:18:23
          inet  addr: 192.168.10.1  Bcast:192.168.10.255  Mask:255.255.255.0
                  inet6 addr: fe80::6a05:caff:fe05:1823/64 Scope:Link
                      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                      RX packets:2056564 errors:0 dropped:0 overruns:0 frame:0
                      TX packets:115681 errors:0 dropped:0 overruns:0 carrier:0
                      collisions:0 txqueuelen:1000
                      RX bytes:2409577044 (2.2 GiB)  TX bytes:116501176 (111.1 MiB)
                      Memory:f0280000-f02a0000

x |
```

- 3** On Linux systems, unless you make network connection changes persistent, a system reboot resets network connection changes, and the host-to-radio connection is lost. To retain the host-to-radio connection after a computer reboot, see “Make Changes Persistent on Linux” on page 1-50.
- 4** The host computer configuration is complete. Continue to “Verify Hardware Connection” on page 1-44.

## See Also

### More About

- “Make Changes Persistent on Linux” on page 1-50
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

# Configure Ethernet Connection Manually on Mac

## In this section...

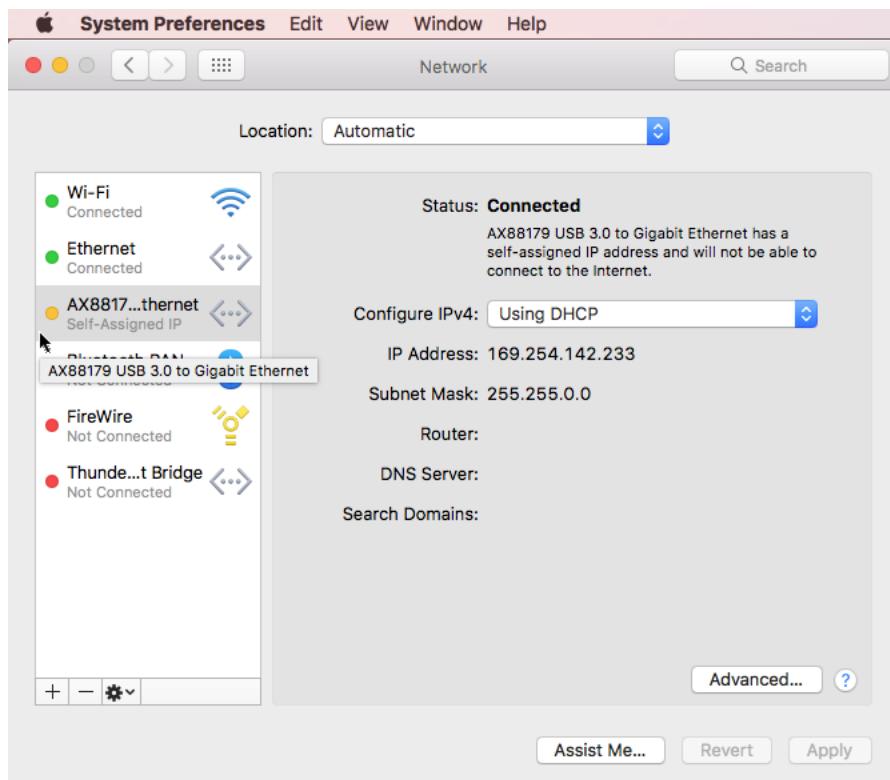
"Configure Ethernet Connection Via Mac System Preferences" on page 1-39

"Configure Ethernet Connection Via Mac Terminal Window" on page 1-40

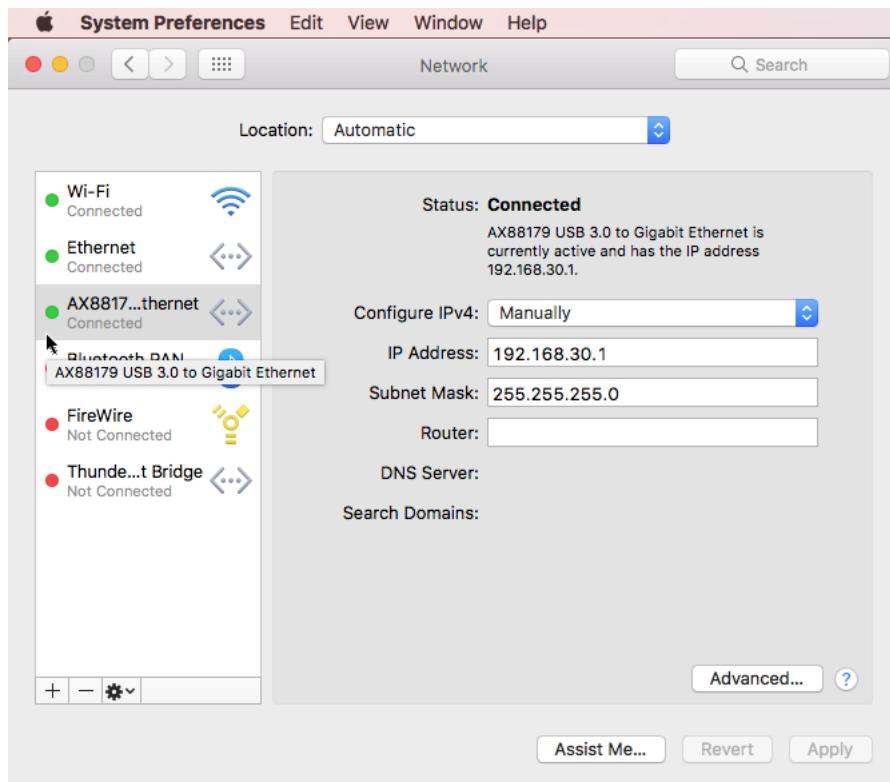
After running the Hardware Setup steps of the installer, if the radio is still not detected, you can attempt to configure the network interface manually via the Mac System Preferences or running commands in a terminal window.

## Configure Ethernet Connection Via Mac System Preferences

- 1 Click **Network** from **System Preferences**. From the left pane, select the network connection that is connected to the USRP radio. If your Mac has only one network interface card (NIC), see "Using One Ethernet Port" on page 1-49.



- 2 The default IPV4 configuration is **Using DHCP**. The default IP address 169.254.142.233 is a private IP address automatically assigned by the operating system. For more information, see <https://tools.ietf.org/html/rfc3927>.
- 3 Set **Configure IPv4** to **Manually**. Set **IP Address** to 192.168.10.X, where X is any integer from 1 to 255 except 2, and **Subnet Mask** to 255.255.255.0. Click **Apply**.



If your radio is on another subnet, meaning the first three octets of the IP address field are not 192.168.30, then enter the IP address values of your radio for the first three octets.

- 4 The host computer configuration is complete. Continue to “Verify Hardware Connection” on page 1-44.

## Configure Ethernet Connection Via Mac Terminal Window

Use these commands to determine the IP address of the radio. After determining the IP address for the radio, update the network interface IP address, and ping the radio to verify that the host-to-radio connection is working.

- 1 View the list of network services, by entering this command at the Mac command prompt.

```
$ networksetup -listallnetworkservices
```

For the host computer used in this example, this is the output.

```
An asterisk (*) denotes that a network service is disabled.  
Wi-Fi  
Ethernet  
AX88179 USB 3.0 to Gigabit Ethernet  
FireWire  
Bluetooth PAN  
Thunderbolt Bridge
```

- 2 Determine which Ethernet adapter is used for the internet connection by pinging the IP address associated with each network service returned. When you identify the connection name of the network adapter connected to the Internet, avoid selecting that network adapter for the radio. For more information, see “Using One Ethernet Port” on page 1-49.

- 3** For this example, the radio is attached to the AX88179 USB 3.0 to Gigabit Ethernet. Run this command to find the IP address and Ethernet address associated with AX88179 USB 3.0 to Gigabit Ethernet.

```
$ networksetup -getinfo "AX88179 USB 3.0 to Gigabit Ethernet"
```

```
DHCP Configuration
IP address: 169.254.142.233
Subnet mask: 255.255.0.0
Router: (null)
Client ID:
IPv6: Automatic
IPv6 IP address: none
IPv6 Router: none
Ethernet Address: 8c:ae:4c:f4:f4:e5
```

- 4** Run the `ifconfig` command, and find the network interface corresponding to the Ethernet address 8c:ae:4c:f4:f4:e5.

```
$ ifconfig
```

```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
    inet 127.0.0.1 netmask 0xffff0000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
            nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=10b<RXCSUM,TXCSUM,VLAN_HWTAGGING,AV>
    ether 0c:4d:e9:b8:28:aa
    inet6 fe80::1431:delf:143c:e8fe%en0 prefixlen 64 secured scopeid 0x4
        inet 172.21.152.123 netmask 0xffffffff broadcast 172.21.152.255
            nd6 options=201<PERFORMNUD,DAD>
            media: autoselect (1000baseT <full-duplex,flow-control,energy-efficient-ethernet>)
            status: active
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ether a8:8e:24:a2:10:6d
    inet6 fe80::140a:c54f:c99a:68a6%en1 prefixlen 64 secured scopeid 0x6
        inet 172.31.205.144 netmask 0xffffffff broadcast 172.31.205.255
            nd6 options=201<PERFORMNUD,DAD>
            media: autoselect
            status: active
fw0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 4078
    lladdr 28:0b:5c:ff:fe:15:fc:32
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect <full-duplex>
    status: inactive
en2: flags=963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX> mtu 1500
    options=60<TS04,TS06>
    ether 32:00:11:5f:c3:20
    media: autoselect <full-duplex>
    status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
    ether 0a:8e:24:a2:10:6d
    media: autoselect
    status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
    ether de:27:8b:39:9e:74
    inet6 fe80::dc27:8bff:fe39:9e74%awdl0 prefixlen 64 scopeid 0xa
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
bridge0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=63<RXCSUM,TXCSUM,TS04,TS06>
    ether 32:00:11:5f:c3:20
    Configuration:
        id 0:0:0:0:0:0 priority 0 hellotime 0 fwddelay 0
        maxage 0 holdcnt 0 proto stp maxaddr 100 timeout 1200
        root id 0:0:0:0:0:0 priority 0 ifcost 0 port 0
        ipfilter disabled flags 0x2
    member: en2 flags=3<LEARNING,DISCOVER>
        ifmaxaddr 0 port 8 priority 0 path cost 0
```

```
nd6 options=201<PERFORMNUD,DAD>
media: <unknown type>
status: inactive
utun0: flags=8051<UP,POINTOPOINT,RUNNING,MULTICAST> mtu 2000
    inet6 fe80::7e23:4719:93e:b73c%utun0 prefixlen 64 scopeid 0xc
        nd6 options=201<PERFORMNUD,DAD>
en4: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=b<RXCSUM,TXCSUM,VLAN_HWTAGGING>
    ether 8c:ae:4c:f4:f4:e5
    inet6 fe80::1433:8cb7:fc90:4c37%en4 prefixlen 64 secured scopeid 0xd
        inet 169.254.142.233 netmask 0xfffff000 broadcast 169.254.255.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect (1000baseT <full-duplex,flow-control>)
        status: active
```

The output shows, you can see that **en4** is the network interface corresponding to the Ethernet address **8c:ae:4c:f4:f4:e5**, and its IP address is **169.254.142.233**.

- 5 Set the last two numbers of the IP address **169.254.142.233** to **255**, and send a broadcast ping from **en4**. The output for this example shows a successful ping.

```
$ ping -b en4 -c 5 169.254.255.255
```

```
PING 169.254.255.255 (169.254.255.255): 56 data bytes
64 bytes from 169.254.142.233: icmp_seq=0 ttl=255 time=0.137 ms
64 bytes from 192.168.30.7: icmp_seq=0 ttl=32 time=1.268 ms
64 bytes from 169.254.142.233: icmp_seq=1 ttl=255 time=0.292 ms
64 bytes from 192.168.30.7: icmp_seq=1 ttl=32 time=1.488 ms
64 bytes from 169.254.142.233: icmp_seq=2 ttl=255 time=0.273 ms
64 bytes from 192.168.30.7: icmp_seq=2 ttl=32 time=1.422 ms
64 bytes from 169.254.142.233: icmp_seq=3 ttl=255 time=0.261 ms
64 bytes from 192.168.30.7: icmp_seq=3 ttl=32 time=1.322 ms
64 bytes from 169.254.142.233: icmp_seq=4 ttl=255 time=0.225 ms

--- 169.254.255 ping statistics ---
5 packets transmitted, 5 packets received, +4 duplicates, 0.0% packet loss
round-trip min/avg/max/stddev = 0.137/0.743/1.488/0.570 ms
```

The broadcast ping return shows the IP address currently associated with **en4** and the IP address associated with the radio.

- 6 Based on the discovered radio IP address **192.168.30.7**, assign an IP address to the network interface on the host computer from the same subnet, such as **192.168.30.1**. Use a subnet mask of **255.255.255.0**.

```
$ sudo networksetup -setmanual "AX88179 USB 3.0 to Gigabit Ethernet" 192.168.30.1 255.255.255.0
```

Verify that the host computer can ping the IP address assigned to the radio.

```
$ ping -c 5 192.168.30.7
```

```
PING 192.168.30.7 (192.168.30.7): 56 data bytes
64 bytes from 192.168.30.7: icmp_seq=0 ttl=32 time=1.230 ms
64 bytes from 192.168.30.7: icmp_seq=1 ttl=32 time=1.388 ms
64 bytes from 192.168.30.7: icmp_seq=2 ttl=32 time=1.368 ms
64 bytes from 192.168.30.7: icmp_seq=3 ttl=32 time=1.450 ms
64 bytes from 192.168.30.7: icmp_seq=4 ttl=32 time=1.499 ms

--- 192.168.30.7 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.230/1.387/1.499/0.091 ms
```

## See Also

### More About

- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

# Verify Hardware Connection

## In this section...

- “Check Ethernet Configuration” on page 1-44
- “Check Host-to-Radio Connection” on page 1-44
- “Check Subnet Values on Host and Radio” on page 1-46

These sections show you how to confirm each part of the host-to-radio connection is configured and they are communicating successfully with each other.

## Check Ethernet Configuration

To make sure that all Ethernet cable and port connections and settings are optimal for host-to-radio communication, follow these steps.

- 1 Check that the radio is turned on.
- 2 Check that the Ethernet cable is plugged into the host computer and the Ethernet LED is on.
- 3 Check that the Ethernet cable is plugged into the radio and the Ethernet LED is on.
- 4 If you have multiple Ethernet ports on your host computer, check that the radio is connected to the desired Ethernet port.
- 5 Check that the TCP/IPv4 properties of the host computer Ethernet connection match those shown in “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13.
- 6 Check that the host computer and the radio have the same subnet value. Specifically, the first three octets of the IP address of the host computer must match the first three octets of the IP address of the radio. The last octet must be unique for the radio and the host computer. For example, the IP address 192.168.10.X, where X is a unique integer in the range [1, 254] for the radio and host computer. See “Check Subnet Values on Host and Radio” on page 1-46.

After you complete these steps, the Ethernet configuration is most likely configured correctly, and you can continue to verify the other connection checks.

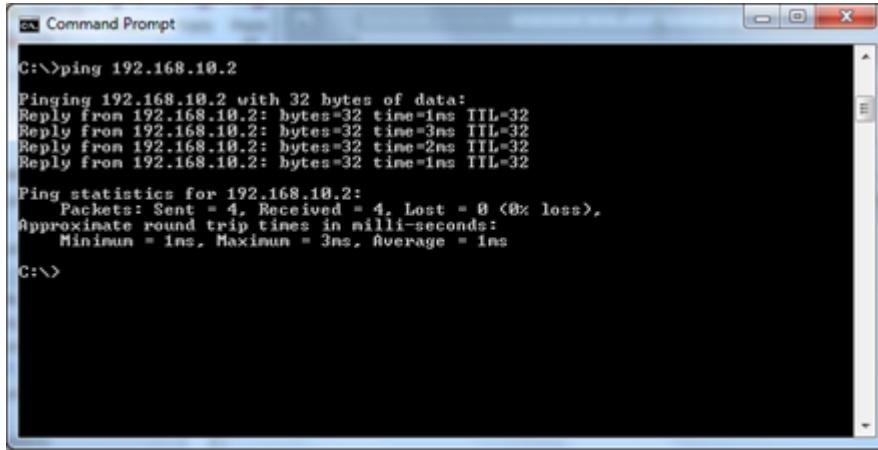
## Check Host-to-Radio Connection

- “Check Windows — Radio Connection” on page 1-44
- “Check Linux — Radio Connection” on page 1-45
- “Check MAC — Radio Connection” on page 1-45

### Check Windows — Radio Connection

Check if the host computer can communicate with the USRP radio through the Ethernet port by using the `ping` command.

- 1 Open a Command Prompt window.
- 2 Try to contact the USRP radio using the `ping` command with the IP address of the radio. For example, if the IP address of the radio is 192.168.10.2, use this command.  
`C:\>ping 192.168.10.2`
- 3 In this example the output shows the connection is successful. After you confirm a successful connection, go to “Check Subnet Values on Host and Radio” on page 1-46.



```
C:\>ping 192.168.10.2

Pinging 192.168.10.2 with 32 bytes of data:
Reply from 192.168.10.2: bytes=32 time=1ms TTL=32
Reply from 192.168.10.2: bytes=32 time=3ms TTL=32
Reply from 192.168.10.2: bytes=32 time=2ms TTL=32
Reply from 192.168.10.2: bytes=32 time=1ms TTL=32

Ping statistics for 192.168.10.2:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 3ms, Average = 1ms

C:\>
```

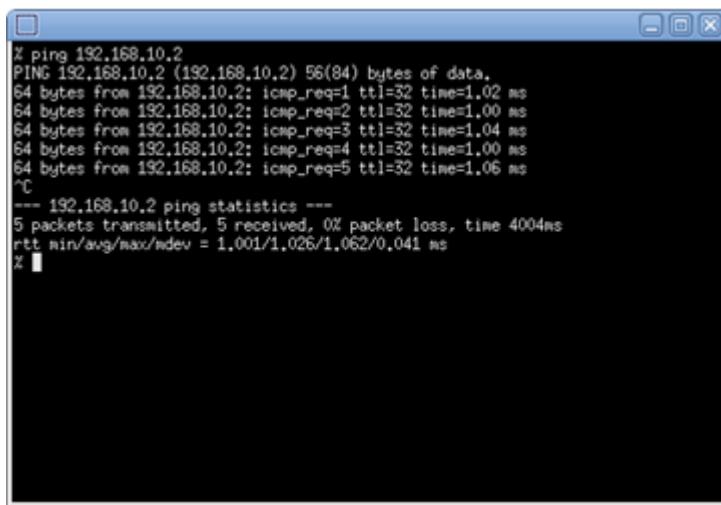
If you get the message "Request timed out", see "Check Ethernet Configuration" on page 1-44.

### **Check Linux – Radio Connection**

Check if the host computer can communicate with the USRP radio through the Ethernet port by using the **ping** command.

- 1** Open a Linux terminal window.
- 2** Try to contact the USRP radio using the **ping** command with the IP address of the radio. For example, if the IP address of the radio is 192.168.10.2, use this command.  

```
%ping 192.168.10.2
```
- 3** In this example the output shows the connection is successful. After you confirm a successful connection, go to "Check Subnet Values on Host and Radio" on page 1-46.



```
% ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_req=1 ttl=32 time=1.02 ms
64 bytes from 192.168.10.2: icmp_req=2 ttl=32 time=1.00 ms
64 bytes from 192.168.10.2: icmp_req=3 ttl=32 time=1.04 ms
64 bytes from 192.168.10.2: icmp_req=4 ttl=32 time=1.00 ms
64 bytes from 192.168.10.2: icmp_req=5 ttl=32 time=1.06 ms
^C
--- 192.168.10.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.001/1.026/1.062/0.041 ms
%
```

If you get the message "Request timed out", see "Check Ethernet Configuration" on page 1-44.

### **Check MAC – Radio Connection**

Check if the host computer can communicate with the USRP radio through the Ethernet port by using the **ping** command.

- 1** Open a Macintosh terminal window.
- 2** Try to contact the USRP radio using the **ping** command with the IP address of the radio. For example, if the IP address of the radio is 192.168.10.2, use this command.

```
% ping 192.168.10.2
```
- 3** In this example the output shows the connection is successful. After you confirm a successful connection, go to “Verify MATLAB Connection to USRP Radio” on page 1-22.

```
% ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_req=1 ttl=32 time=1.02 ms
64 bytes from 192.168.10.2: icmp_req=2 ttl=32 time=1.00 ms
64 bytes from 192.168.10.2: icmp_req=3 ttl=32 time=1.04 ms
64 bytes from 192.168.10.2: icmp_req=4 ttl=32 time=1.00 ms
64 bytes from 192.168.10.2: icmp_req=5 ttl=32 time=1.06 ms
^C
--- 192.168.10.2 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4004ms
rtt min/avg/max/mdev = 1.001/1.026/1.062/0.041 ms
%
```

If you get the message "Request timed out", see “Check Ethernet Configuration” on page 1-44.

## Check Subnet Values on Host and Radio

The host computer and the USRP radio must have the same subnet values for successful communication. To confirm that the host computer and the radio have the same subnet values, use these instructions.

Both Windows and Linux instructions use the **ping** command to check for a response from the hardware at the IP address you specify. If you get a response from the device, you can consider this step successful and move on to “Verify MATLAB Connection to USRP Radio” on page 1-22.

---

**Note** You can skip this step if you successfully pinged the radio from the host computer using the procedure in “Check Host-to-Radio Connection” on page 1-44. A successful ping confirms that the subnet values on the host computer and the radio are the same.

---

- “Windows” on page 1-46
- “Linux” on page 1-47

### Windows

The broadcast address on the LAN to which you connect your radio is the IP address with 255 as the last octet. For example, if your IP address is 192.168.10.1, the broadcast address for the network is 192.168.10.255. Refer to “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13 to find the address of your host computer.

Use the ping command to discover all devices connected to this Ethernet port.

```
ping 192.168.10.255
```

In some systems, you might need to add a -b option, as in `ping -b 192.168.10.255`. In this example, this is the system output.

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The title bar includes standard window controls (minimize, maximize, close) and the title. The main area displays the following text:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

H:\>ping 192.168.10.255

Pinging 192.168.10.255 with 32 bytes of data:
Reply from 192.168.10.2: bytes=32 time=1ms TTL=32

Ping statistics for 192.168.10.255:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 1ms, Average = 1ms

H:\>
```

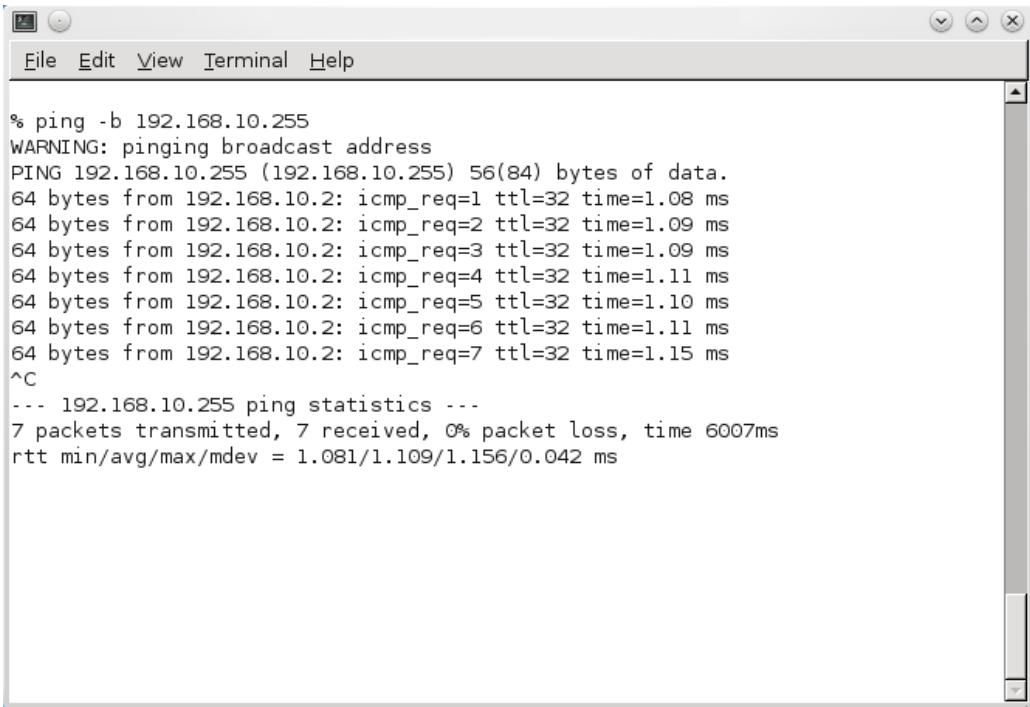
## Linux

- 1 Run the `ifconfig` command in a Linux terminal to find the IP configuration of the Ethernet port to which you connect your radio.

Alternatively, run `ifconfig ethx`, where `eth` is the Ethernet port, and `x` is a nonnegative number.

- 2 The broadcast address for this port is 192.168.10.255. Use the `ping` command to discover all devices connected to this port.

```
ping -b 192.168.10.255
```



A screenshot of a terminal window titled "Terminal". The window shows the command "% ping -b 192.168.10.255" being run. The output includes a warning about pinging a broadcast address, followed by seven ICMP echo requests (icmp\_req=1 to 7) with their respective TTL values (32) and times (ranging from 1.08 ms to 1.15 ms). The command '^C' is shown as an interrupt. Finally, the ping statistics are displayed, indicating 7 packets transmitted, 7 received, 0% packet loss, a total time of 6007ms, and an rtt min/avg/max/mdev of 1.081/1.109/1.156/0.042 ms.

```
% ping -b 192.168.10.255
WARNING: pinging broadcast address
PING 192.168.10.255 (192.168.10.255) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_req=1 ttl=32 time=1.08 ms
64 bytes from 192.168.10.2: icmp_req=2 ttl=32 time=1.09 ms
64 bytes from 192.168.10.2: icmp_req=3 ttl=32 time=1.09 ms
64 bytes from 192.168.10.2: icmp_req=4 ttl=32 time=1.11 ms
64 bytes from 192.168.10.2: icmp_req=5 ttl=32 time=1.10 ms
64 bytes from 192.168.10.2: icmp_req=6 ttl=32 time=1.11 ms
64 bytes from 192.168.10.2: icmp_req=7 ttl=32 time=1.15 ms
^C
--- 192.168.10.255 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6007ms
rtt min/avg/max/mdev = 1.081/1.109/1.156/0.042 ms
```

## See Also

### More About

- “Resolving Ethernet Subnet Conflict” on page 1-52
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

## Using One Ethernet Port

If the host computer has only one Ethernet connection available and you disconnect the Internet connection to use the network interface card (NIC) for the host-to-radio connection, you must reconnect to the Internet using that NIC before logging out the host computer.

---

**Caution** If you do not reconnect the host computer to the Internet using that NIC before logging out, you might lose the Internet configuration settings on the host computer and your ability to log on. If you lose your ability to log on to the host computer, contact your system administrator to reconfigure its Internet configuration settings.

---

### See Also

#### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

## Make Changes Persistent on Linux

On Linux systems, unless you make network connection changes persistent, a system reboot resets the network connection changes and loses the host-to-radio connection. To retain the host-to-radio connection after a computer reboot, make the network connection changes to the host computer persistent by making these changes in the `/etc/network/interfaces`/`etc/sysctl.conf`, and `/etc/security/limits.conf` files.

- Edit `/etc/network/interfaces` to define settings for `eth1`. Use an IP address on the same subnet as your radio (that is, the same first three octets match those of your radio) and a unique value for the fourth octet. For more information, see “Check Subnet Values on Host and Radio” on page 1-46.

```
auto eth1
iface eth1 inet static
    address 192.168.30.1
    netmask 255.255.255.0
```

- Edit `/etc/sysctl.conf` to add lines defining these settings for `net.core.rmem_max` and `net.core.wmem_max`.

```
net.core.rmem_max=50000000
net.core.wmem_max=33554432
```

For more information, see [https://files.ettus.com/manual/page\\_transport.html#transport\\_udp\\_linux](https://files.ettus.com/manual/page_transport.html#transport_udp_linux).

- To define the maximum real-time scheduling priority `rtprio`, edit `/etc/security/limits.conf` and add this line. Use your group name in place of GROUP.

```
@GROUP - rtprio 99
```

To determine your group name, run this command in your Linux terminal.

```
$ id -gn
```

For more information, see [https://files.ettus.com/manual/page\\_general.html#general\\_threading](https://files.ettus.com/manual/page_general.html#general_threading).

## See Also

### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5
- “Manual USRP Radio Support Package Hardware Setup” on page 1-24

## Linux Systems with No prlimit Command

If your Linux machine does not have the *prlimit* command use this command to define the maximum real-time scheduling priority *rtprio*. Use your group name in place of GROUP

```
@GROUP - rtprio 99
```

To determine your group name run:

```
$ id -gn
```

For more information, see [https://files.ettus.com/manual/page\\_general.html#general\\_threading](https://files.ettus.com/manual/page_general.html#general_threading).

These changes are not persistent. To retain these settings in your Linux account, see “Make Changes Persistent on Linux” on page 1-50.

## Resolving Ethernet Subnet Conflict

Follow these instructions if the host-to-radio connection cannot be established because the IP address of the radio is on a subnet being used by the host computer for another Ethernet port or for the wireless network interface. For this scenario, *NIC1* refers to the Ethernet port or the wireless network interface connected to the internet or external network. *NIC2* refers to the network interface card (NIC) you intend to use to establish the host-to-radio connection. To diagnose and resolve the conflict follow these steps.

- 1** “Check Subnet Values on Host and Radio” on page 1-46
- 2** Work with your system administrator to disable *NIC1*.
- 3** After disabling *NIC1*, confirm that you are able to establish a connection to the radio using *NIC2* on the conflicting IP address subnet.
- 4** Run the installer to establish a host-to-radio connection on the conflicting IP address subnet.
- 5** After confirming that the radio and *NIC2* can communicate, exit the installer, and change the radio IP address to a different subnet. Use the `setsdruiip` function to set the radio IP address to a different subnet than the one used by *NIC1*. Changing only the third octet of the IP address is sufficient.
- 6** Start the installer again to establish a connection between the host computer using *NIC2* and the radio on its new IP address subnet.
- 7** Enable *NIC1* to reestablish your access to the external network (or the Internet).

## See Also

### More About

- “Check Subnet Values on Host and Radio” on page 1-46

# USRP Radio Firmware Update

## In this section...

- “Why Download New Firmware?” on page 1-53
- “Updating USRP Radio Firmware” on page 1-53

## Why Download New Firmware?

The Communications Toolbox Support Package for USRP Radio uses a specific version of the UHD software on the host computer side. If the USRP radio has a different version of UHD firmware installed, you might not be able to communicate with the USRP radio and use the support package.

**Note** For the Bus Series devices, you do not need to check or update the firmware. It is automatically loaded when the device is plugged in.

You can update the UHD firmware using the `sdruload` function.

## Updating USRP Radio Firmware

Before updating the radio firmware, make sure that the radio is connected to and communicating with the host computer. If not, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5.

**Note** Custom firmware image is not supported for X300 or X310.

### Update Networked Series Radio Firmware

Calling the `sdruload` function in MATLAB with the following parameters loads the default images to a radio at default IP address 192.168.10.2:

```
sdruload('Device','dev')
```

Depending on which Networked Series device you have connected, replace '`dev`' with `n200` or `n210`.

You can specify additional parameters, such as a non default IP address and non default firmware and FPGA images. See `sdruload`.

### Update USRP2 Radio Firmware

Calling the `sdruload` function in MATLAB with the following parameters writes the default FPGA and UHD firmware image to an SD card:

```
sdruload('Device','USRP2')
```

**Note** Before loading firmware for a USRP2 Radio, you must manually install `usrp2_card_burner.py` (for all platforms) and Python and dd (only for Windows platforms) follow these directions.

## 1 Download usrp2\_card\_burner.py

```
% See the GNU General Public License at https://www.gnu.org/licenses
cd(fullfile(sdruroot,'uhdapps','utils'))
websave('usrp2_card_burner.py','https://raw.githubusercontent.com/EttusResearch/uhd/4286d07d7a31812ba16d67d3c7e0c05a27e444cb/host/')

if isunix
    system('chmod +x usrp2_card_burner.py');
end
```

## 2 Install Python for Windows

```
% See the license for Python at https://docs.python.org/2/license.html
if ispc
    cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot));
    if ~exist('3P.instrset','dir')
        mkdir 3P.instrset
    end
    cd 3P.instrset
    mkdir pythonwin.instrset
    websave('python-2.7.3.amd64.msi','https://www.python.org/ftp/python/2.7.3/python-2.7.3.amd64.msi');
    system(['msiexec /a python-2.7.3.amd64.msi /qn TARGETDIR=' pwd '\pythonwin.instrset\python']);
end
```

## 3 Download dd for Windows

```
% See the license for dd at http://www.chrysocome.net/gpl
if ispc
    cd(fullfile(matlabshared.supportpkg.getSupportPackageRoot));
    if ~exist('3P.instrset','dir')
        mkdir 3P.instrset
    end
    cd 3P.instrset
    websave('dd-0.5.zip','http://www.chrysocome.net/downloads/f5897020e67e51472d1962606fc8ad69/dd-0.5.zip');
    unzip('dd-0.5.zip',fullfile(sdruroot,'uhdapps','utils'));
end
```

---

## See Also

## More About

- “What to Do After Installation” on page 1-55

## What to Do After Installation

After you have successfully installed the support package, you can do any of the following:

- Experiment with the Support Package for USRP Radio Featured Examples.
- Check out “Radio Management” topics. These topics explain how to create blocks and System objects, and how to adjust radio settings.
- Tweak the performance of your design with “Performance” optimizations.

In addition, the section “Common Problems and Fixes” on page 2-2 can help you troubleshoot performance issues and any errors you could encounter.

---

**Note** Each time you want to use the Communications Toolbox Support Package for USRP Radio with MATLAB, call the function `setupsdr` in MATLAB. This function performs certain behind-the-scenes operations so that you can use this support package without errors each time.

## Uninstall Support Packages

To uninstall a hardware support package, follow these steps:

- 1 On the MATLAB **Home** tab, in the **Environment** section, click **Add-Ons > Manage Add-Ons**.
- 2 Select the support package and then click **Uninstall**.

### See Also

#### Related Examples

- “Get and Manage Add-Ons”
- “Supported Hardware – Software-Defined Radio”

# Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection

## In this section...

["Host Computer Ethernet Options" on page 1-57](#)

["Configure Ethernet Connection Using Installer" on page 1-57](#)

These sections guide you through the instructions to configure a host computer for an Ethernet-based USRP N3xx radio (N300, N310, N320, N321) connection using the installer. To run the installer, see "Install Communications Toolbox Support Package for USRP Radio" on page 1-4.

**Note** If you are using a USB-based radio connection, see "Configure Host Computer for USB-Based Radio Connection" on page 1-7.

## Host Computer Ethernet Options

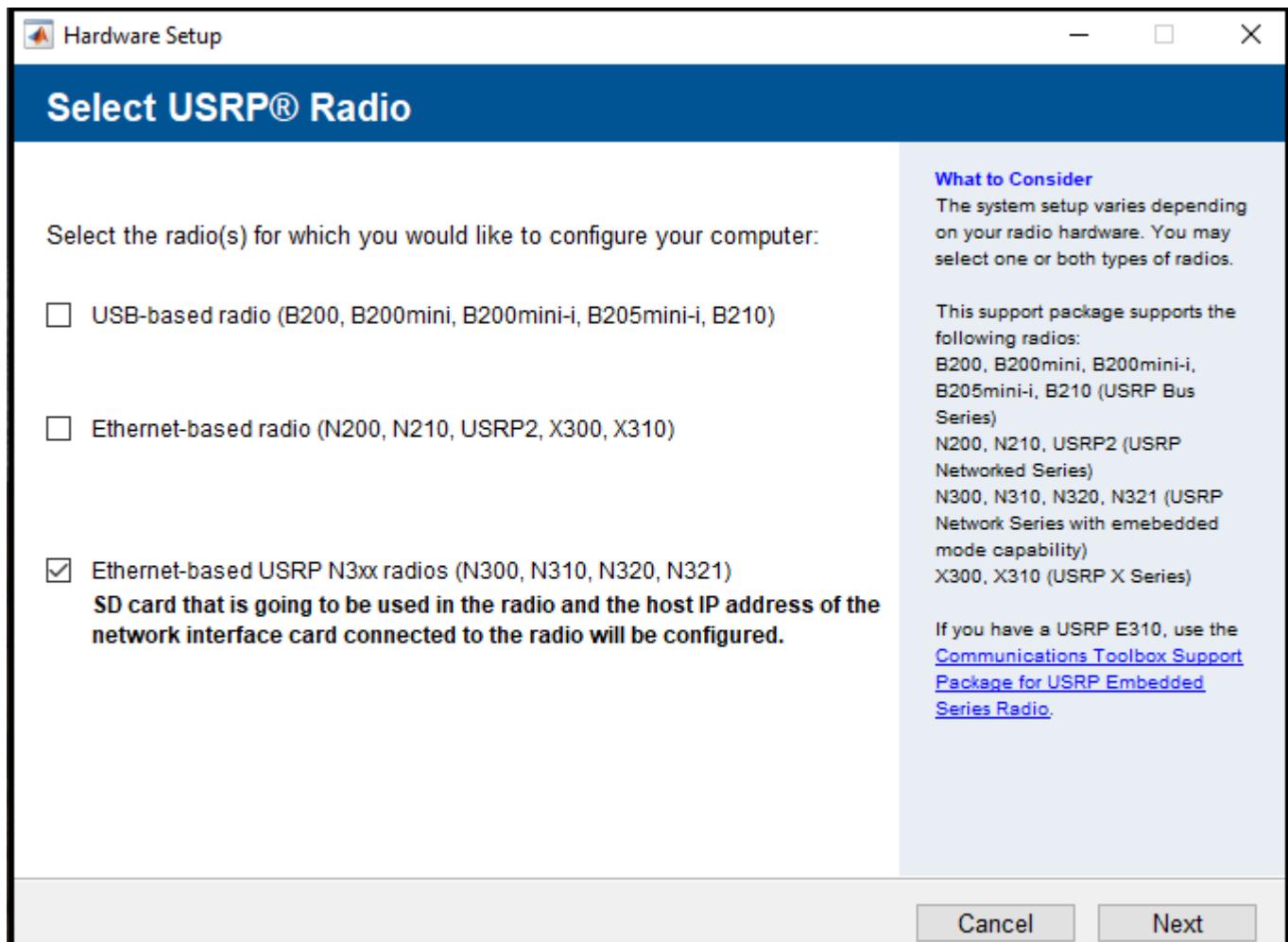
The Ethernet connection is often referred to as a network connection. You can use an integrated network interface card (NIC) with a Gigabit Ethernet cable. This connection is necessary for transmitting data, such as an FPGA or firmware image, from the host computer to the radio hardware. It is also necessary for sending and receiving signals to and from the radio hardware.

To have simultaneous internet access in the absence of a wireless connection, the host computer must have two Ethernet connections. If the host computer has only one Ethernet connection available, see "Using One Ethernet Port" on page 1-49.

## Configure Ethernet Connection Using Installer

Unless otherwise noted, the installer instructions for configuring the Ethernet connection between the host and radio is the same for Windows, Linux, and Macintosh computers.

- 1 To have the installer guide you through the setup of the host-to-radio Ethernet connection, select **Ethernet-based USRP N3xx radios (N300, N310, USRP2, X300, X310)**, and then click **Next**.



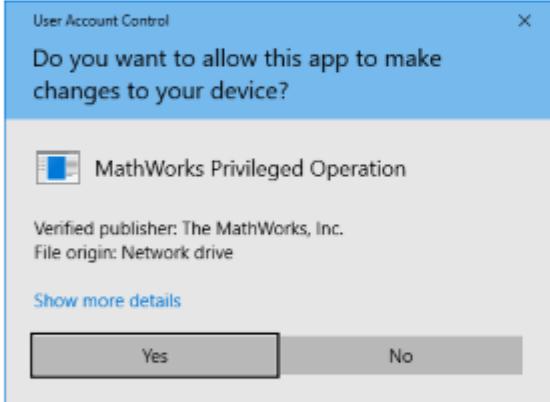
- 2 Check that the host machine has the correct configuration and required accessories.
  - a Administrator privileges — Because the software runs OS commands for configuring the network card, you must have administrator privileges for the guided radio hardware setup. For windows, If you see the User Account Control dialog box, click **Yes** to continue the setup process.
    - For Linux: If you see the User Login dialog box, enter the administrator password and click **login**.
    - For Mac: If you see osascript wants to make changes dialog box, enter the password and click **OK**.
  - b Gigabit Ethernet connection — This connection is often referred to as a network connection. You can use the small form-factor pluggable (SFP) network interface with a Gigabit Ethernet cable.
  - c SD card reader and writable SD card — If the host machine does not have an integrated card reader, use an external USB SD card reader.

**Hardware Setup**

## USRP® N3XX Network-mode setup checklist

Make sure you have:

- Elevated user permissions. If the following User Account Control window pops up, you must click 'Yes' to continue with the setup process.



The dialog box is titled "User Account Control" and asks "Do you want to allow this app to make changes to your device?". It shows the publisher as "MathWorks Privileged Operation" and the file origin as "Network drive". There are "Yes" and "No" buttons at the bottom.

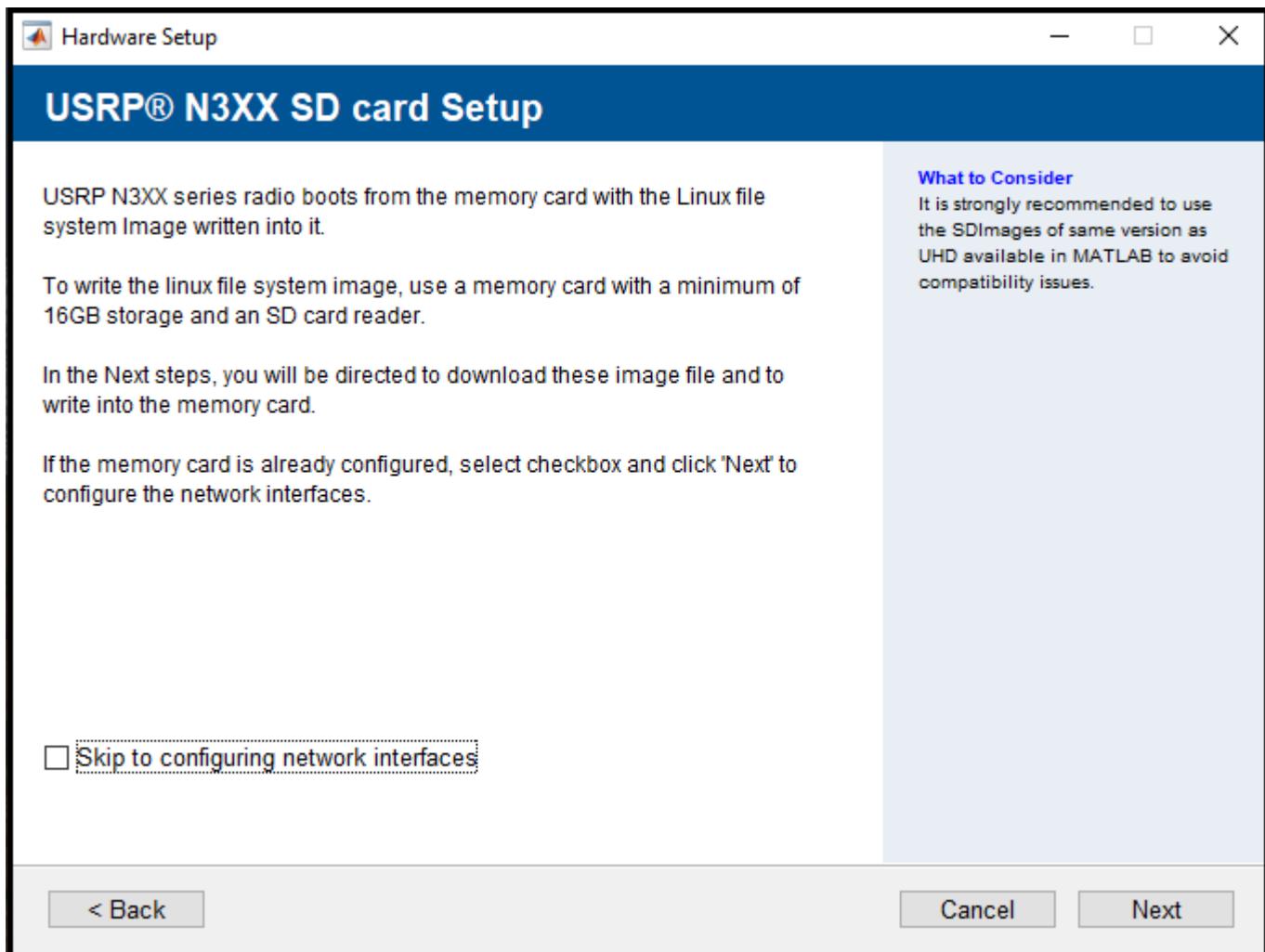
- 1 or 10 Gigabit SFP+ Ethernet Interface
- microSD card reader
- Writable microSD card (16GB or larger)

**What to Consider**  
The process to setup the host computer and USRP® N3XX radio hardware can take upto 15 minutes.

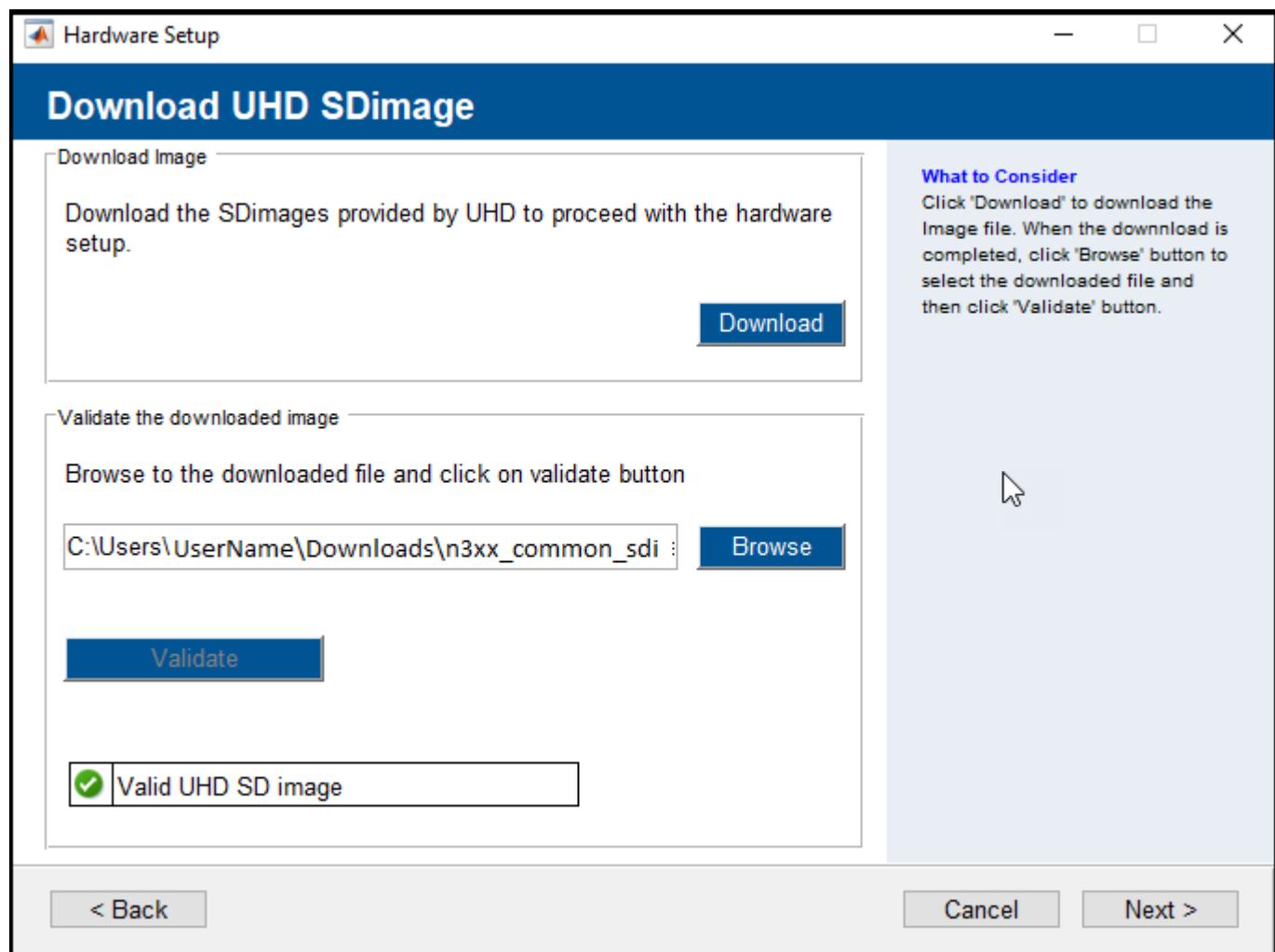
**Gigabit Ethernet connection**  
USRP® N3XX are connected to SFP+ ports to communicate with Host PC. Use SFP+ Adaptor with wired Etherenet cable or Ethernet cable with SFP terminations to the SFP+ ports of the radio.

< Back      Cancel      Next >

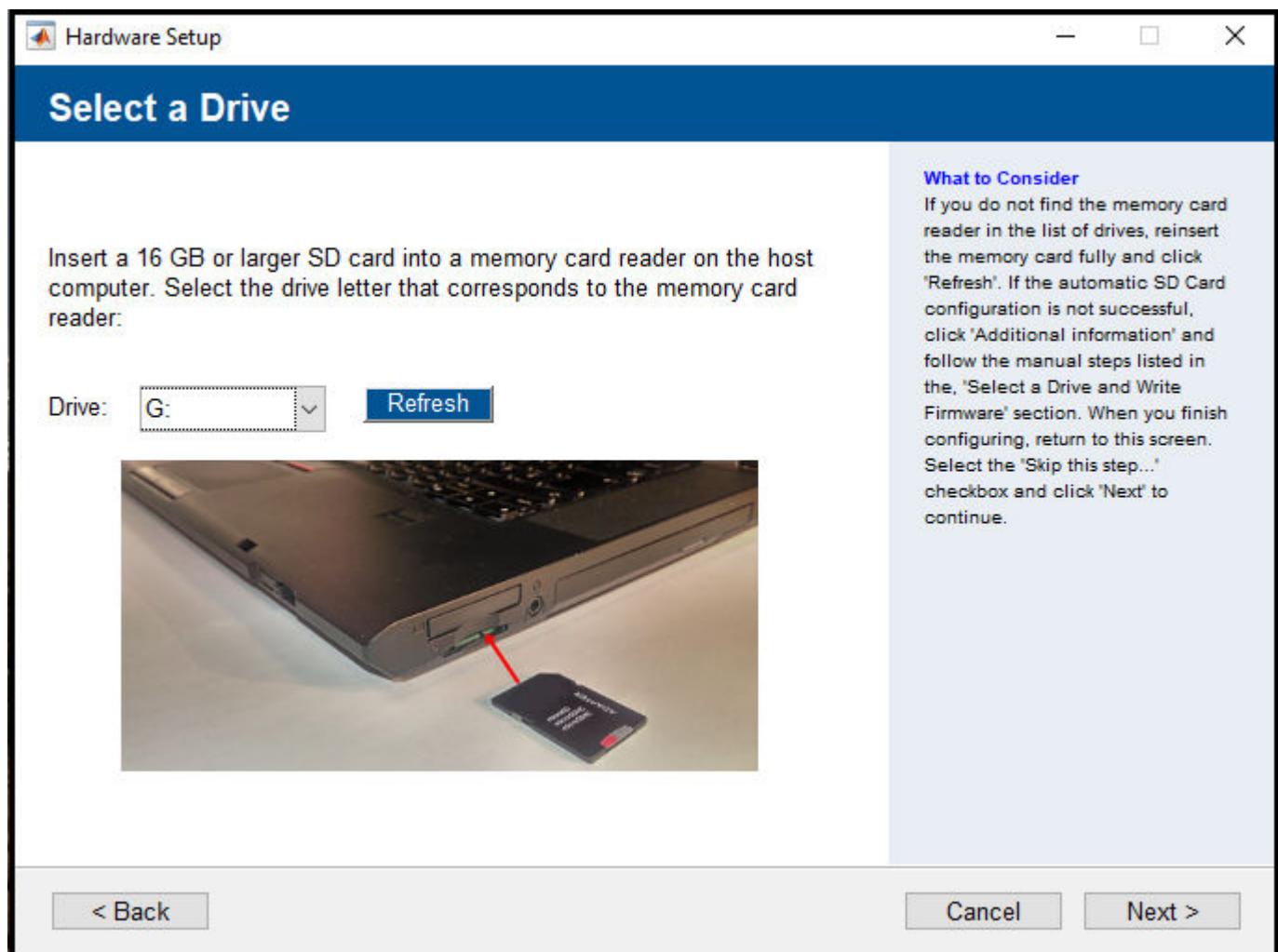
- 3** If you have already set up an SD card with the Linux system file image, select **Skip to configuring network interfaces**. Click **Next** and continue to step 4. To set up the SD memory card, click **Next**, and then follow these steps.



- a Download the UHD SDimage and validate it.

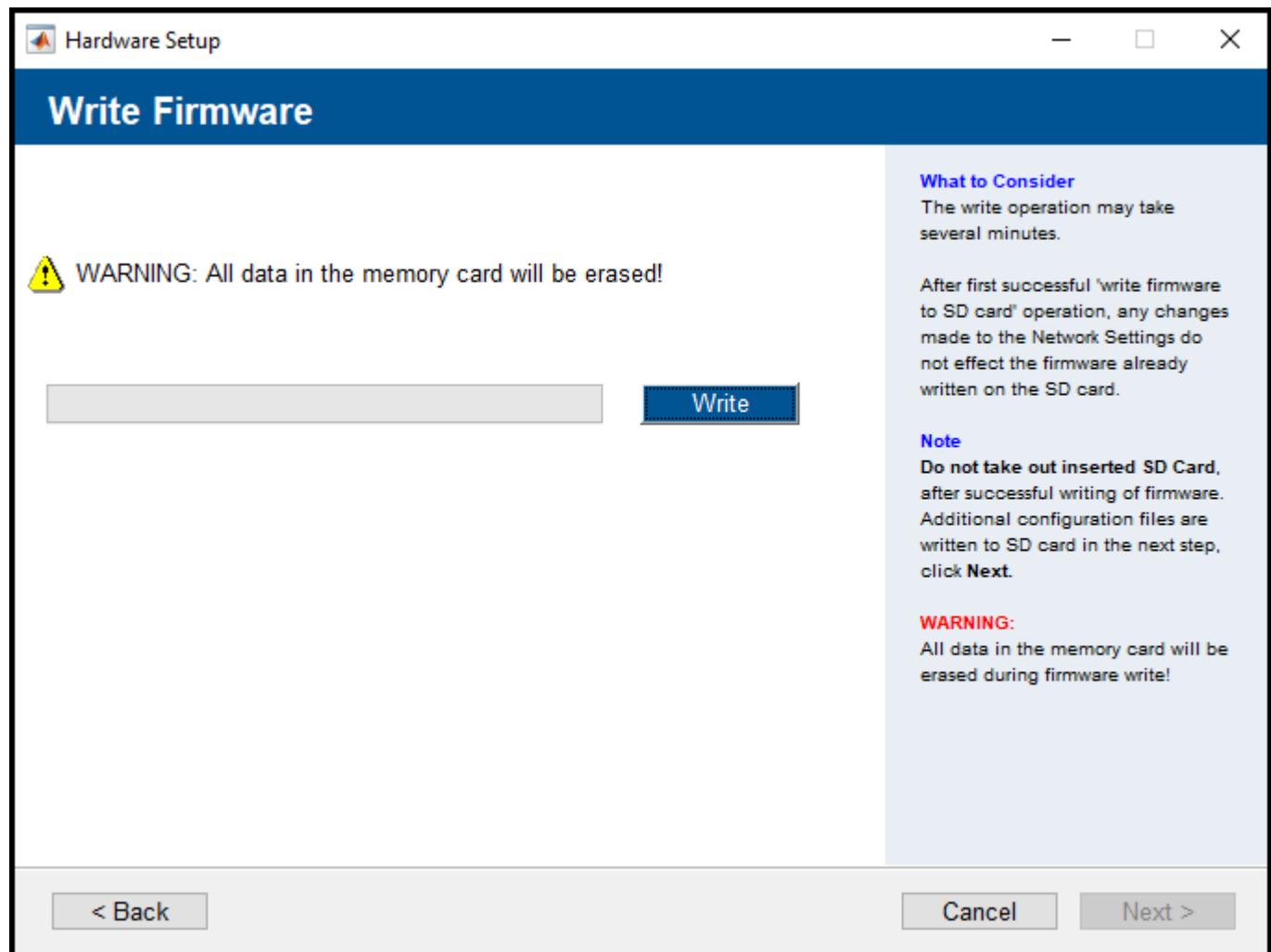


- b Insert a 16 GB or larger SD memory card into the selected drive on the host computer. Then, click **Next**.

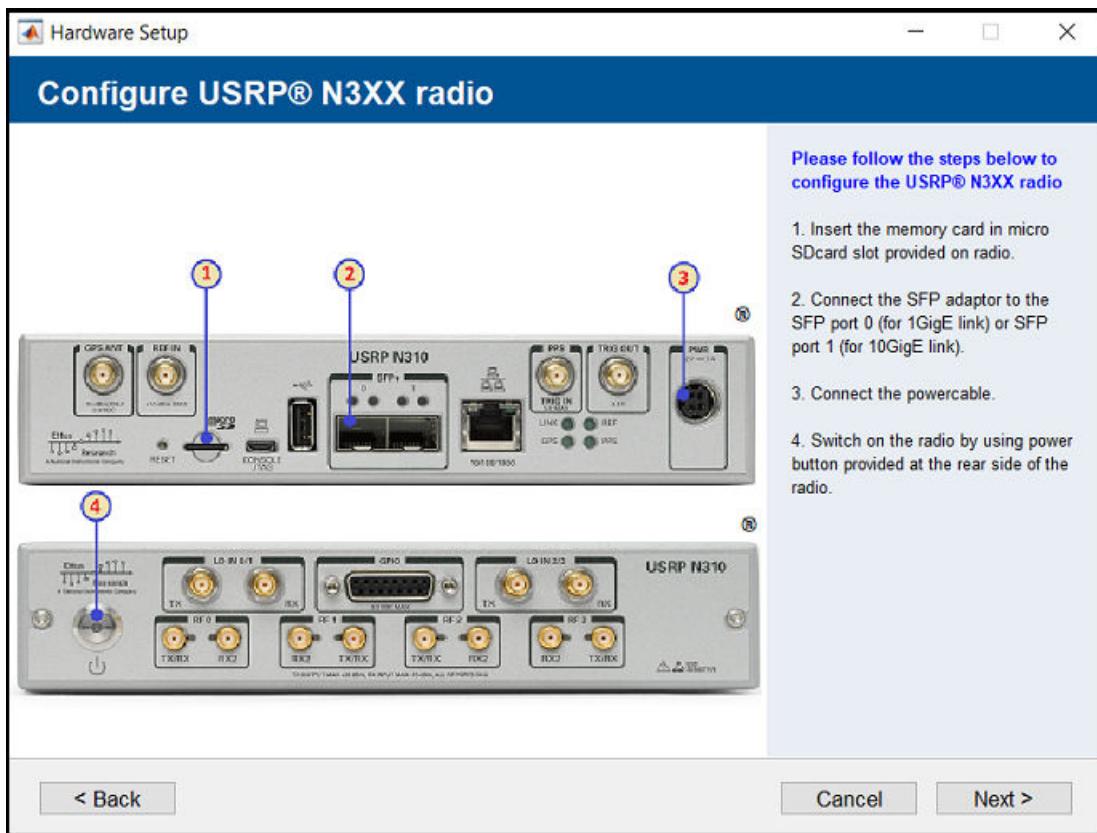


When you click **Next**, a warning message appears. The warning message indicates that all previous data on the SD memory card will be erased during the firmware write.

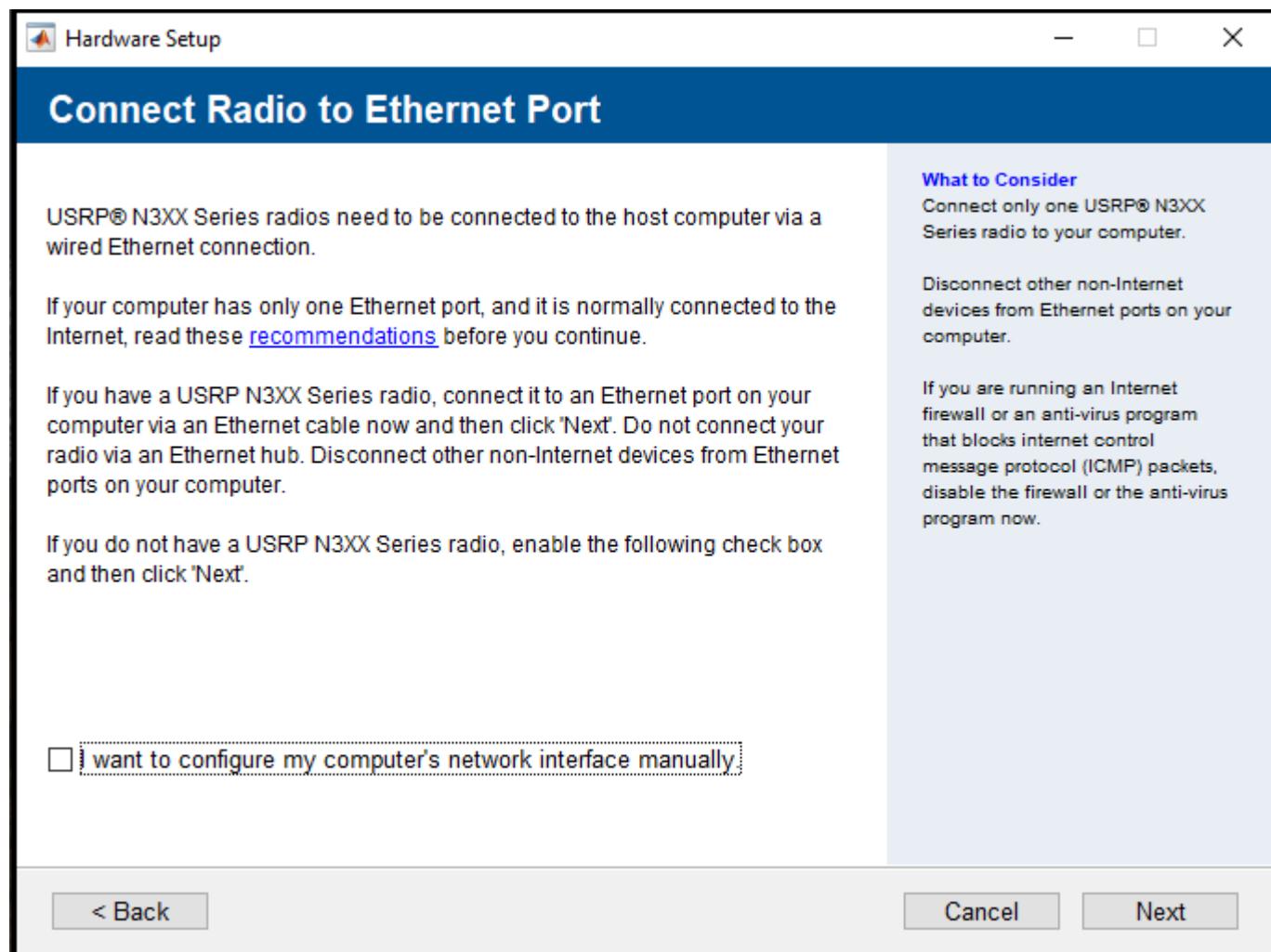
- c When you are ready to proceed with the SD card image download, click **Write**.



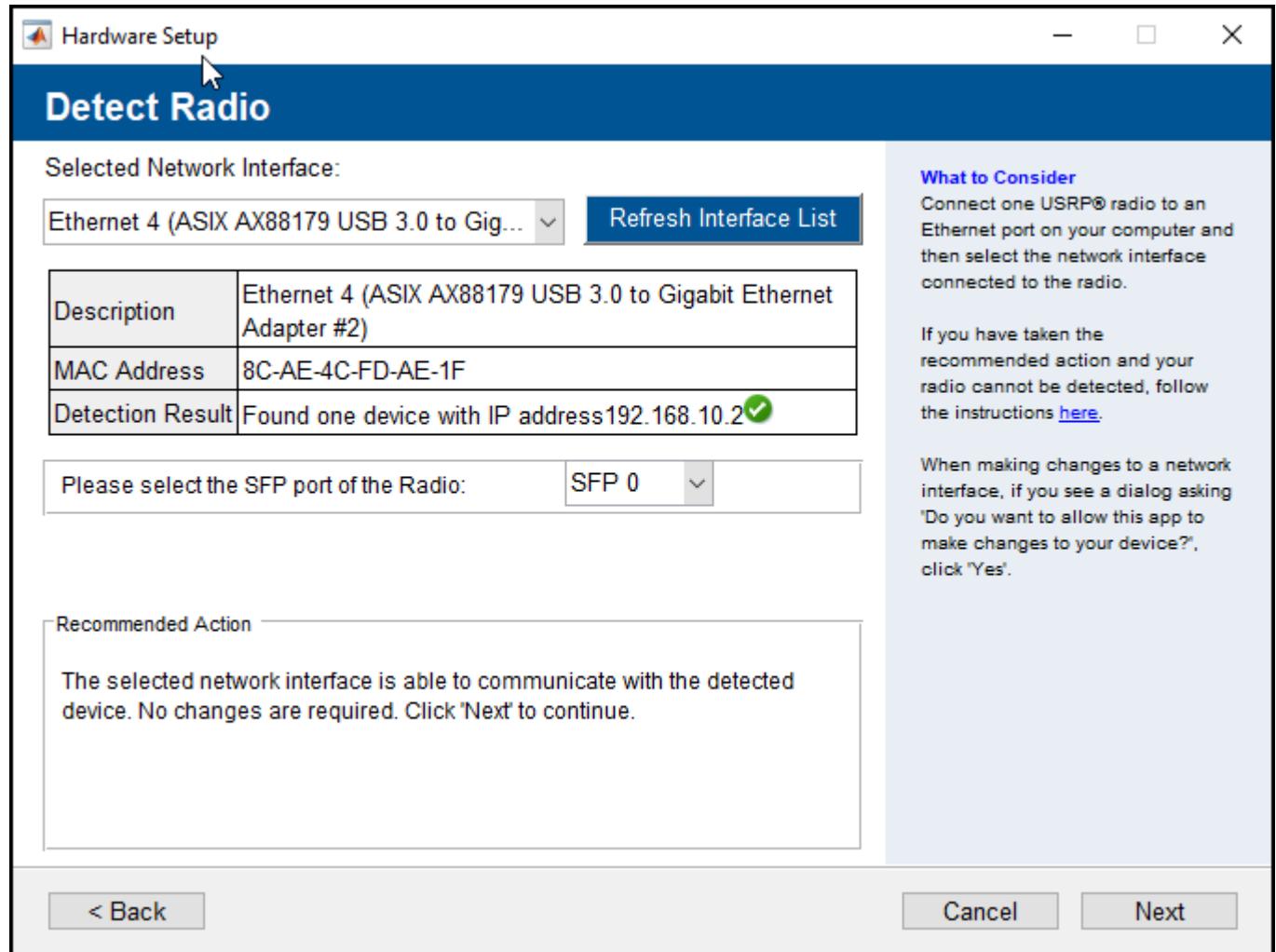
- 4 Configure the USRP N3xx radio as instructed in this figure.



- 5** If you do not have an N3xx radio connected to the host computer, see “Configure Network Interface Using Installer with No USRP N3xx Radio Connected” on page 1-73.
- To have the installer configure the host-to-radio Ethernet connection, click **Next**. If your computer has only one NIC, see “Using One Ethernet Port” on page 1-49.

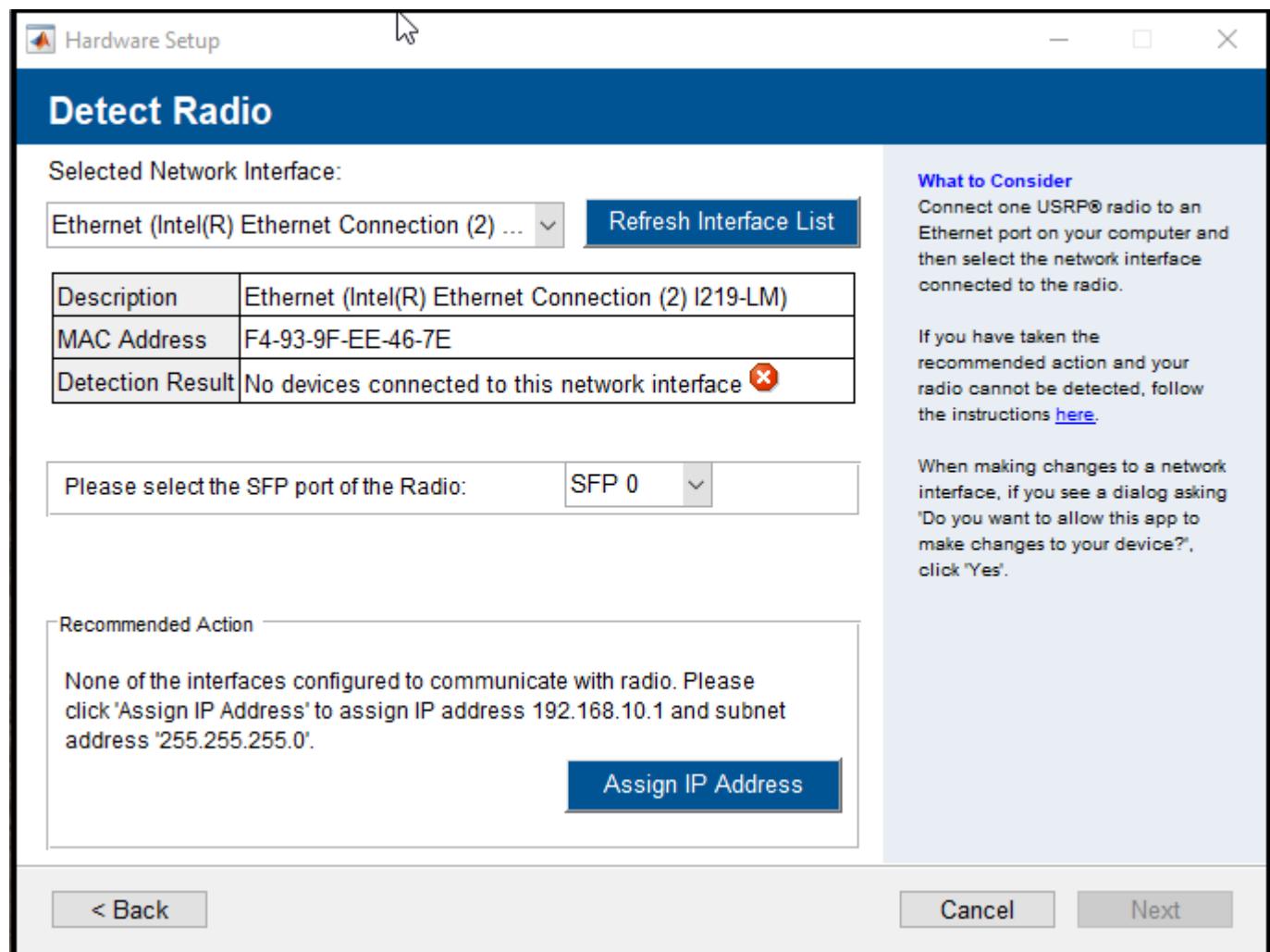


Confirm that the **Detection Result** field lists the IP address of the configured N3xx radio.

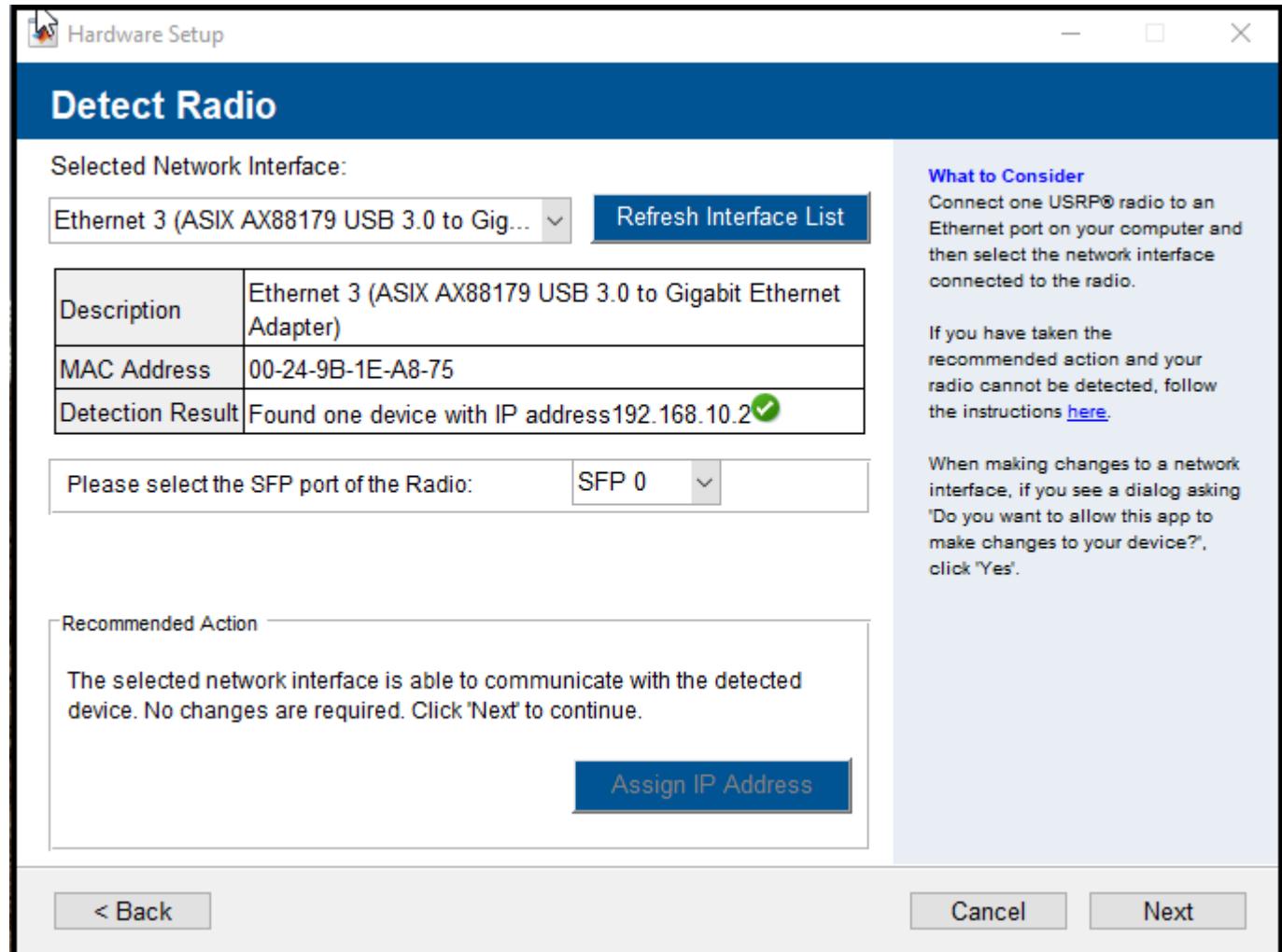


- To manually configure the network interface of your host computer, select **I want to configure my computer's network interface manually**. Then, click **Next**.

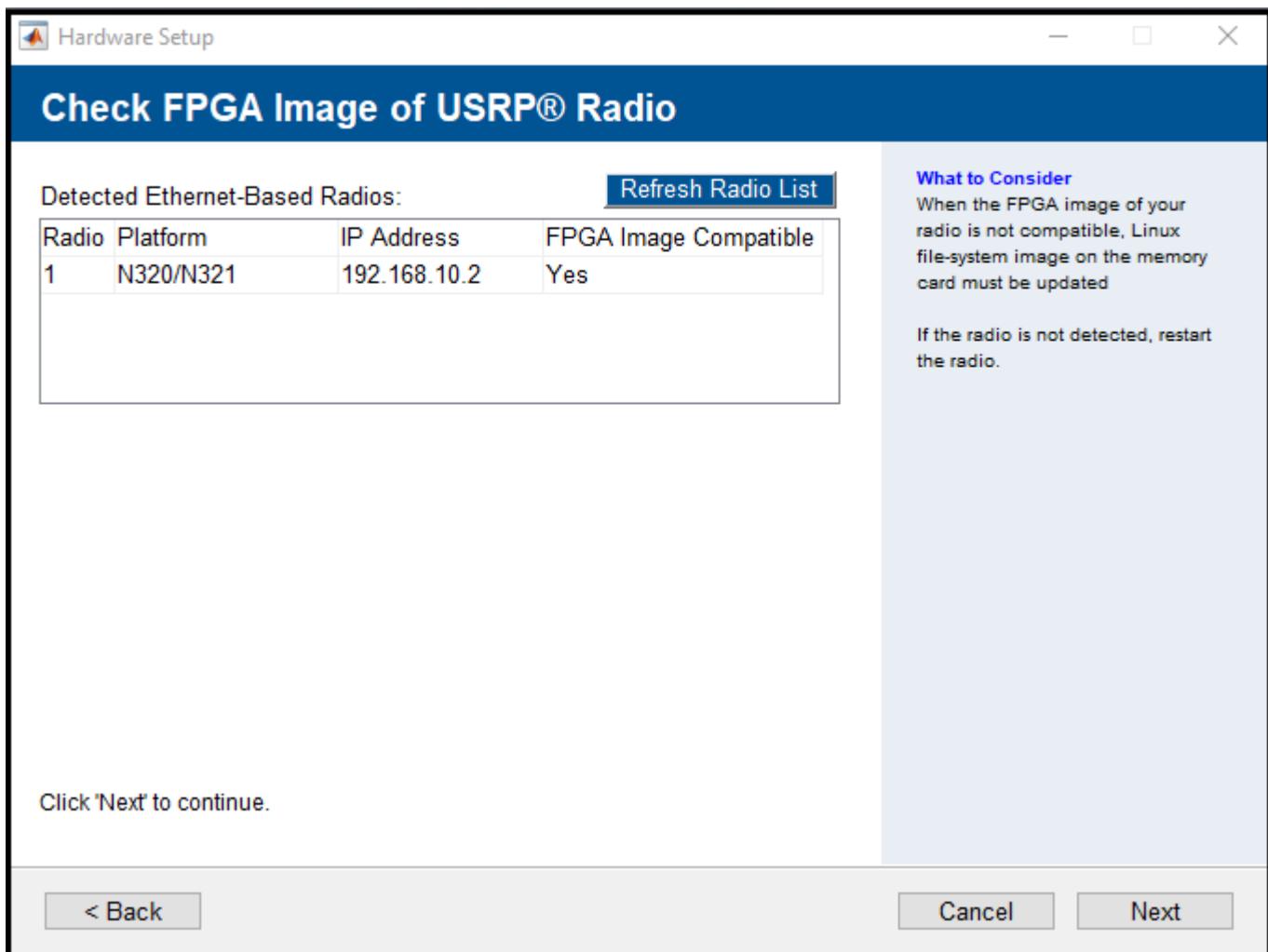
If your radio is not detected, check the **Recommended Action** section. If you are requested to apply any changes to the host IP address and the subnet mask of the selected network interface, click **Assign IP Address**.



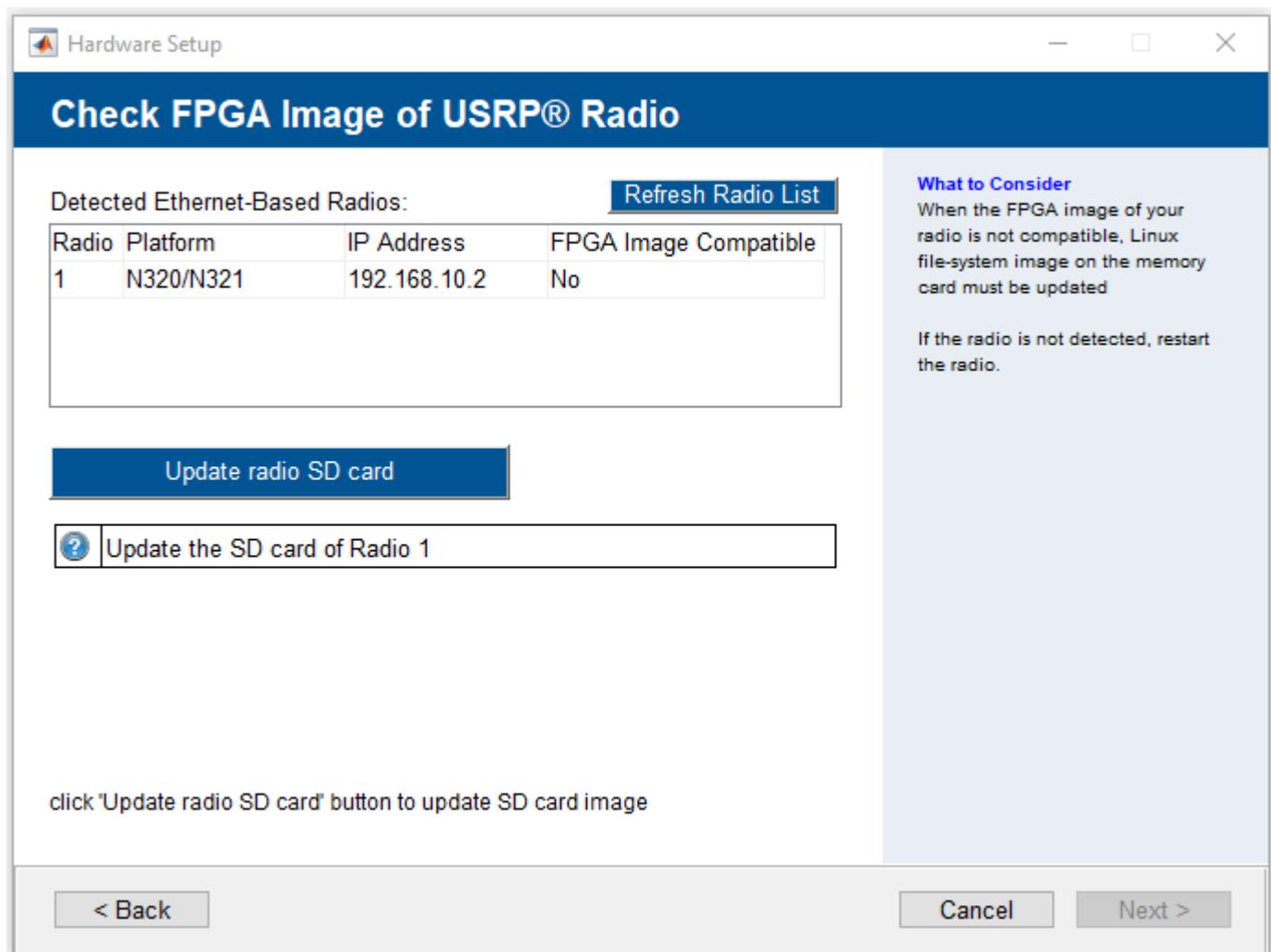
Confirm that the **Detection Result** field lists the IP address of the configured N3xx radio.



- 6** If the FPGA image of your N3xx radio is compatible with the software version being installed, Click **Next**.



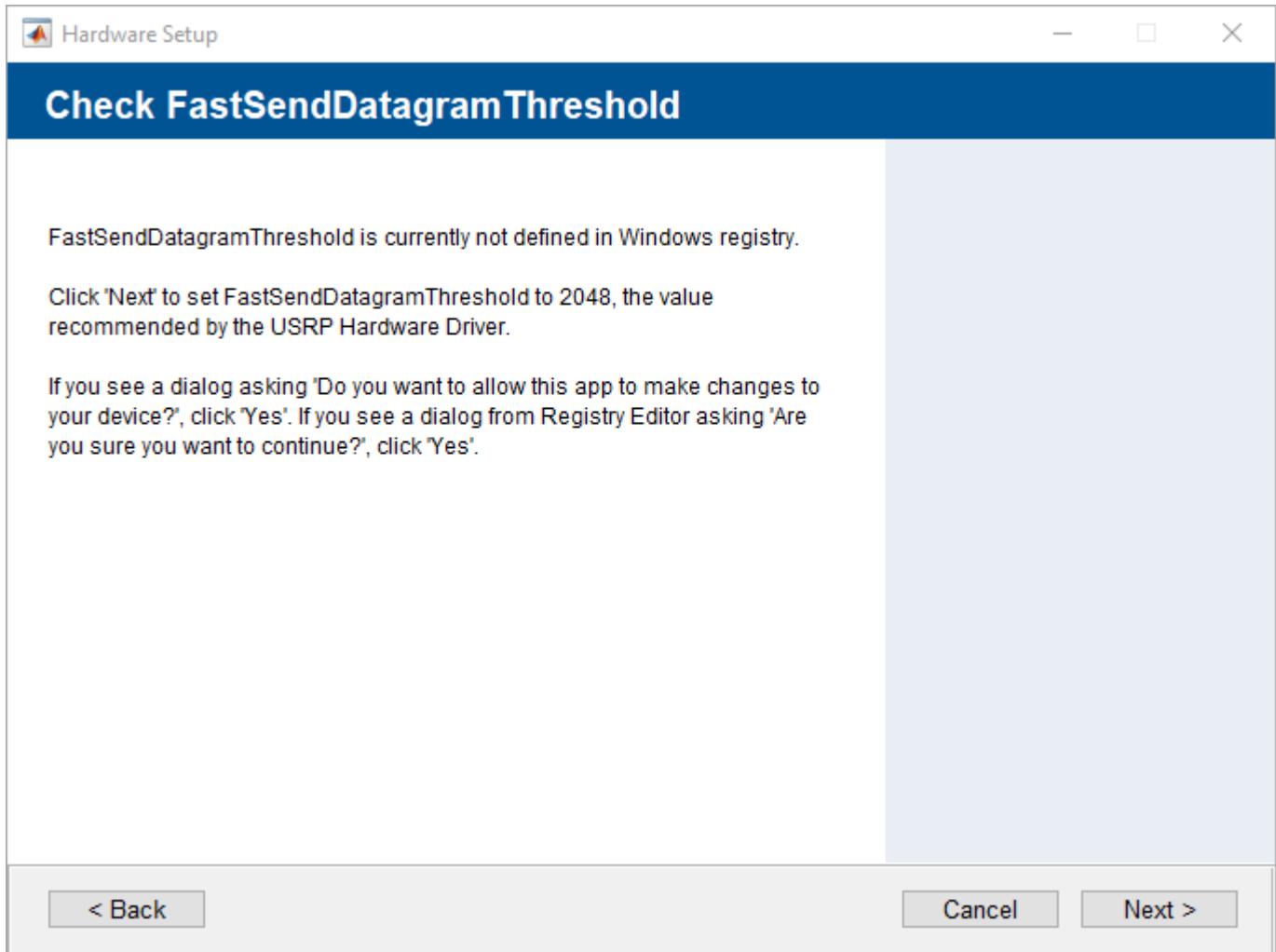
If the FPGA image of your N3xx radio is not compatible with the software version being installed, click **Update radio SD card** to update the SD card image.



#### 7 This step applies for Windows only.

If FastSendDataGramThreshold is not defined in the Windows registry, the installer prompts you to set it to the value recommended by the USRP hardware driver from Ettus Research. To set FastSendDataGramThreshold to the recommended value, click **Next**.

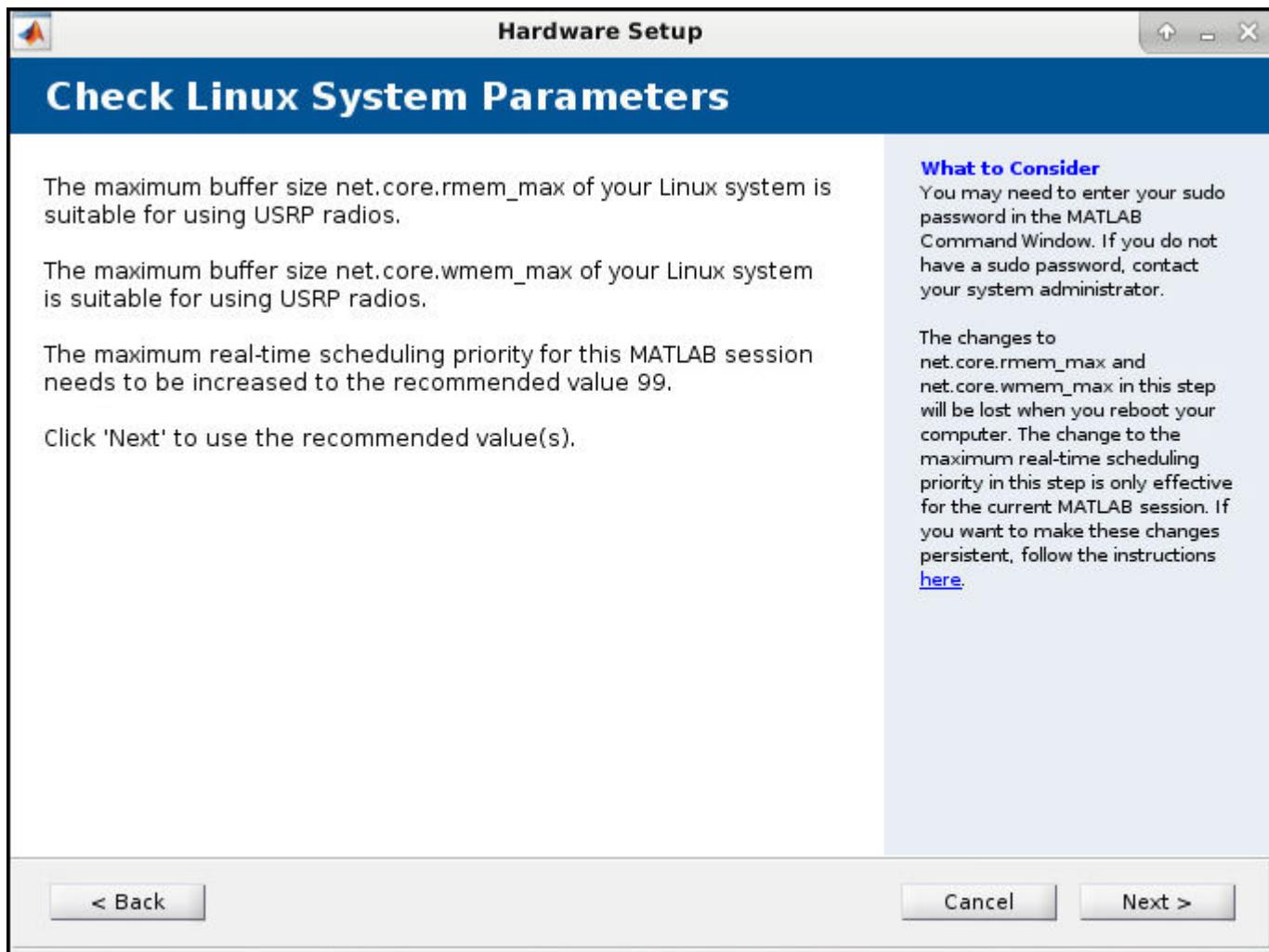
If the **Do you want to allow this app to make changes to your device?** dialog box appears, click **Yes** to continue. If the **Do you want to allow this app to make changes to your device?** dialog box appears from the Registry Editor, click **Yes** to continue.



**8 This step applies for Linux only.**

Ettus Research recommends setting the maximum buffer sizes for `net.core.rmem_max`, `net.core.wmem_max`, and real-time scheduling as indicated in this figure. To use these recommended values, click **Next**.

These changes are not persistent. To retain these settings in your account, see “Make Changes Persistent on Linux” on page 1-50.



- 9** After enabling your Ethernet-based device, confirm the host-to-radio communication link by testing the radio connection. For details, see “Test Radio Connection” on page 1-21.

## See Also

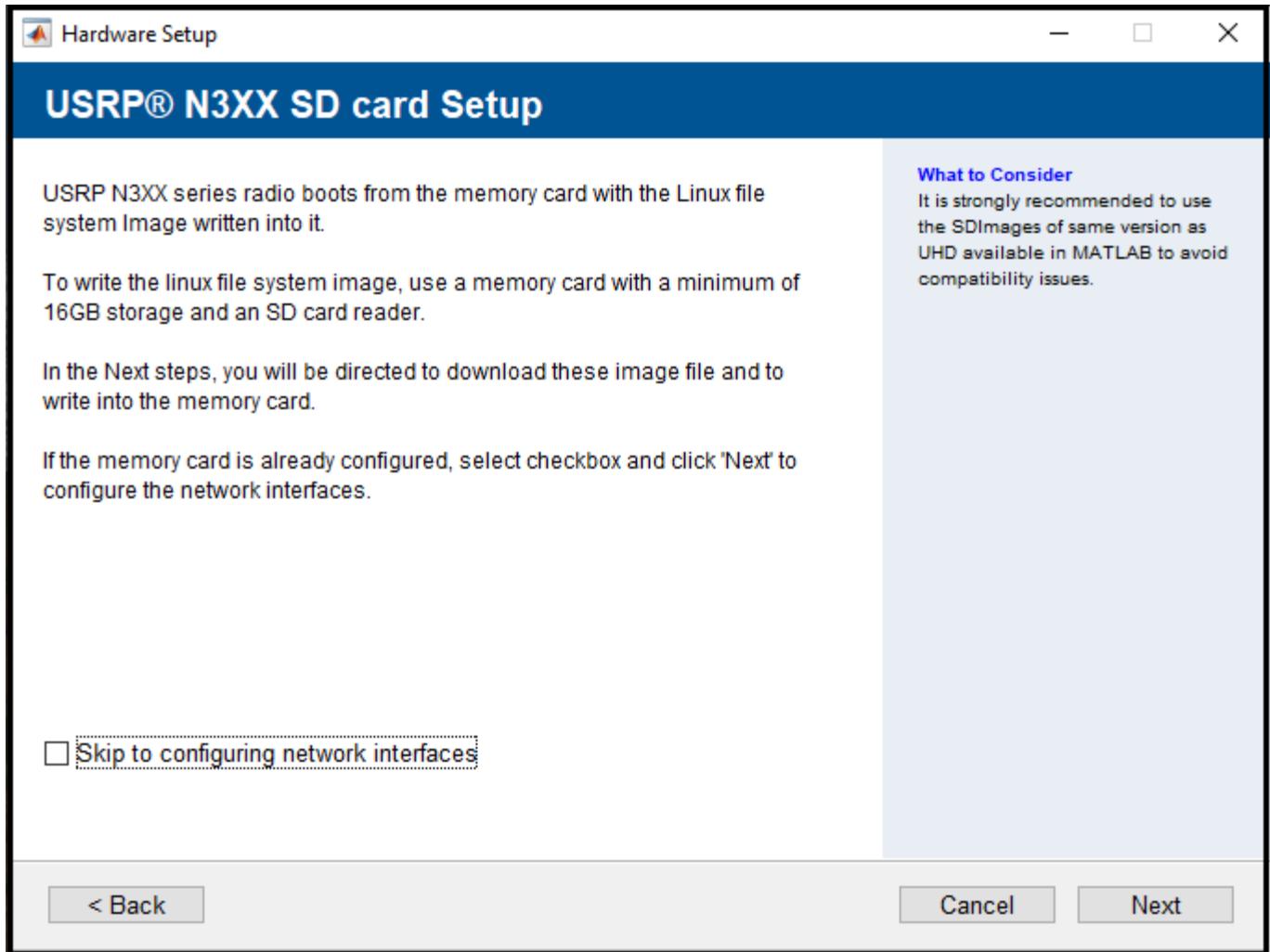
### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5

## Configure Network Interface Using Installer with No USRP N3xx Radio Connected

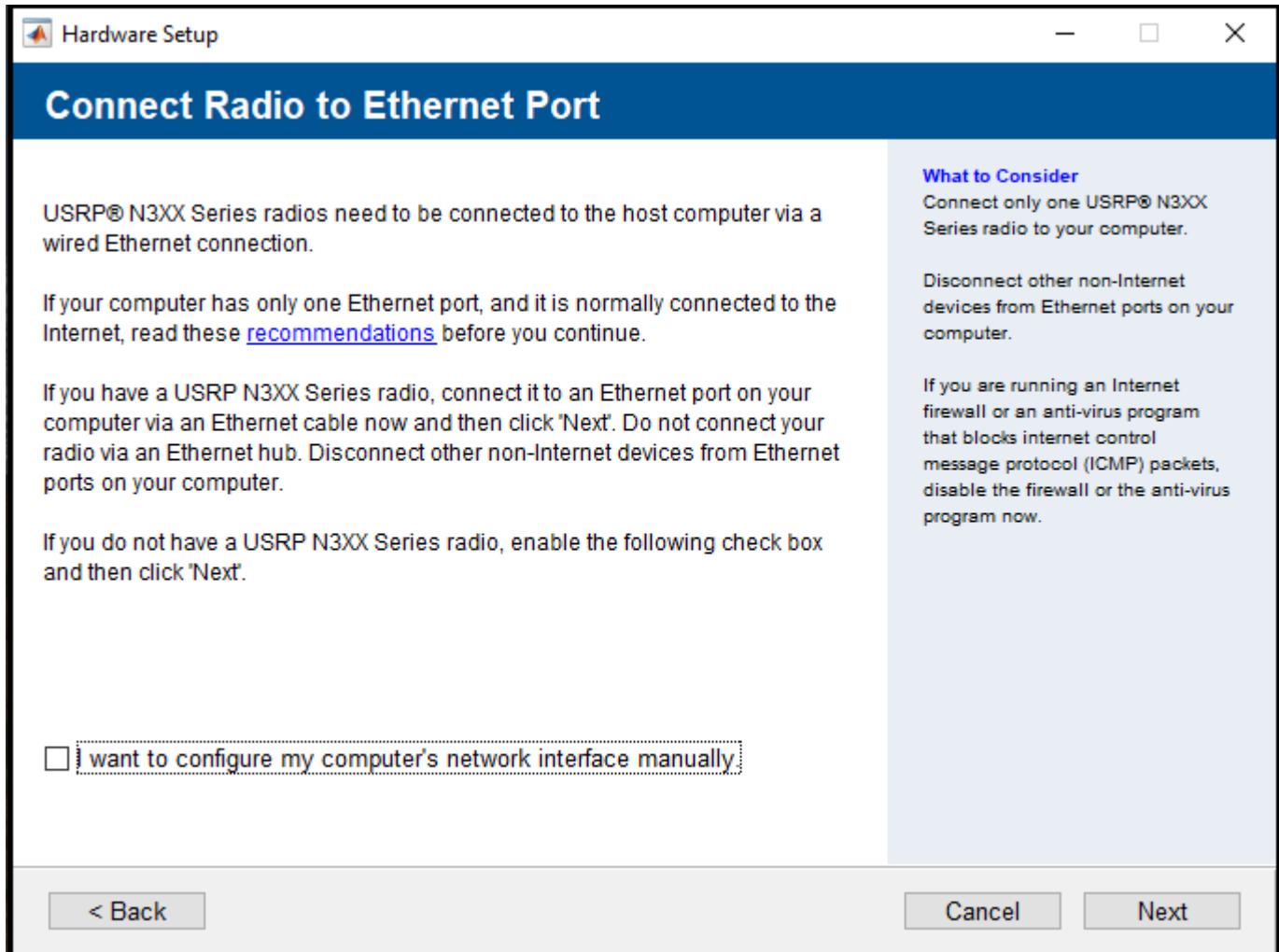
These instructions describe installer steps to configure the host computer network interface with no USRP N3xx radio connected. While making changes to a network interface, if you see a dialog box asking **Do you want to allow this app to make changes to your device?**, click **Yes** to continue.

- 1 If you have already set up an SD card with the Linux system file image, select **Skip to configuring network interfaces**. Click **Next**.



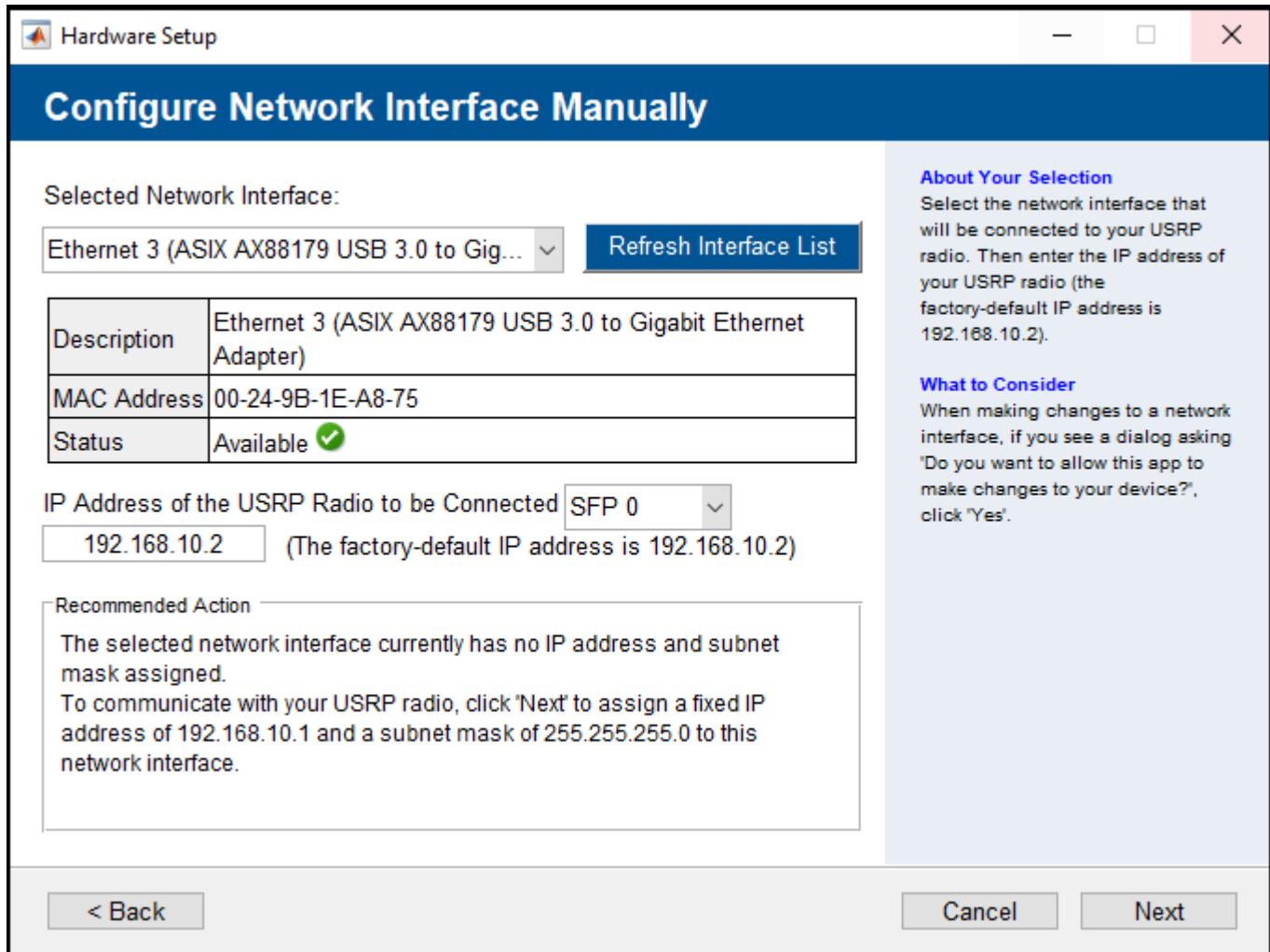
- 2 If you do not have an Ethernet-based USRP N3xx radio connected to the host computer, select **I want to configure my computer's network interface manually**, before selecting **Next**. If your computer has only one NIC, see “Using One Ethernet Port” on page 1-49.

Selecting **I want to configure my computer's network interface manually** enables you to configure the host computer without connecting an Ethernet-based USRP N3xx radio.



- 3** Select the network interface you want to configure, confirm the **Status** indicates Available, and then enter the IP address of your radio.

Click **Next** to continue.



- 4 Proceed to the FPGA image screen in the “Configure Host Computer for Ethernet-Based USRP N3xx Radio Connection” on page 1-57 instructions.

## See Also

### More About

- “Guided USRP Radio Support Package Hardware Setup” on page 1-5



# Common Problems and Fixes

---

## Common Problems and Fixes

### In this section...

- “Cannot Log into Computer” on page 2-2
- “Ethernet Subnet Conflict” on page 2-2
- “Ping command times out” on page 2-3
- “Firmware is incompatible with host build” on page 2-3
- “USRP radio is busy” on page 2-3
- “USRP radio is not responding” on page 2-4
- “Unexpected number of samples in burst reception” on page 2-5
- “Buffer could not be resized” on page 2-6
- “UHD driver cannot set thread priorities” on page 2-6
- “Function findsdru throws error” on page 2-6
- “Getting overruns or underruns” on page 2-8
- “Slow Response for SDR System Objects and Blocks” on page 2-8
- “No devices found” on page 2-9
- “USB 2.0 not fast enough with Bus Series radios” on page 2-9
- “8-bit Transport in Streaming Mode Causes libuhd error” on page 2-10
- “Burst Mode Failure” on page 2-10

## Cannot Log into Computer

### Problem

The host computer has only one Ethernet connection available, and you disconnected the internet connection to use the network interface card (NIC) for the host-to-radio connection. Then, you did not reconnect to the Internet using that NIC before logging out.

### Possible Solution

For more information, see “Using One Ethernet Port” on page 1-49.

## Ethernet Subnet Conflict

### Problem

The host-to-radio connection cannot be established because the IP address of the radio is on a subnet that is being used by the host computer for another Ethernet port or for the wireless network interface.

### Possible Solution

Follow the directions provided at “Resolving Ethernet Subnet Conflict” on page 1-52.

## Ping command times out

### Problem

Ping command returns a message indicating it cannot reach the hardware.

### Possible Solution

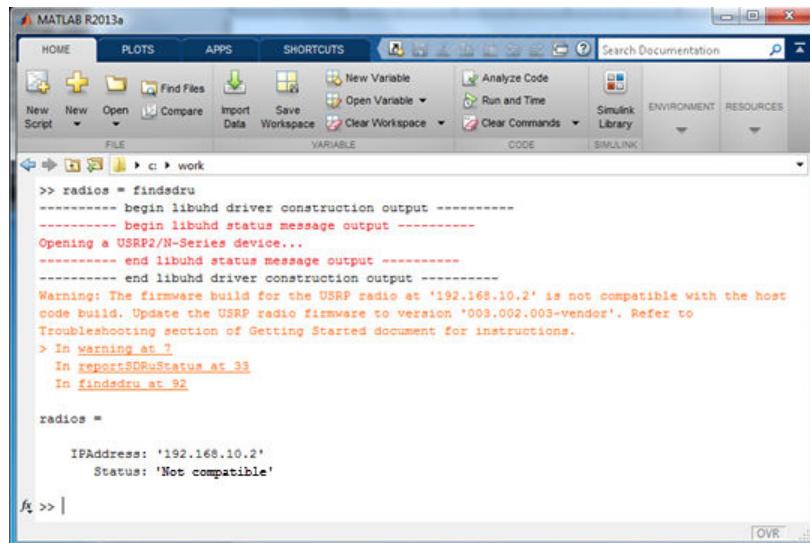
You can try any of the following:

- “Check Ethernet Configuration” on page 1-44
- Check that your firewalls are either disabled or set up to pass data from the subnet configured on your radio. The factory default subnet configuration on the radio is 192.168.10.X.

## Firmware is incompatible with host build

### Problem

The function `findsdr` gives the following warning message: 'Not compatible'.



A screenshot of the MATLAB R2013a interface. The command window displays the following text:

```

>> radios = findsdr
----- begin libuhd driver construction output -----
----- begin libuhd status message output -----
Opening a USRP2/N-Series device...
----- end libuhd status message output -----
----- end libuhd driver construction output -----
Warning: The firmware build for the USRP radio at '192.168.10.2' is not compatible with the host
code build. Update the USRP radio firmware to version '003.002.003-vendor'. Refer to
Troubleshooting section of Getting Started document for instructions.
> In warning at 7
  In reportSDRStatus at 33
  In findsdr at 92

radios =

```

The warning message is highlighted in red: "Warning: The firmware build for the USRP radio at '192.168.10.2' is not compatible with the host code build. Update the USRP radio firmware to version '003.002.003-vendor'. Refer to Troubleshooting section of Getting Started document for instructions."

The firmware installed on the USRP radio is incompatible with the UHD software version of the support package.

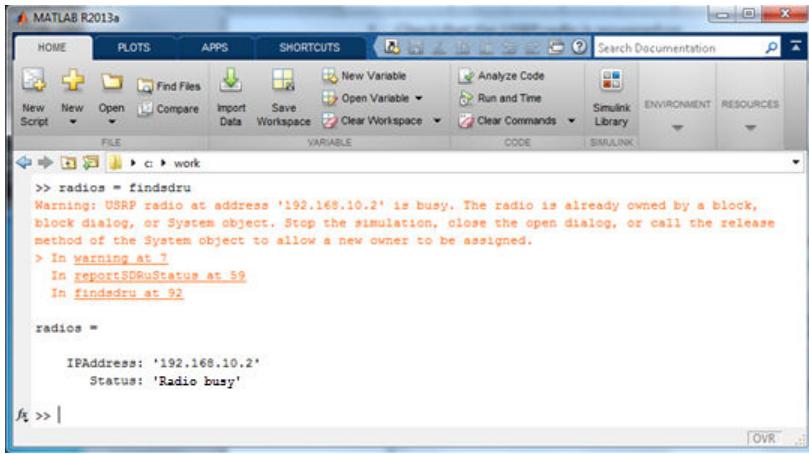
### Possible Solution

Update the firmware for your USRP radio. See “USRP Radio Firmware Update” on page 1-53.

## USRP radio is busy

### Problem

MATLAB returns the following warning message from a call to function `findsdr`: Busy.



```
>> radios = findsdru
Warning: USRP radio at address '192.168.10.2' is busy. The radio is already owned by a block,
block dialog, or System object. Stop the simulation, close the open dialog, or call the release
method of the System object to allow a new owner to be assigned.
> In warning at 7
  In reportSDRStatus at 59
  In findsdru at 92

radios =
IPAddress: '192.168.10.2'
Status: 'Radio busy'

ft >> |
```

The USRP radio is in use by another MATLAB or Simulink entity. USRP radios can become busy when any of the following conditions occur:

- A Simulink simulation is in progress.
- A receiver or transmitter block mask is open.
- A locked receiver or transmitter System object™ is in memory.

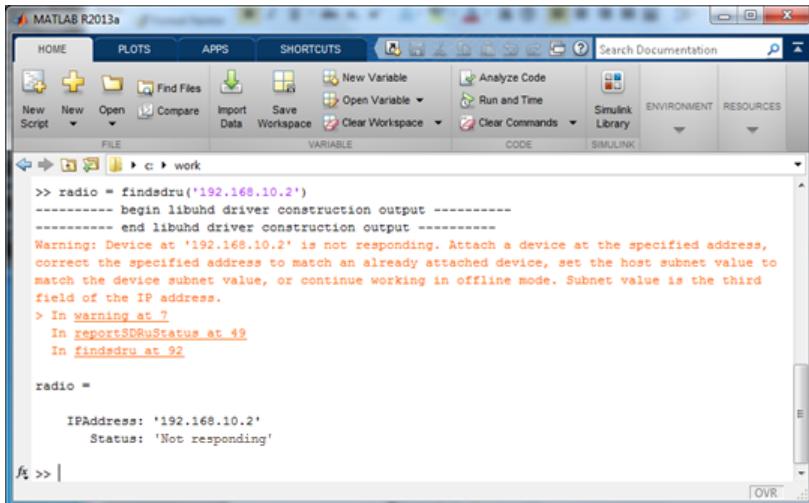
### Possible Solution

You can release the radio by stopping the simulation, closing the block, or calling the `release` method of the System object.

## USRP radio is not responding

### Problem

The function `findsdru(IPAddress)`, where *IPAddress* is the IP address of a USRP radio, returns the following warning message: Not responding.



```
>> radio = findsdru('192.168.10.2')
----- begin libuhd driver construction output -----
----- end libuhd driver construction output -----
Warning: Device at '192.168.10.2' is not responding. Attach a device at the specified address,
correct the specified address to match an already attached device, set the host subnet value to
match the device subnet value, or continue working in offline mode. Subnet value is the third
field of the IP address.
> In warning at 7
  In reportSDRStatus at 49
  In findsdru at 92

radio =
IPAddress: '192.168.10.2'
Status: 'Not responding'

ft >> |
```

## Possible Solution

This warning indicates that your subnet configuration is incorrect. For example, if the USRP radio has an IP address of 192.168.10.2, but the host IP address is on another subnet and has an IP address of 192.168.X.1, where X is a number other than 10.

Correct the host IP address so that it matches the subnet value of the USRP radio as described in “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13 .

Alternatively, there may be an Ethernet connection problem between the host computer and the USRP radio. See “Check Ethernet Configuration” on page 1-44.

## Unexpected number of samples in burst reception

### Problem

You may get an error message similar to the following while using burst mode:

```
Could not execute UHD driver command in 'receiveData_c':
libmwusrp_uhd_capi:receiveData:ErrWrongRecvSize
Did not receive expected number of samples in a burst reception.
This is likely due to overflow within the burst.
Use 'sysctl' to update the OS socket buffer size.
Expected: 2752000
Found : 94120
```

This error occurs when the MATLAB or Simulink software does not receive the requested number of samples from the USRP radio.

### Possible Reasons and Solutions

The following are known causes of this problem:

- On Linux systems, the OS socket buffer size may not be large enough for proper communication. Increase the socket size as described in the Ettus Research™ UHD - Transport (Sockets).
- The Ethernet card is not able to provide high-speed communication. You may want to try Intel® chipsets, which can provide high-quality connection in such cases.
- The firewall or virus protection program on your system may be blocking or slowing down your connection. Turning off the firewall or virus program may eliminate this problem.

---

**Note** Turning off your firewall may expose your host computer to unauthorized access through the Internet.

- Some laptops may lose their Ethernet settings when the Ethernet connection is interrupted, for example when power cycling the USRP device. Check the Ethernet connection settings as described in “Configure Host Computer for Ethernet-Based Radio Connection” on page 1-13.
- With laptops, try connecting the laptop to a power supply. Most laptops are configured for better battery life and compromise processing performance when they are not connected to a power supply. For both battery mode and AC power supply mode, make sure that you have a power setting corresponding to maximum processing performance. Some laptop manufacturers also provide advanced power settings to help chose a plan for maximum CPU performance.

## Buffer could not be resized

### Problem

You may see one of the following messages from the UHD driver in the MATLAB command window.

The recv buffer could not be resized:

```
----- begin libuhd warning message output -----
The recv buffer could not be resized sufficiently.
Target sock buff size: 50000000 bytes.
Actual sock buff size: 131071 bytes.
See the transport application notes on buffer resizing.
Please run: sudo sysctl -w net.core.rmem_max=50000000
----- end libuhd warning message output -----
```

The send buffer could not be resized:

```
----- begin libuhd warning message output -----
The send buffer could not be resized sufficiently.
Target sock buff size: 1048576 bytes.
Actual sock buff size: 131071 bytes.
See the transport application notes on buffer resizing.
Please run: sudo sysctl -w net.core.wmem_max=1048576
----- end libuhd warning message output -----
```

### Possible Solution

If you encounter either of these messages, run the sysctl commands provided in the message in a Linux shell.

## UHD driver cannot set thread priorities

### Problem

You may see a warning stating that the UHD driver was not able to set the thread priority.

### Possible Solution

This warning is harmless. You can get more information on this subject at Thread priority scheduling.

## Function findsdru throws error

### Problem

The function `findsdru` may crash or throw an error similar to the following:

```
??? Invalid MEX-file
<SDRUInstallRoot>/bin/glnxa64/usrp_uhd_mapi.mexa64':
<SDRUInstallRoot>/bin/glnxa64/libmwusrp_uhd_capi.so:
    undefined symbol: _ZN3uhd7warning16register_handlerERKSsRKN5boost8functionIFvSsEEE
Error in ==> mapiPrivate at 19
[retStr, errStat, errStr] = usrp_uhd_mapi(cmd);
Error in ==> findsdru at 25
[flatAddrList, errStat, errStr] = mapiPrivate('findsdru');
```

## Possible Reasons and Solutions

This type of error usually occurs when:

- A correct version of the libuhd or Boost libraries was *not* loaded.
- An incorrect version of these libraries *was* loaded.

This situation may happen if either the system path does not contain the correct path information for these libraries or a previously installed version of these libraries shadows the Support Package for USRP Radio required libraries.

You can diagnose this issue by following these steps:

- 1 Navigate to <SDRuInstallRoot>/bin/glnxa64 within the MATLAB command window. For example:

```
cd c:/work/sdr/sdru/bin/glnxa64
```

- 2 Type the following command:

```
!ldd libmwusrp_uhd_capi.so
```

You should see messages similar to the following:

```
linux-vdso.so.1 => (0x00007fffff35ff000)
libuhd.so.003=>
<SDRuInstallRoot>/glnxa64/commusrp/bin/glnxa64/.libuhd.so.003 (0x00007fc9476bb000)

libstdc++.so.6=>
<MATLABROOT>/sys/os/glnxa64/libstdc++.so.6 (0x00007fc9473b4000)

libm.so.6 => /lib/libm.so.6 (0x00007fc947113000)

libgcc_s.so.1 =>
<MATLABROOT>/sys/os/glnxa64/libgcc_s.so.1 (0x00007fc946efd000)

libpthread.so.0 => /lib/libpthread.so.0 (0x00007fc946ce0000)

libc.so.6 => /lib/libc.so.6 (0x00007fc94697f000)

libboost_date_time.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_date_time.so.1.44.0
(0x00007fc94676d000)

libboost_filesystem.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_filesystem.so.1.44.0
(0x00007fc946549000)

libboost_program_options.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_program_options.so.1.44.0
(0x00007fc9462ef000)

libboost_regex.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_regex.so.1.44.0 (0x00007fc945fdd000)

libboost_system.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_system.so.1.44.0 (0x00007fc945dd9000)

libboost_thread.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_thread.so.1.44.0 (0x00007fc945bc2000)

libboost_unit_test_framework.so.1.44.0=><MATLABROOT>/bin/glnxa64/libboost_unit_test_framework.so.1.44.0 (0x00007fc945906000)

librt.so.1 => /lib/librt.so.1 (0x00007fc9456fd000)

libdl.so.2 => /lib/libdl.so.2 (0x00007fc9454f9000)

/lib64/ld-linux-x86-64.so.2 (0x00007fc947d73000)

libicuuc.so.44=><MATLABROOT>/bin/glnxa64/libicuuc.so.44 (0x00007fc945196000)

libicui18n.so.44=><MATLABROOT>/bin/glnxa64/libicui18n.so.44(0x00007fc944d9e000)

libicudata.so.44=><MATLABROOT>/bin/glnxa64/libicudata.so.44 (0x00007fc943d5e000)
```

Make sure that all the boost libraries are pulled from the MATLAB installation locations but not from the system (for example, /usr/lib or /lib), or some other local installation.

If you do not get these results, for example if either the libraries are not being found or different versions of boost or libuhd are found, then the LD\_LIBRARY\_PATH is likely incorrect. You can check the value of the LD\_LIBRARY\_PATH by typing the following command in the MATLAB command window.

```
getenv('LD_LIBRARY_PATH')

ans =
<MATLABR00T>/sys/os/glnxa64:
<MATLABR00T>/bin/glnxa64:
<MATLABR00T>/extern/lib/glnxa64:
<MATLABR00T>/sys/java/jre/glnxa64/jre/lib/amd64/native_threads:
<MATLABR00T>/sys/java/jre/glnxa64/jre/lib/amd64/server:
<MATLABR00T>/sys/java/jre/glnxa64/jre/lib/amd64:
<SDRuInstallRoot>/glnxa64/commusrp/bin/glnxa64
```

This example shows the default value as created by MATLAB and the `setupsdr` function.

## Getting overruns or underruns

### Problem

Model is not running in real time.

### Possible Solution

If your model is not running in real time, you can:

- Use Burst mode. See “Burst-Mode Buffering” on page 5-2.
- Reduce transport data rate to 8-bit. See “Change Transport Data Rate” on page 4-29.
- Use vector-based processing
- Accelerate with code generation

## Slow Response for SDR System Objects and Blocks

### Problem

It seems to take a long time to display an SDRu object, or to display an SDRu object property, or to open an SDRu block mask.

### Possible Solution

When any of those operations occur, the object or block is communicating with the USRP radio to access actual device values. This process can take some seconds, and is to be expected.

## No devices found

### Problem

When you plug in a radio and run function `findsdr`, the function may return a message stating "No devices found". However, the firmware image is loaded automatically and it shows that the host computer is able to recognize the radio.

### Possible Solution

Ettus Research has confirmed that this issue is known to occur with certain models/revisions of USB controllers. The solution is to disconnect then reconnect the radio, possibly a few times.

```
===== Connect radio to host computer =====
```

```
>> x = findsdr
Loading firmware image: /Users/Shared/supportpackages/usrpradio/toolbox/shared/sdr/sdru/uhdapps/...
x =
Platform: ''
IPAddress: ''
SerialNum: ''
Status: 'No devices found'
```

```
===== Disconnect the radio and re-connect =====
```

```
>> x = findsdr
Loading firmware image: /Users/Shared/supportpackages/usrpradio/toolbox/shared/sdr/sdru/uhdapps/...
Loading FPGA image: /Users/Shared/supportpackages/usrpradio/toolbox/shared/sdr/sdru/uhdapps/image...
x =
Platform: 'B210'
IPAddress: ''
SerialNum: 'ECR04ZDBT'
Status: 'Success'
```

## USB 2.0 not fast enough with Bus Series radios

### Problem

USB 2.0 connection is not fast enough for certain high sample rate applications when using a Bus Series radio.

### Possible Solution

You can use a USB 3.0 connection to get a more reliable connection for high-speed needs. See <https://kb.ettus.com/B200/B210/B200mini/B205mini#FAQ>.

## 8-bit Transport in Streaming Mode Causes libuhd error

### Problem

A UHD exception occurs when you attempt 8-bit transport in streaming mode using a Bus Series radio.

```
Error using comm.SDRuReceiver/stepImpl  
Could not execute UHD driver command in 'receiveData_c': libmwusrp_uhd_capi:receiveData:ErrBadPacket The  
communication transport received a mal-formed packet.
```

### Possible Solution

You can either use burst mode buffering or change the transport data type to 16-bit.

- To use burst mode, see “Burst-Mode Buffering” on page 5-2.
- To change transport data type to 16-bit, see “Change Transport Data Rate” on page 4-29.

## Burst Mode Failure

### Problem

You encounter the following error:

```
Error using comm.SDRuReceiver/stepImpl  
Could not execute UHD driver command in 'receiveData_c': libmwusrp_uhd_capi:receiveData:ErrWrongRecvSize  
Did not receive expected number of samples in a burst reception.  
This is likely due to overflow within the burst. Use 'sysctl' to update the OS socket buffer size.  
Expected: 4000000  
Found   : 163741
```

### Possible Solution

- Change transport data type to `int8`. This change can possibly double the achievable sample rate. See “Change Transport Data Rate” on page 4-29,
- If you have a Bus Series radio, make sure that you use a USB 3 connection.

See the recommended USB 3.0 controllers from Ettus Research: [https://kb.ettus.com/About\\_USRP\\_Bandwidths\\_and\\_Sampling\\_Rates](https://kb.ettus.com/About_USRP_Bandwidths_and_Sampling_Rates).

You can use the `benchmark_rate` utility from UHD to test whether a transport speed can be sustained between the radio and the host PC. This example demonstrates testing the transmit rate:

```
call_uhd_app('benchmark_rate','--tx_rate 20e6','-echo');  
  
linux; GNU C++ version 4.7.2; Boost_105600; UHD_003.009.001-vendor  
  
Creating the usrp device with: ...  
-- Detected Device: B210  
-- Operating over USB 3.  
-- Initialize CODEC control...  
-- Initialize Radio control...  
-- Performing register loopback test... pass  
-- Performing register loopback test... pass  
-- Performing CODEC loopback test... pass  
-- Performing CODEC loopback test... pass  
-- Asking for clock rate 16.000000 MHz...
```

```
-- Actually got clock rate 16.000000 MHz.
-- Performing timer loopback test... pass
-- Performing timer loopback test... pass
-- Setting master clock rate selection to 'automatic'.
Using Device: Single USRP:
    Device: B-Series Device
    Mboard 0: B210
    RX Channel: 0
        RX DSP: 0
        RX Dboard: A
        RX Subdev: FE-RX2
    RX Channel: 1
        RX DSP: 1
        RX Dboard: A
        RX Subdev: FE-RX1
    TX Channel: 0
        TX DSP: 0
        TX Dboard: A
        TX Subdev: FE-TX2
    TX Channel: 1
        TX DSP: 1
        TX Dboard: A
        TX Subdev: FE-TX1

-- Asking for clock rate 20.000000 MHz...
-- Actually got clock rate 20.000000 MHz.
-- Performing timer loopback test... pass
-- Performing timer loopback test... pass
Testing transmit rate 20.000000 Msps on 1 channels
```

Benchmark rate summary:

```
Num received samples:      0
Num dropped samples:      0
Num overflows detected:   0
Num transmitted samples:  200058544
Num sequence errors:      0
Num underflows detected:  0
```

This second example demonstrates testing the receive rate:

```
call_uhd_app('benchmark_rate', '--rx_rate 20e6', '-echo');

linux; GNU C++ version 4.7.2; Boost_105600; UHD_003.009.001-vendor
```

```
Creating the usrp device with: ...
-- Detected Device: B210
-- Operating over USB 3.
-- Initialize CODEC control...
-- Initialize Radio control...
-- Performing register loopback test... pass
-- Performing register loopback test... pass
-- Performing CODEC loopback test... pass
-- Performing CODEC loopback test... pass
-- Asking for clock rate 16.000000 MHz...
-- Actually got clock rate 16.000000 MHz.
-- Performing timer loopback test... pass
```

```
-- Performing timer loopback test... pass
-- Setting master clock rate selection to 'automatic'.
Using Device: Single USRP:
Device: B-Series Device
Mboard 0: B210
RX Channel: 0
RX DSP: 0
RX Dboard: A
RX Subdev: FE-RX2
RX Channel: 1
RX DSP: 1
RX Dboard: A
RX Subdev: FE-RX1
TX Channel: 0
TX DSP: 0
TX Dboard: A
TX Subdev: FE-TX2
TX Channel: 1
TX DSP: 1
TX Dboard: A
TX Subdev: FE-TX1

-- Asking for clock rate 20.000000 MHz...
-- Actually got clock rate 20.000000 MHz.
-- Performing timer loopback test... pass
-- Performing timer loopback test... pass
Testing receive rate 20.000000 Msps on 1 channels

Benchmark rate summary:
Num received samples: 199989048
Num dropped samples: 0
Num overflows detected: 0
Num transmitted samples: 0
Num sequence errors: 0
Num underflows detected: 0
```

- With laptops, try connecting the laptop to a power supply. Most laptops are configured for better battery life and compromise processing performance when they are not connected to a power supply. For both battery mode and AC power supply mode, make sure that you have a power setting corresponding to maximum processing performance. Some laptop manufacturers also provide advanced power settings to help chose a plan for maximum CPU performance.

# Hardware Discovery

---

## Find USRP Devices Connected To Computer

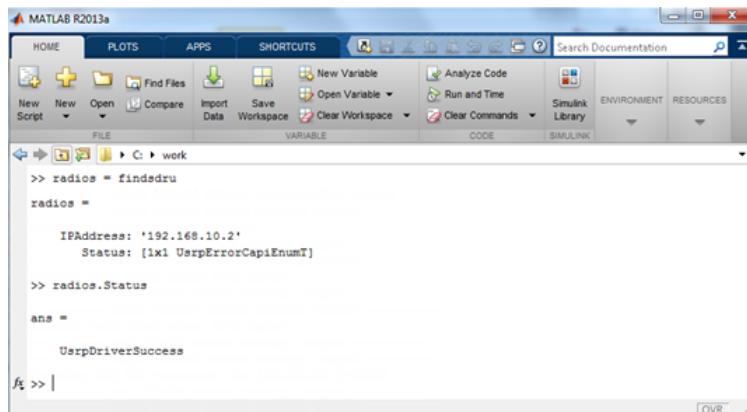
If you are not sure which USRP radio devices are connected to your computer, you can use a helper function to find them.

Type the following at the MATLAB command line:

```
radios = findsdru
```

The variable, `radios`, is a structure that contains information on the USRP radios connected to the host computer.

- If you get a successful status, it means that MATLAB can communicate with the USRP radio and the radio is ready to be used.
- If the function cannot find a radio, MATLAB returns an empty IPAddress field or a status other than Success. See the section on “Common Problems and Fixes” on page 2-2 for possible causes and solutions.
- If the function finds one or more radios, MATLAB displays a message similar to the following.



A screenshot of the MATLAB R2013a interface. The command window shows the following text:

```
MATLAB R2013a
HOME PLOTS APPS SHORTCUTS Search Documentation
New New Open Compare Import Data Save Workspace New Variable Open Variable Run and Time
FILE VARIABLE CODE SIMULINK ENVIRONMENT RESOURCES
>> radios = findsdru
radios =
    IPAddress: '192.168.10.2'
    Status: [1x1 UsrpErrorCapiEnumT]
>> radios.Status
ans =
    UsrpDriverSuccess
f1 >> |
```

- The IPAddress field is the IP address of the USRP radio.
- The Status field is the status of the radio. Type `radios.Status` in the command window and press enter to see status value.

In this example, the USRP IP address is 192.168.10.2 and its status is `USRPDriverSuccess`. The successful status indicates that MATLAB can communicate with the USRP radio and the radio is ready to be used.

If you encounter difficulties communicating with the radio from MATLAB, see “Common Problems and Fixes” on page 2-2.

---

**Note** The displayed screen includes messages from the UHD driver, delimited by --- begin libuhd driver construction output --- and --- end libuhd driver construction output --- or --- begin libuhd status message output --- and --- end libuhd status message output ---. These messages are usually benign, but in some cases they may refer to steps that you can take to increase the performance of the USRP radio connection. Follow the recommended steps when suitable. You can also refer to the section “Common Problems and Fixes” on page 2-2.

---

After you have found radios attached to your computer, you can use the function `probesdru` to get detailed information about a particular USRP radio.



# Radio Management

---

- “Check Radio Connection” on page 4-2
- “Run in Offline Mode” on page 4-4
- “Query and Set IP Addresses” on page 4-5
- “Radio Configuration” on page 4-7
- “Single Channel Input and Output Operations” on page 4-8
- “Multiple Channel Input and Output Operations” on page 4-14
- “Detect Underruns and Overruns” on page 4-22
- “Data Frame Lengths” on page 4-26
- “Apply Conditional Execution” on page 4-27
- “Change Transport Data Rate” on page 4-29
- “Transmit and Receive Using External Clock” on page 4-34
- “Desired Vs. Actual Parameter Values” on page 4-37
- “Supported Data Types” on page 4-40

## Check Radio Connection

### In this section...

"Block Connection" on page 4-2

"System Object Connection" on page 4-2

### Block Connection

You can verify that an SDRu block is connected to USRP radio by examining the block mask.

- 1** Open model with an SDRu Transmitter or SDRu Receiver block.
- 2** To open the block mask, double-click the block.

If the block is connected, then the Hardware panel shows values for these parameters.

- Motherboard
- Receiver subdevice
- Transmitter subdevice
- Minimum center frequency
- Maximum center frequency
- Minimum gain
- Maximum gain
- Gain step size

If the block is not connected, the Hardware panel shows the following message: "No device found at the specified IP address. Connect a USRP device to your host computer at the specified IP address or continue to work in off-line mode. You may need to close and reopen the dialog to detect the device."

A block can connect to only one subdevice (transmitter or receiver) at a time, on the same IP address.

### System Object Connection

To verify that your SDRu System object is connected to the USRP hardware, follow these steps:

- 1** Construct a transmitter or receiver System object.

```
radio=comm.SDRuTransmitter
```

or

```
radio=comm.SDRuReceiver
```

- 2** Enter these commands.

```
a = findsdru()  
a =  
    struct with fields:
```

```
Platform: 'N200/N210/USRP2'  
IPAddress: '192.168.30.7'  
SerialNum: 'F2A02F'  
Status: 'Success'
```

This function finds the IP address and status of each USRP radio connected to the host.

```
a(1).IPAddress  
ans =  
192.168.30.7  
a(1).Status  
ans =  
Success
```

A System object can connect to only one subdevice (transmitter or receiver) at a time, on the same IP address. If you change property values of an SDRu System object and display the actual values, the object connects and then immediately disconnects once the properties are displayed. SDRu System objects connect to a subdevice when you call the object as a function (or call the step method), and stays connected until you call the release method.

## See Also

### More About

- “Verify MATLAB Connection to USRP Radio” on page 1-22
- “Verify Hardware Connection” on page 1-44

## Run in Offline Mode

You can program an application or generate code with the default Receiver or Transmitter block or System object without having to connect to a USRP device.

- 1** Use `findsdru` function to see available device IP addresses.
- 2** Set the **IP address** block parameter or `IPAddress` System object property to any valid IP address returned by the function (whether the device is connected or not). You can also leave the address as the default (192.168.10.2).
- 3** Use the SDRu Receiver or Transmitter block or System object.

**Note** For System objects, tab compilation on the `IPAddress` property does not work to find available IP addresses. Use `findsdru` function to see available device IP addresses.

---

## See Also

**Functions**  
`findsdru`

# Query and Set IP Addresses

## In this section...

- “Block Parameter IP Address” on page 4-5
- “System Object Property IP Address” on page 4-5
- “Set IP Address with setsdruip Function” on page 4-5

## Block Parameter IP Address

If the SDRu Transmitter or SDRu Receiver block is connected to the USRP device, the valid IP address appears in the **IP address** parameter in the block mask. You can also change the IP address of the connected USRP device.

- 1 Open model with an SDRu Transmitter or SDRu Receiver block.
- 2 To open the block mask, double-click the block.
- 3 Type a dotted quad IP address in the **IP address** field present under Radio Connection.

## System Object Property IP Address

If the SDRu Receiver or Transmitter System object is connected to USRP device, you can retrieve the valid IP address by querying the **IPAddress** property.

For example:

```
radio IPAddress
ans =
192.168.10.2
```

You can change the IP address of the connected USRP device by using the same property.

```
radio IPAddress = '192.168.10.4'
radio =
struct with fields:
    IPAddress: '192.168.10.4'
```

## Set IP Address with setsdruip Function

The function **setsdruip** sets the IP address of the USRP device at an IP address that you provide. The current IP address is replaced with the new IP address.

For example, set the IP address of the USRP radio at IP address 192.168.30.22 to 192.168.30.20 with this command:

```
setsdruip('192.168.30.22', '192.168.30.20');
```

## See Also

**Functions**  
setsdruip

## More About

- “Check Host-to-Radio Connection” on page 1-44

# Radio Configuration

The Communications Toolbox Support Package for USRP Radio enables your simulation to communicate with the following USRP radio hardware platforms from Ettus Research.

USRP Bus Series (B-series) radios communicate with the host PC using a USB connection and have an Analog Devices® AD9364 RF front end.

USRP Networked Series (N-series) radios communicate with the host PC using an Ethernet connection. N-series radios can be configured with various RF daughterboards for SISO or MIMO communications.

USRP X Series radios are high-performance, scalable software defined radio (SDR) platforms for designing and deploying next generation wireless communications systems. X-series radios communicate with the host PC using an Ethernet connection. X-series radios can be configured with various RF daughterboards for SISO or MIMO communications.

The valid range for radio settings, such as center frequency, LO offset, transmitter gain, and receiver gain, depends on your radio configuration. For detailed information about the various X-series, N-series, or B-series radio configurations, see the Ettus Research™ website.

## See Also

### More About

- “Single Channel Input and Output Operations” on page 4-8
- “Multiple Channel Input and Output Operations” on page 4-14

### External Websites

- Ettus Research™

# Single Channel Input and Output Operations

## In this section...

["Perform SISO Operations with SDRu System Objects" on page 4-8](#)

["Perform SISO Operations with SDRu Blocks" on page 4-9](#)

N200, N210, USRP2, and B200 radios are single transceiver designs, so the default channel mapping value is read-only. B210, X300, X310, N300, N320, and N321 radios support two channels for data transmission and reception, and N310 radios support four channels for data transmission and reception. To configure B210, X300, X310, N300, N310, N320, or N321 radios for single input single output (SISO) operations, use a single channel (1, 2, 3, or 4).

## Perform SISO Operations with SDRu System Objects

This example shows how to set channel 2 of a B210 radio for SISO operations with an SDRu transmitter System object™.

Create an SDRu transmitter System object for data transmission. Configure a B210 radio with its serial number set to 31B92DD. Change the channel to channel 2 by setting the `ChannelMapping` property. Display the configured properties.

```
tx = comm.SDRuTransmitter(...  
    'Platform','B210', ...  
    'SerialNum','31B92DD', ...  
    'ChannelMapping',2);  
  
disp(tx)  
  
comm.SDRuTransmitter with properties:  
  
    Platform: 'B210'  
    SerialNum: '31B92DD'  
    ChannelMapping: 2  
    CenterFrequency: 2.4500e+09  
    LocalOscillatorOffset: 0  
        Gain: 8  
    PPSSource: 'Internal'  
    ClockSource: 'Internal'  
    MasterClockRate: 32000000  
    InterpolationFactor: 512  
    TransportDataType: 'int16'  
    EnableBurstMode: false
```

Create a DPSK modulator as the data source by using the `comm.DPSKModulator` System object. Inside a for loop, transmit the data using the SDRu transmitter System object.

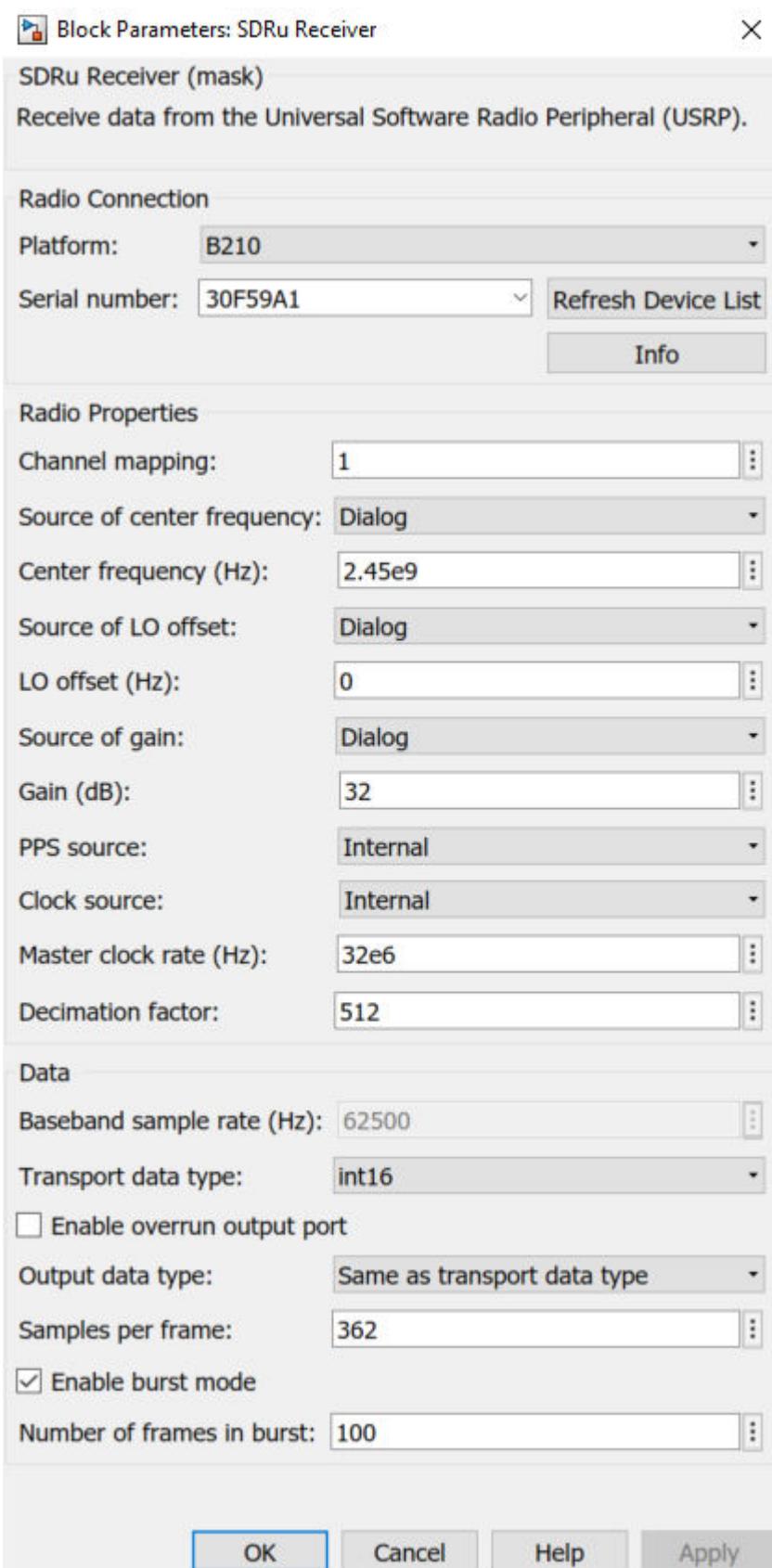
```
mod = comm.DPSKModulator('BitInput',true);  
for counter = 1:20  
    data = randi([0 1],30,1);  
    modSignal = mod(data);  
    tx(modSignal);  
end
```

You can perform these same steps with a `comm.SDRuReceiver` System object instead of the SDRu transmitter System object.

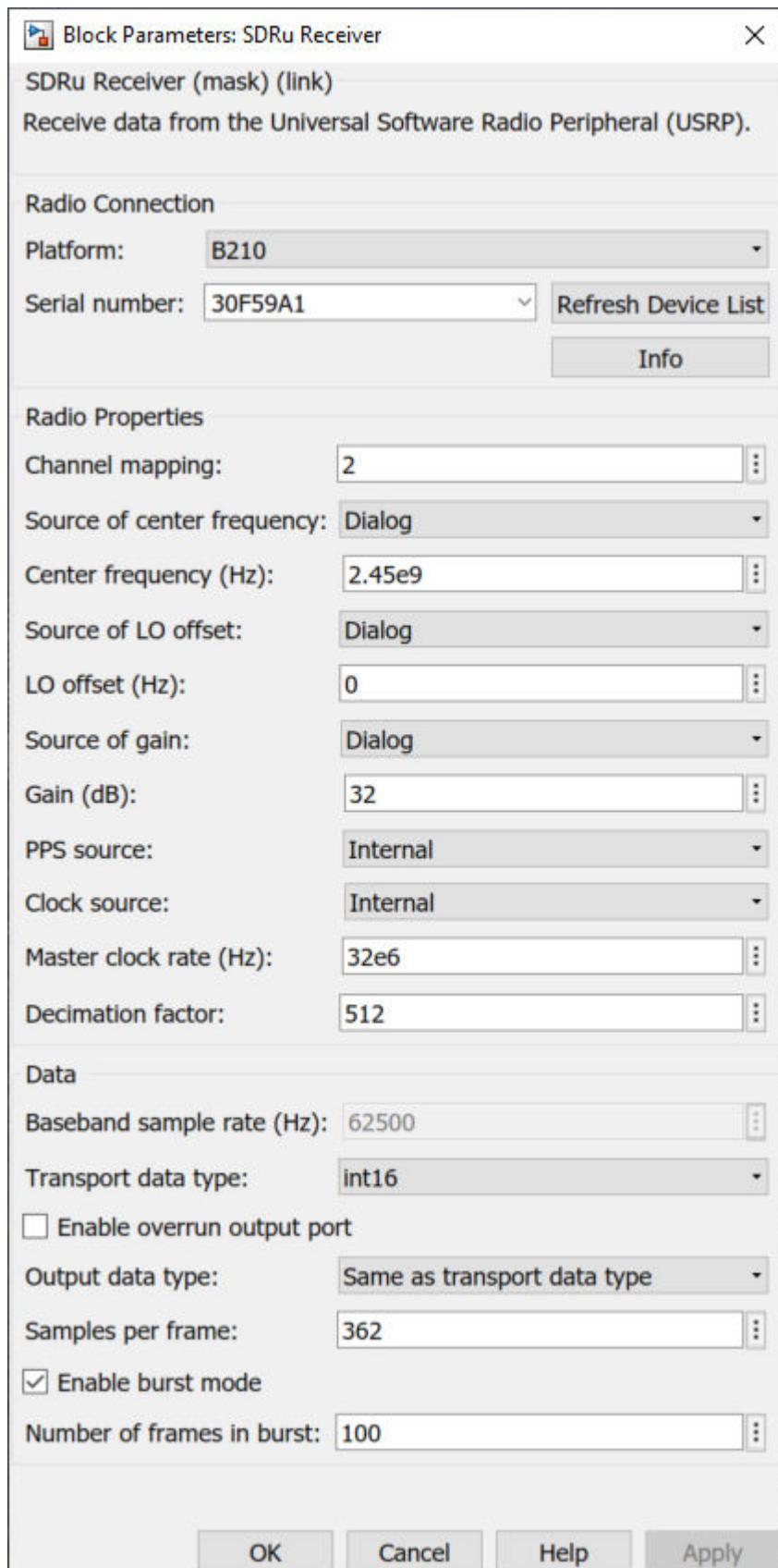
## Perform SISO Operations with SDRu Blocks

This example shows how to set channel 2 of a B210 radio for SISO operations with an SDRu Receiver block.

- 1 Open the mask of an SDRu Receiver block. Set the **Platform** parameter to **B210**. This figure shows the mask for an SDRu Receiver block.



- 2 Change the channel to channel 2 in SISO mode by setting the **Channel mapping** parameter to 2. Then, click **OK**.



You can perform these same steps with the SDRu Transmitter block instead of the SDRu Receiver block.

## See Also

### More About

- “Multiple Channel Input and Output Operations” on page 4-14
- “Single and Multiple Channel Output” on page 8-19

# Multiple Channel Input and Output Operations

## In this section...

- “About MIMO Mode” on page 4-14
- “Perform MIMO Operations with SDRu System Objects” on page 4-14
- “Perform MIMO Operations with SDRu Blocks” on page 4-16
- “Perform MIMO Operations Bundling Multiple Radios” on page 4-17

For multiple input multiple output (MIMO) operations, you can use multichannel radios or use single channel radios bundled together.

## About MIMO Mode

You can use MIMO operations to help achieve better performance in your communications system. Space-time block coding can increase the signal-to-noise ratio (SNR). Spatial multiplexing can increase data rates.

To prepare waveforms for MIMO mode, see “Multiple-Input Multiple-Output (MIMO)” and LTE Toolbox™ features.

The MIMO functionality in this support package transmits signals through the TX/RX port and receives signals through the RX2 port.

## Perform MIMO Operations with SDRu System Objects

### Transmit over Multiple Channels with SDRu System Object

Create a System object™ for a platform that supports MIMO mode. This example uses the B210 radio.

```
txradio = comm.SDRuTransmitter('Platform','B210','SerialNum','31B92DD')

txradio =
    comm.SDRuTransmitter with properties:

        Platform: 'B210'
        SerialNum: '31B92DD'
        ChannelMapping: 1
        CenterFrequency: 2.4500e+09
        LocalOscillatorOffset: 0
        Gain: 8
        PPSSource: 'Internal'
        ClockSource: 'Internal'
        MasterClockRate: 32000000
        InterpolationFactor: 512
        TransportDataType: 'int16'
        EnableBurstMode: false
```

Set the ChannelMapping property to [1 2] to indicate that both channels are in use.

```
txradio.ChannelMapping = [1 2];
```

For the B210 radios only, change the master clock rate to any value up to the supported maximum of 30.72 MHz. This hardware limitation for using two-channel operations applies to the B210 radios only. For other radios, you can set the master clock rate to any of the supported values.

```
txradio.MasterClockRate = 16e6;
```

Create a comm.DPSKModulator System object to modulate the transmitted signals.

```
mod = comm.DPSKModulator('BitInput',true);
```

Transmit the data. The System object generates two signals, one for each channel.

```
for i = 1:5
    data1 = randi([0 1],3e4,1);
    data2 = randi([0 1],3e4,1);
    modSignal1 = mod(data1);
    modSignal2 = mod(data2);
    txradio([modSignal1 modSignal2]);
end
```

Release the System object.

```
release(txradio);
```

## Receive from Multiple Channels with SDRu System Object

Create an SDRu Receiver System object™ for a platform that supports MIMO mode. This example uses the B210 radio.

```
rxradio = comm.SDRuReceiver('Platform','B210','SerialNum','31B92DD')

rxradio =
  comm.SDRuReceiver with properties:

    Platform: 'B210'
    SerialNum: '31B92DD'
    ChannelMapping: 1
    CenterFrequency: 2.4500e+09
    LocalOscillatorOffset: 0
    Gain: 8
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 32000000
    DecimationFactor: 512
    TransportDataType: 'int16'
    OutputDataType: 'Same as transport data type'
    SamplesPerFrame: 362
    EnableBurstMode: false
```

Set the ChannelMapping property to indicate that both channels are in use.

```
rxradio.ChannelMapping = [1 2];
```

For B210 radios only, change the master clock rate to any value up to the supported maximum of 30.72 MHz. This hardware limitation for using two-channel operations applies to the B210 radios only. For other radios, you can set the master clock rate to any of the supported values.

```
rxradio.MasterClockRate = 16e6;
```

Receive the data. Because the System object uses multiple channels, the number of columns returned in **data** is 2.

```
[data,datalen] = rxradio();
```

Release the System object.

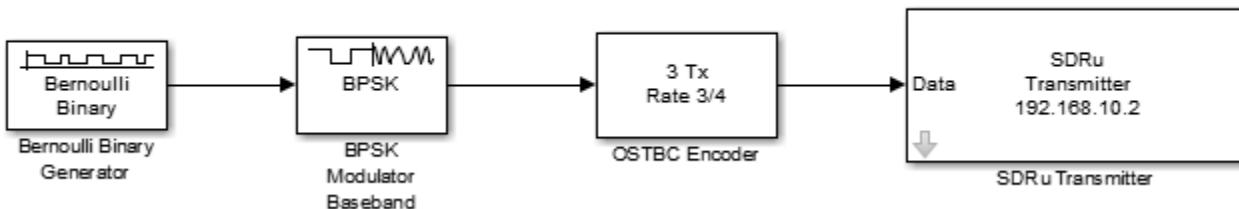
```
release(rxradio);
```

## Perform MIMO Operations with SDRu Blocks

### Transmit over Multiple Channels with SDRu Block

The SDRu Transmitter block can accept matrices at the **data** port. The number of columns is the same as the length of the **Channel mapping** parameter. If you choose to use the optional input ports for center frequency and local oscillator offset, the ports can accept scalars or row vectors of the same length as the **Channel mapping** parameter.

To create a waveform suitable for MIMO transmission, you can use Communications Toolbox blocks to create a design similar to this diagram.



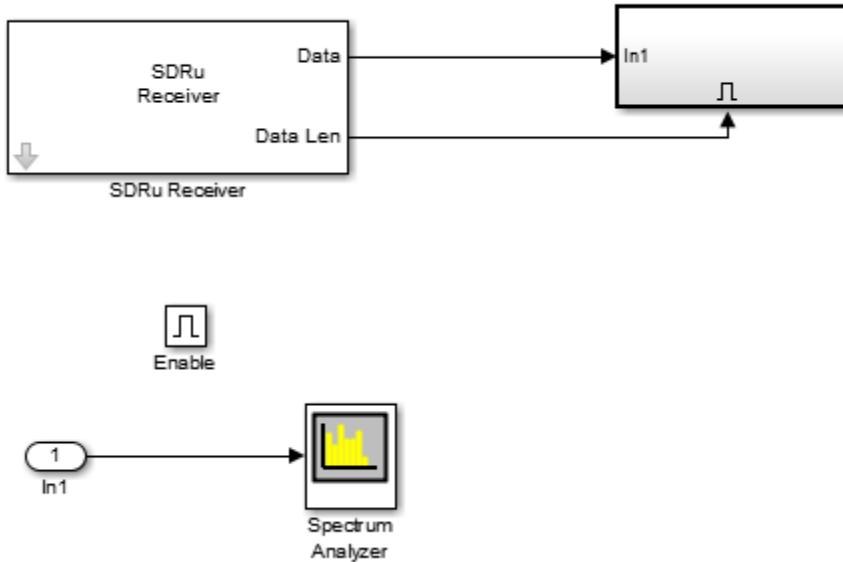
To configure the SDRu Transmitter block, in the block mask:

- 1 Set **Channel mapping** to [1 2] to use both channels.
- 2 Set the values for the **Center frequency**, **LO offset**, and **Gain** parameters as two-element row vectors. To apply the same value to both channels, specify a scalar value. For multiple channels, local oscillator (LO) offset must be 0. This requirement is due to a UHD limitation. You can specify **LO offset** as a scalar (0) or as a vector ([0 0]).
- 3 For B210 radios only, change the master clock rate to any value up to the supported maximum of 30.72 MHz. This hardware limitation for using two-channel operations applies to B210 radios only. For X300 and X310 radios, you can set the master clock rate to any of the supported values.
- 4 Click **OK**.

### Receive from Multiple Channels with SDRu Block

The SDRu Receiver block can output matrices at the **data** port. The number of columns is the same as the length of the **Channel mapping** parameter. If you choose to use the optional input ports for the center frequency, local oscillator offset, and gain, the ports can accept scalars or row vectors of the same length as the **Channel mapping** parameter.

In Simulink, design a model that can process multiple received channels , similar to the model in this figure.



In this example, **Channel mapping** in the SDRu Receiver block is defined as [1 2] to indicate that multiple channels are being used.

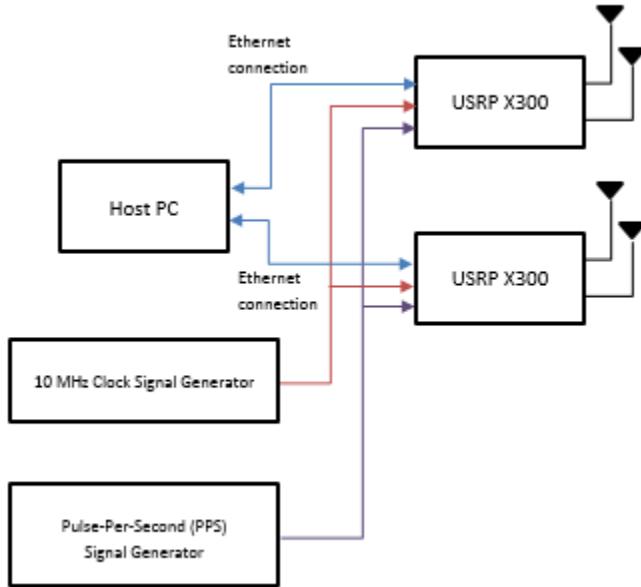
To configure the SDRu Receiver block, in the block mask:

- 1 Set **Channel mapping** to [1 2] to use both channels.
- 2 Set the values for **Center frequency**, **LO offset**, and **Gain** parameters as two-element row vectors. Alternatively, to apply the same value to both channels, specify a scalar value. For multiple channels, **LO offset** must be 0. This requirement is due to a UHD limitation. You can specify **LO offset** as scalar (0) or as a vector ([0 0]).
- 3 For B210 radios only, change the master clock rate to any value up to the supported maximum of 30.72 MHz. This hardware limitation for using two-channel operations applies to B210 radios only. For X300 and X310 radios, you can set the master clock rate to any of the supported values.
- 4 Click **OK**.

## Perform MIMO Operations Bundling Multiple Radios

### Bundle Multiple Radios

To perform MIMO operations involving more than two channels, you must bundle multiple X-series radios or multiple N-series radios. Ettus Research recommends using a common external clock signal source and pulse-per-second (PPS) signal source to bundle multiple radios to act as one radio with multiple channels. This figure shows a four channel MIMO configuration realized by bundling two X300 radios together on the host PC.



The common external 10 MHz clock signal is required for frequency synchronization of channels across bundled radios. The common external PPS signal is required for timing synchronization of channels across bundled radios.

### Receive from Multiple Radios with SDRu System Object

This example shows how to bundle multiple radios for MIMO operations with an SDRu Receiver System object.

Create an SDRu Receiver System object for a platform that supports MIMO mode. This example uses X310 radios.

```

rxRadios = comm.SDRuReceiver('Platform','X310','IPAddress','192.168.20.2,192.168.20.3')
rxRadios =
  comm.SDRuReceiver with properties:
    Platform: 'X310'
    IPAddress: '192.168.20.2,192.168.20.3'
    ChannelMapping: 1
    CenterFrequency: 2.4500e+09
    LocalOscillatorOffset: 0
    Gain: 8
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 200000000
    DecimationFactor: 512
    TransportDataType: 'int16'
    OutputDataType: 'Same as transport data type'
    SamplesPerFrame: 362
    EnableBurstMode: false
  
```

Set the channel mapping to [1 2 3 4] to indicate that four channels are in use.

```
rxRadios.ChannelMapping = [1 2 3 4];
```

Set the center frequency and gain for each channel. Display the configuration information.

```
rxRadios.CenterFrequency = [1 1.1 1.2 1.3]*1e9;
```

```
rxRadios.Gain = [5 6 7 8];
```

```
info(rxRadios)
```

```
ans =
  struct with fields:
    Mboard: {'X310' 'X310'}
    RXSubdev: {'SBxv3 RX' 'SBxv3 RX' 'SBxv3 RX' 'SBxv3 RX'}
    TXSubdev: {'SBxv3 TX' 'SBxv3 TX' 'SBxv3 TX' 'SBxv3 TX'}
    MinimumCenterFrequency: [3800000000 3800000000 3800000000 3800000000]
    MaximumCenterFrequency: [4.4200e+09 4.4200e+09 4.4200e+09 4.4200e+09]
      MinimumGain: [0 0 0]
      MaximumGain: [37.5000 37.5000 37.5000 37.5000]
        GainStep: [0.5000 0.5000 0.5000 0.5000]
        CenterFrequency: [1.0000e+09 1.1000e+09 1.2000e+09 1.3000e+09]
    LocalOscillatorOffset: 0
      Gain: [5 6 7 8]
      MasterClockRate: 200000000
      DecimationFactor: 512
      BasebandSampleRate: 390625
```

Receive the data. Because the System object uses four channels, the matrix returned in **data** contains four columns.

```
[data,datalen] = rxRadios();
```

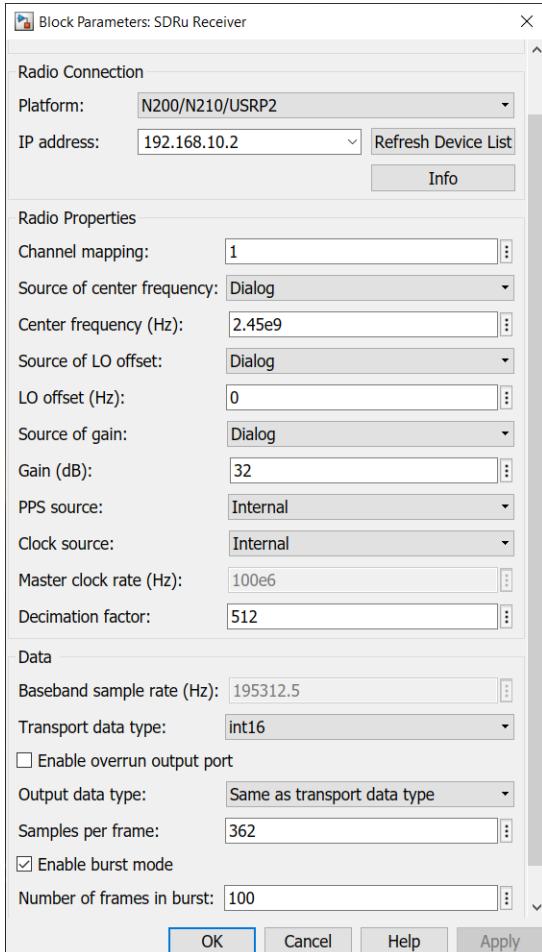
Release the System object.

```
release(rxRadios);
```

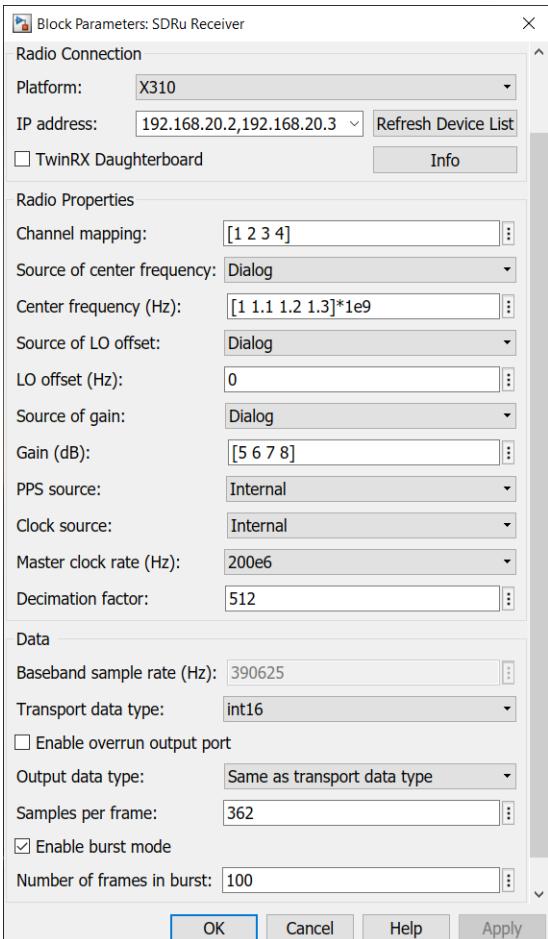
### **Receive from Multiple Radios with SDRu Block**

This example shows how to bundle multiple radios for MIMO operations with an SDRu Receiver block.

- 1 Open the mask of an SDRu Receiver block.



- 2 Set these parameters for reception on multiple X310 radios. Then, click **OK**.
- Set **Platform** to X310.
  - Set the **IP address** to 192.168.20.2, 192.168.20.3 or 192.168.20.2 192.168.20.3 to specify the IP addresses of the X310 radios.
  - Set **Channel mapping** to [1 2 3 4]. Channel mapping values 1 and 2 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.2. Channel mapping values 3 and 4 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.3.
  - Set **Center frequency (Hz)** to [1 1.1 1.2 1.3]\*1e9. Alternatively, to apply the same value to all channels, specify a scalar value.
  - Set **Gain (dB)** to [5 6 7 8]. Alternatively, to apply the same receiver gain value to all channels, specify a scalar value.



The SDRu Receiver block outputs a matrix at its **data** port. The number of columns of the output matrix equals the length of the **Channel mapping** parameter.

## See Also

### More About

- “Single Channel Input and Output Operations” on page 4-8
- “Single and Multiple Channel Output” on page 8-19

## Detect Underruns and Overruns

### In this section...

["Detect Lost Samples Using SDRu Transmitter Block" on page 4-22](#)

["Detect Lost Samples Using SDRu Receiver Block" on page 4-22](#)

["Detect SDRu Transmitter System Object Underruns" on page 4-23](#)

["Detect SDRu Receiver System Object Overruns" on page 4-24](#)

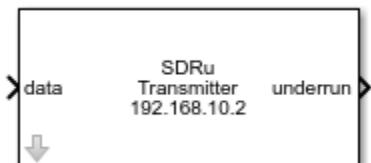
You can detect underruns and overruns using the lost sample indicators of the SDRu blocks and SDRu System object. Use these indicators as a diagnostic tool to determine real-time execution of your designs. If your design is not running in real time, see "Burst-Mode Buffering" on page 5-2.

### Detect Lost Samples Using SDRu Transmitter Block

The SDRu Transmitter block has an optional lost sample port called **underrun**. The lost samples port is disabled by default. To enable it:

- On the SDRu transmitter block, select the **Enable underrun output port** parameter.

To detect underruns during the transmission of radio signals, check the **underrun** output port on the SDRu transmitter block.



During the simulation, to see if any data loss is occurring, check the **underrun** output port of the transmitter.

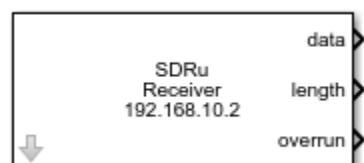
- 0 — Indicates that no data samples were lost
- 1 — Indicates that data samples were lost

### Detect Lost Samples Using SDRu Receiver Block

The SDRu Receiver block has an optional lost sample port called **overrun**. The lost samples port is disabled by default. To enable it:

- On the SDRu receiver block, select the **Enable overrun output port** parameter.

To detect overruns during the reception of radio signals, check the **overrun** output port on the SDRu receiver block.



During the simulation, to see if any data loss is occurring, check the **overrun** output port of the receiver.

- 0 — Indicates that no data samples were lost
- 1 — Indicates that data samples were lost

---

**Note** With burst mode enabled an overrun occurs in between bursts as the streaming resumes, because it is not possible to get continuous data when starting and stopping streaming.

---

## Detect SDRu Transmitter System Object Underruns

When you call the `comm.SDRuTransmitter` System object, the `underrun` output argument indicates discontinuity of the data streaming from the host computer to the USRP radio. If your design is not running in real time, you can adjust properties that reduce the number of transported samples. To approach or achieve real-time performance, you can increase the interpolation factor.

To see if any data loss is occurring, check `underrun` output argument.

- When output is 0 — No underrun detected
- When output is  $\geq 1$  — Underrun detected at transmitter

### Detect Lost Samples using SDRuTransmitter System Object

Configure an B210 radio with serial number set to '`'30F59A1'`'. Set the radio to transmit at 2.5 GHz with an interpolation factor of 125 and master clock rate of 56 MHz.

Create a SDRu Transmitter System object to use for data transmission.

```
tx = comm.SDRuTransmitter(...
    'Platform','B210',...
    'SerialNum','30F59A1',...
    'CenterFrequency',2.5e9,...
    'InterpolationFactor',125,...
    'MasterClockRate', 56e6);
```

Create a DPSK modulator as the data source using `comm.DPSKModulator` System object.

```
modulator = comm.DPSKModulator('BitInput',true);
```

Inside a `for` loop, transmit the data using the `tx` System object and return `underrun` as an output argument. Display the messages when transmitter indicates `underrun` with data loss.

```
for frame = 1:20000
    data = randi([0 1], 30, 1);
    modSignal = modulator(data);
    underrun = tx(modSignal);
    if underrun~=0
        msg = ['Underrun detected in frame # ', int2str(frame)];
    end
end
release(tx)
```

With SDRu transmitter System objects, the `underrun` output indicates data loss. This output is a useful diagnostic tool for determining real-time operation of the System object.

## Detect SDRu Receiver System Object Overruns

When you call the `comm.SDRuReceiver` System object, the `overrun` output argument indicates discontinuity of the data streaming from the USRP radio to the host computer. If your design is not running in real time, you can adjust properties that reduce the number of transported samples. To approach or achieve real-time performance, you can increase the decimation factor.

To see if any data loss is occurring, check `overrun` output argument.

- When output is 0 — No overrun detected
- When output is  $\geq 1$  — Overrun detected at receiver

---

**Note** With burst mode enabled an overrun occurs in between bursts as the streaming resumes, because it is not possible to get continuous data when starting and stopping streaming.

---

## Detect Lost Samples using SDRu Receiver System Object

Configure a B210 radio with serial number set to '31B92DD'. Set the radio to receive at 2.5 GHz with an decimation factor of 125, output data type as 'double' and master clock rate of 56 MHz.

Create a USRP radio receiver System object to use for data reception.

```
rx = comm.SDRuReceiver('Platform','B210', ...
    'SerialNum','31B92DD', ...
    'CenterFrequency',2.5e9, ...
    'MasterClockRate',56e6, ...
    'DecimationFactor',125, ...
    'OutputDataType','double');
```

Capture signal data using `comm.DPSKDemodulator` System object.

```
demodulator = comm.DPSKDemodulator('BitOutput',true);
```

Inside a `for` loop, receive the data using the `rx` System object and return `overrun` as an output argument. Display the messages when receiver indicates `overrun` with data loss.

```
for frame = 1:2000
    [data, len, overrun] = rx();
    demodulator(data);
    if len>0
        if overrun~=0
            msg = ['Overrun detected in frame # ', int2str(frame)];
        end
    end
end
release(rx)
```

With SDRu receiver System objects, the `overrun` output indicates data loss. This output is a useful diagnostic tool for determining real-time operation of the System object.

## See Also

### More About

- “Burst-Mode Buffering” on page 5-2

## Data Frame Lengths

### In this section...

["Set Frame Length in SDRu Receiver Block" on page 4-26](#)

["Set Frame Length in SDRu Receiver System Object" on page 4-26](#)

You can set the frame length of the SDRu Receiver block or `comm.SDRuReceiver` System object to be any integer value. This flexibility enables more straightforward multirate operation and easier modeling of standards-based packet communications. The default value is 362. This value optimally utilizes the underlying Ethernet payloads for a standard 1500-byte MTU.

### Set Frame Length in SDRu Receiver Block

In the block mask of the receiver block, set **Samples per frame** parameter to the desired frame length.

### Set Frame Length in SDRu Receiver System Object

- 1 Create a receiver System object. For example:

```
radio = comm.SDRuReceiver('Platform','B200','SerialNum','30FD838')
```

- 2 Set the **SamplesPerFrame** property to the desired frame length.

```
radio.SamplesPerFrame=1024
```

```
radio =
```

```
System: comm.SDRuReceiver
```

```
Properties:
```

```
    Platform: 'B200'
    SerialNum: '30FD838'
    ChannelMapping: 1
    CenterFrequency: 2.4500e+09
    LocalOscillatorOffset: 0
    Gain: 8
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 32000000
    DecimationFactor: 512
    TransportDataType: 'int16'
    OutputDataType: 'Same as transport data type'
    SamplesPerFrame: 1024
    EnableBurstMode: false
```

# Apply Conditional Execution

## In this section...

["Using Conditional Execution with SDRu Receiver Block" on page 4-27](#)

["Using Conditional Execution with SDRu Receiver System Object" on page 4-27](#)

The SDRu receiver block and System object components have a length output that can be used to trigger conditional execution of components downstream from them.

## Using Conditional Execution with SDRu Receiver Block

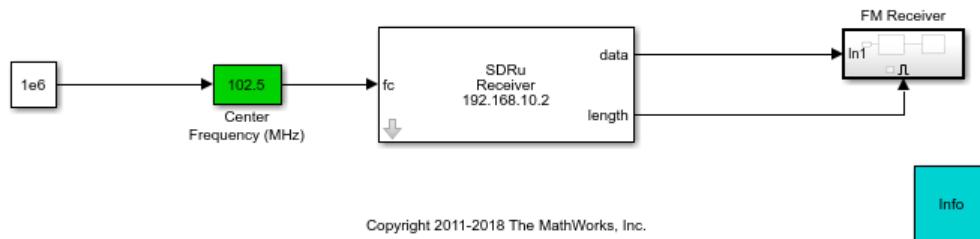
The SDRu Receiver block has a static length port that outputs either 0 or the frame length specified in the block mask. When the output is 0, Simulink is running faster than the USRP hardware, and the hardware did not have any new data to send to Simulink at that instant of sample time. When the output is the specified frame length, then the hardware sent valid data.

Any processing downstream of the SDRu Receiver block needs to run only when there is valid data to process, so control it with an enabled subsystem. The length output serves as a convenient control signal for such an enabled subsystem.

To get more information on sample time, see "What Is Sample Time?" (Simulink).

### Example

The SDRu Receiver block has a length port for validating the presence of data.



## Using Conditional Execution with SDRu Receiver System Object

The `comm.SDRuReceiver` System object uses a data length output when the object is called as a function to indicate the length of data packets streaming to MATLAB from the USRP hardware. If the length is 0, then the hardware has not sent any valid data. If the length equals the specified `FrameLength`, then it has sent valid data. Any downstream processing must be conditioned on the presence of valid data.

---

**Note** Set up your USRP hardware and make sure that the host PC is communicating with the radio before you run this example.

---

### Example

```
radio = comm.SDRuReceiver('Platform','N200/N210/USRP2', ...
    'IPAddress','192.168.10.2','CenterFrequency',102.5e6, ...
```

```
'Gain',30,'DecimationFactor',500, ...
'SamplesPerFrame',4000,'OutputDataType','single');

% USRP N200/N210/USRP2 master clock rate is 100 MHz
sampleRate = 100e6/500;
frameTime = 4000/sampleRate;

radioList = findsdru('192.168.10.2');

if strcmp(radioList.Status,'Success')
    % Loop until the example reaches 10 seconds
    timeCounter = 0;
    while timeCounter < 10
        [data, len] = radio();
        if len > 0
            % Received valid data, so run downstream algorithm

            % <insert your algorithm and code to process>

            % Update counter
            timeCounter = timeCounter + frameTime;
        end
    end
end
```

# Change Transport Data Rate

## In this section...

["Set Transport Data Type in Simulink" on page 4-29](#)

["Set Transport Data Type in MATLAB" on page 4-29](#)

The USRP radio samples the signals received at the antenna. The radio then transports the samples to the host computer. By default, the transport data rate uses 16 bits for the in-phase component and 16 bits for the quadrature component, making each sample require 32 bits to transport. You can increase the transfer speed by reducing the signal precision to 8 bits. Using the `int8` setting for transport data type makes the transfer speed about 2x faster, and helps you avoid overruns and burst mode failures.

**Note** When you use 8-bit transport, make sure that your gain setting is sufficiently high for the signal to use the entire 8-bit range. This setting results in a quantization step that is 256 times bigger.

## Set Transport Data Type in Simulink

The SDRu Receiver and SDRu Transmitter blocks each contain the parameter **Transport data type** for setting the transport type. The default transport data type is `int16`.



### Set 16-Bit Transport Data Type in SDR Blocks



### Set 8-Bit Transport Data Type in SDR Blocks

## Set Transport Data Type in MATLAB

The `comm.SDRuReceiver` and `comm.SDRuTransmitter` System objects each contain the `TransportDataType` property for setting the transport type. Set `TransportDataType` to `int8` for 8-bit transport, or to `int16` for 16-bit transport. The default transport data type is `int16`.

This code example demonstrates how the quantization step size is increased when you use 8-bit transport. Enter this code in the MATLAB command window and observe the differences in the plots.

**Note** Before you run this example, set up your USRP hardware and make sure that the host PC is communicating with the radio.

- 1 Create radio object. This example uses a B210 radio with the serial number 'F5BA6A'. When you run this code, configure the System object for the type of radio that you have, and include the appropriate IP address or serial number. If necessary, adjust the center frequency for the best results.

```
radio = comm.SDRuReceiver('Platform','B210','SerialNum','F5BA6A');
2 Set radio properties.
```

```
radio.CenterFrequency = 102.5e6;
radio.Gain = 65;
radio.MasterClockRate = 20e6;
radio.DecimationFactor = 100;
radio.FrameLength = 4000;
radio.OutputDataType = 'double';
radio.EnableBurstMode = true;
radio.NumFramesInBurst= 20;
```

**3** Use 16-bit transport.

```
radio.TransportDataType = 'int16';
data1 = zeros(4000,20);
for i = 1:20
    len = 0;
    while len <= 0
        [data1(:,i), len] = radio();
    end
end
```

**4** Plot received signals and release radio.

```
figure
plot(data1(:,1))
title('Received Signal (TransportDataType = ''int16'')')
xlabel('In-phase Component');
ylabel('Quadrature Component');
axis([-0.04 0.04 -0.04 0.04]);
```

```
release(radio)
```

**5** Use 8-bit transport.

```
radio.TransportDataType = 'int8';

data2 = zeros(4000,20);
for i = 1:20
    len = 0;
    while len <= 0
        [data2(:,i), len] = radio();
    end
end
```

**6** Plot received signals.

```
figure
plot(data2(:,1))
title('Received Signal (TransportDataType = ''int8'')')
xlabel('In-phase Component');
ylabel('Quadrature Component');
axis([-0.04 0.04 -0.04 0.04]);
```

**7** Compare quantization step sizes.

```
r = real(data1(:));
qstep1 = min(r(r>0))

qstep1 =
3.0519e-05
```

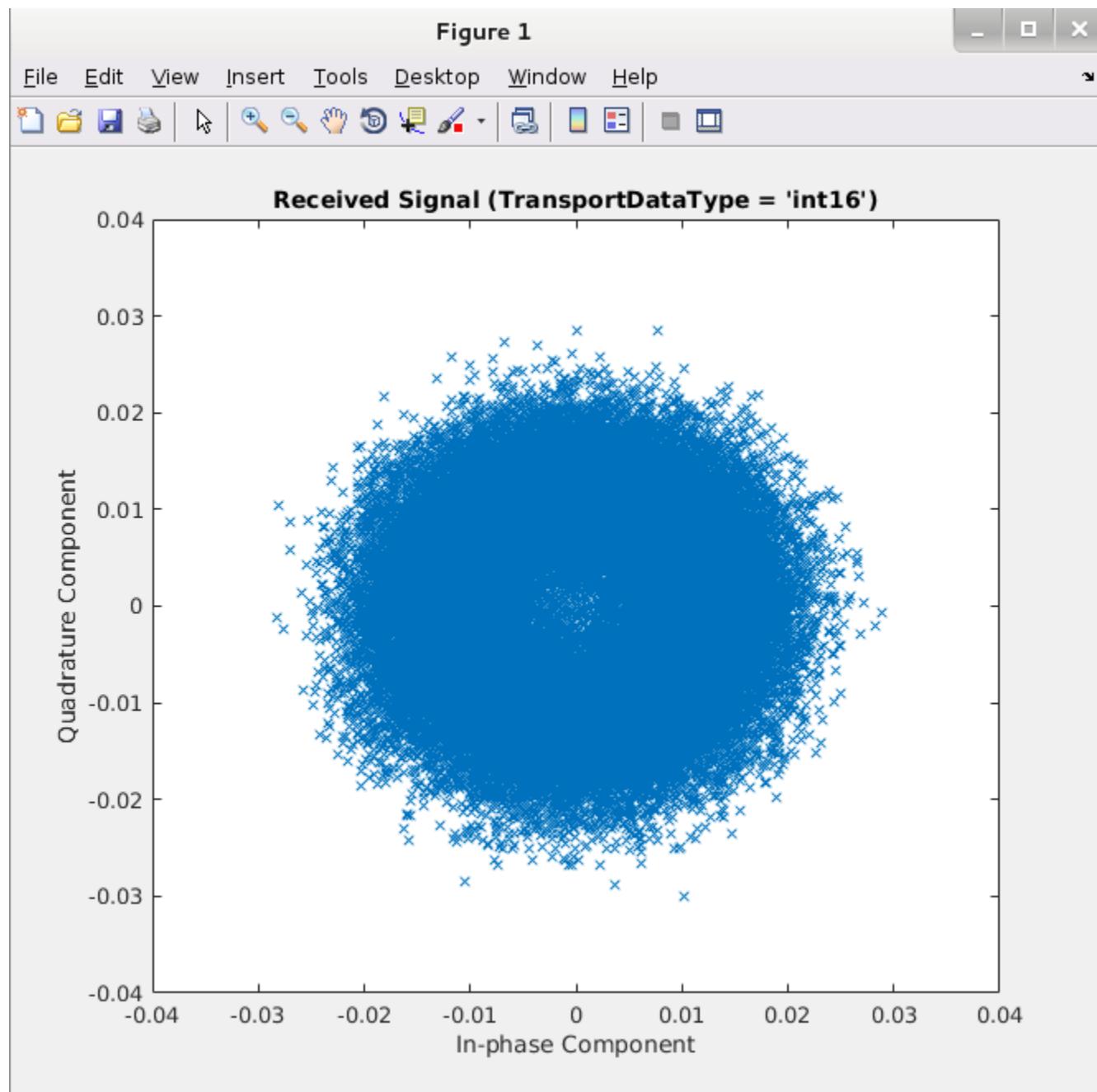
```
r = real(data2(:));
qstep2 = min(r(r>0))

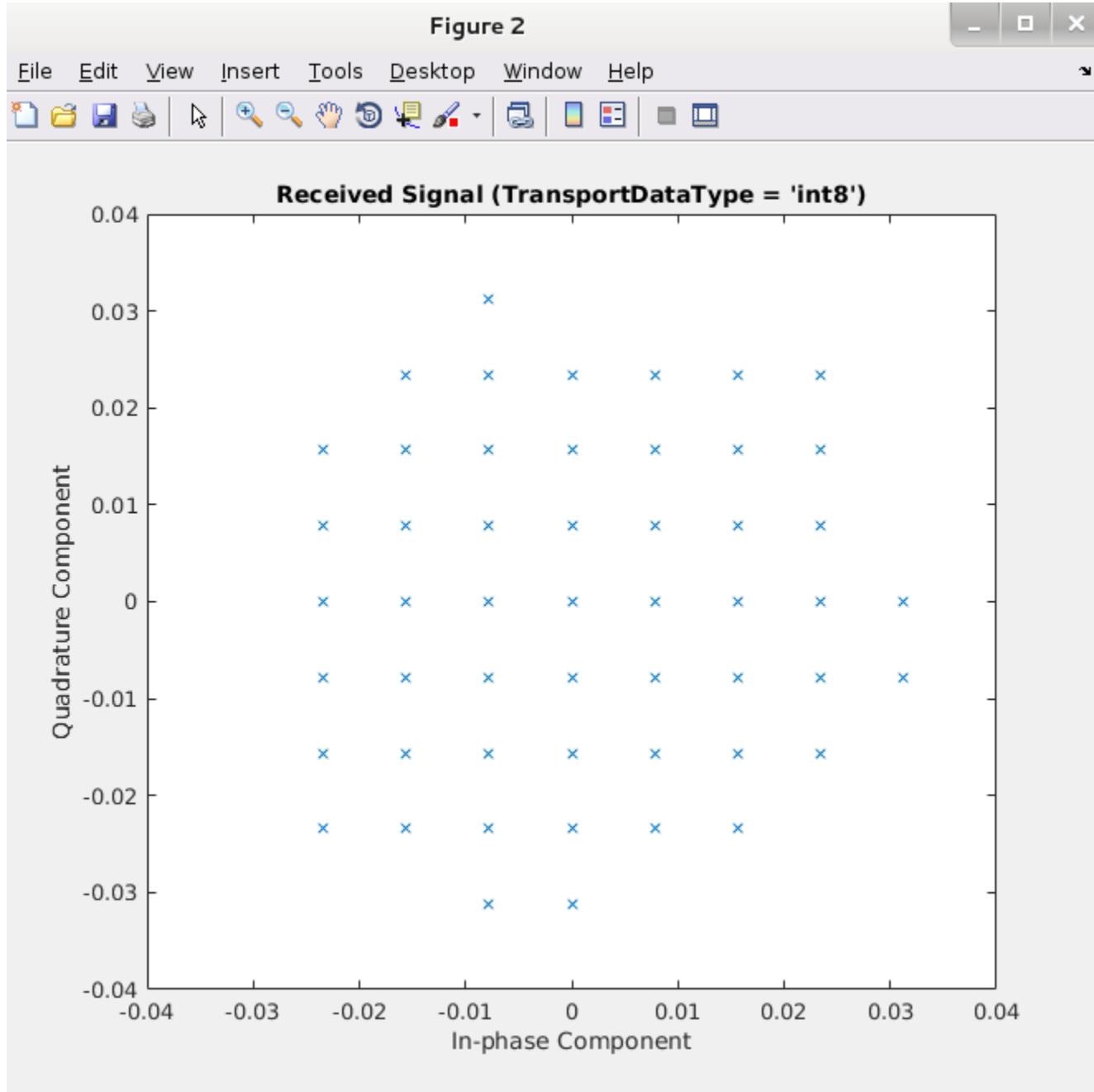
qstep2 =
0.0078

qstep2/qstep1

ans =
256
```

The quantization step size is 256 times larger when 8-bit transport is used.





**Quantization Step Plot Using 8-Bit Transport**

## See Also

### Related Examples

- “Detect Underruns and Overruns” on page 4-22
- “Burst-Mode Buffering” on page 5-2

## Transmit and Receive Using External Clock

### In this section...

["Specify External Clock in SDRu Blocks" on page 4-34](#)

["Specify External Clock in SDRu System Objects" on page 4-34](#)

An external clock generator can provide a more accurate and stable clock signal than the internal clock in the radio. When multiple radios are connected to the same external clock generator, their RF frequencies are synchronized. You can use the external clock with both a transmitter and receiver, or just transmitters, or just receivers. When you use external clock support for a transmitter and a receiver, you eliminate the need for carrier frequency offset compensation in the receiver.

For B-series radios, the external clock port is labeled "10 MHz." For N-series, USRP2, and X-series radios, the external clock port is labeled "REF IN." Connect the external clock to the radio for this feature to work.

### Specify External Clock in SDRu Blocks

To indicate that you want the radio to use an external clock for simulation, set **Clock source** to External.



If you specify an external clock and the external clock is not connected to the radio when you run the simulation, you get this message: "Unable to detect external clock signal."

### Specify External Clock in SDRu System Objects

To specify the clock source in the receiver and transmitter System objects, use the `ClockSource` property. To use an external clock for SDR applications, set `ClockSource` to 'External'. The default value for `ClockSource` is 'Internal'.

```
radio = comm.SDRuReceiver('ClockSource','External')
```

`comm.SDRuReceiver` with properties:

```

    Platform: 'N200/N210/USRP2'
    IPAddress: '192.168.10.2'
    CenterFrequencySource: 'Property'
        CenterFrequency: 2.4500e+09
        ActualCenterFrequency: 0
    Local0scillatorOffsetSource: 'Property'
        Local0scillatorOffset: 0
    ActualLocal0scillatorOffset: 0
        GainSource: 'Property'
            Gain: 8
            ActualGain: 0
        ClockSource: 'External'
    DecimationFactorSource: 'Property'
        DecimationFactor: 512
    ActualDecimationFactor: 0

```

```

TransportDataType: int16
    SampleRate: 1
    OutputDataType: 'Same as transport data type'
    FrameLength: 362
EnableBurstMode: 0

```

If you specify an external clock and the external clock is not connected to the radio when you run the simulation, you get this message: "Unable to detect external clock signal."

## Synchronize Receiver and Transmitter with External Clock Signal

This example shows you how to correctly synchronize two radios with an external clock signal. This example uses one N-series radio and one B-series radio. When you run the code, make sure that you configure the System objects with your specific radio types.

- 1** Connect the two radios to a common clock generator.
- 2** Start MATLAB and run `sdrussetup`.
- 3** Start a second session of MATLAB and run `sdrussetup`.
- 4** In the first session of MATLAB, run the following transmitter code (remember to substitute your radio types for the ones used in this example):

```

basebandFs = 400e3;
fc = 900e6;
radio = comm.SDRuTransmitter('Platform', 'N200/N210/USRP2', ...
    'IPAddress', '192.168.10.2', ...
    'InterpolationFactor', 100e6 / basebandFs, ...
    'Gain', 15);

% Add a frequency offset of 900 Hz
% Carrier synchronizer in the receiver will estimate this value
radio.CenterFrequency = fc + 900; % Relative error is 1 ppm

% Connect the output of a 10 MHz clock generator to the REF IN port of the radio
radio.ClockSource = 'External';

txFilter = comm.RaisedCosineTransmitFilter('RolloffFactor', 0.5, ...
    'OutputSamplesPerSymbol', 4);

```

```

while true
    qpskSymbols = pskmod(randi([0 3], 2048, 1), 4, pi/4);
    samples = txFilter(qpskSymbols);
    radio(samples); % Send a randomly generated QPSK signal
end

```

- 5** In the second session of MATLAB, run the following receiver code:

```

% Run the following in a separate MATLAB session
basebandFs = 400e3;
fc = 900e6;
frameLen = 4000;

% Connect the output of the same clock generator driving the transmitter
% to the 10 MHz port of the B210 radio
radio = comm.SDRuReceiver('Platform', 'B210', ...
    'SerialNum', 'F5BA6A', ...
    'MasterClockRate', 8e6, ...
    'DecimationFactor', 8e6 / basebandFs, ...
    'Gain', 35, ...
    'OutputDataType', 'double', ...
    'FrameLength', frameLen, ...
    'EnableBurstMode', true, ...
    'NumFramesInBurst', 20, ...
    'CenterFrequency', fc, ...
    'ClockSource', 'External'); % Use external clock

rcosDecim = 4;
rxFilter = comm.RaisedCosineReceiveFilter('RolloffFactor', 0.5, ...
    'InputSamplesPerSymbol', 4, ...
    'DecimationFactor', rcosDecim);

carrSync = comm.CarrierSynchronizer('Modulation', 'QPSK', ...
    'SamplesPerSymbol', 1, ...
    'DampingFactor', 0.707, ...

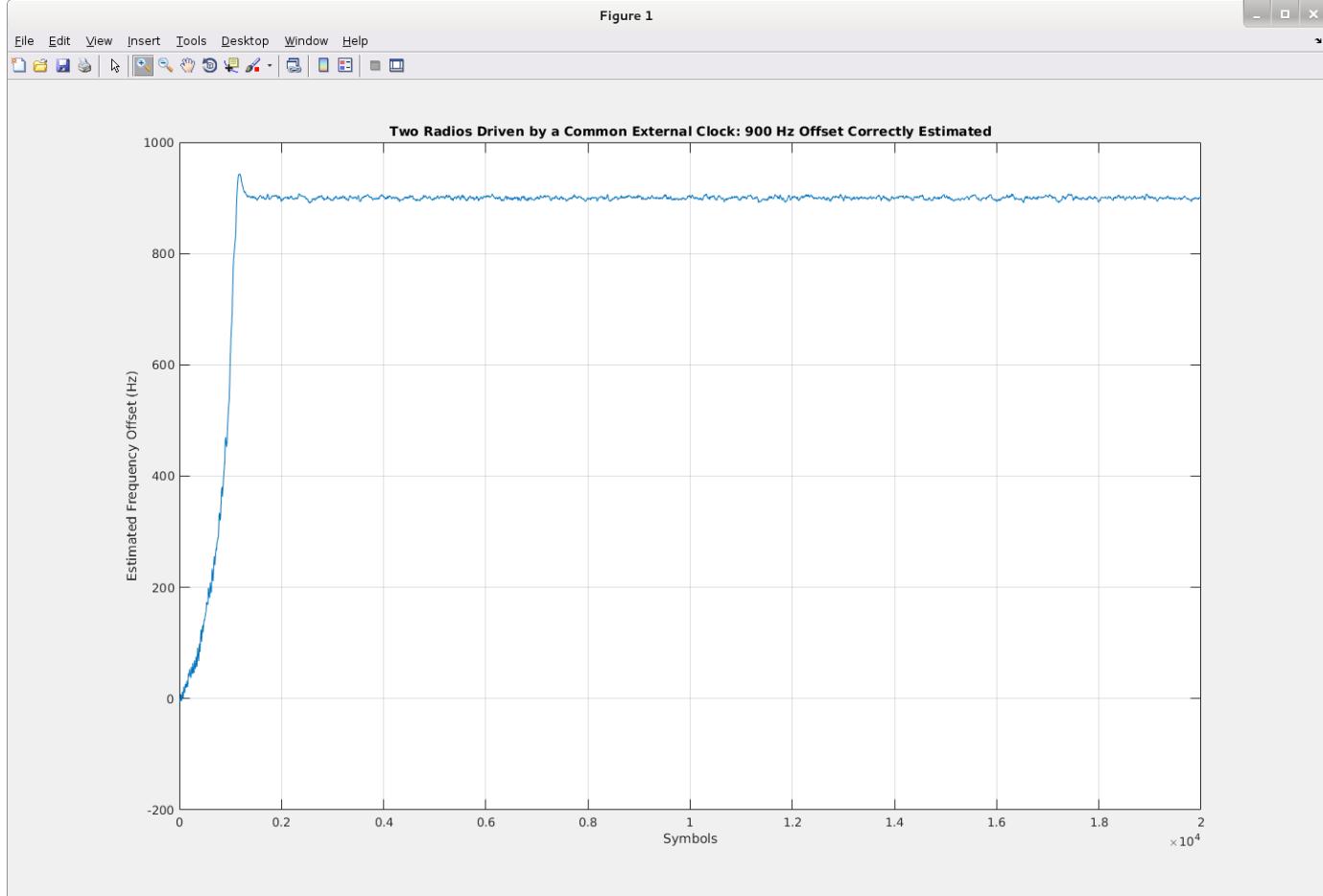
```

```
'NormalizedLoopBandwidth', 0.01);

phaseEstimate = zeros(frameLen/racosDecim, 20);
for i = 1:20;
    [~, phaseEstimate(:,i)] = carrSync(rxFilter((radio())));
end

rad2Hz = (basebandFs/racosDecim)/(2*pi);
freqEstimate = filter(ones(200,1)/200, 1, ... % Moving average
                      diff(unwrap(phaseEstimate(:)))*rad2Hz);
plot(freqEstimate)
grid on
 xlabel('Symbols')
 ylabel('Estimated Frequency Offset (Hz)')
 title('Two Radios Driven by a Common External Clock: 900 Hz Offset Correctly Estimated');
```

- 6 In the second session, you should see a graph similar to the following:



You can see that the estimated frequency offset is very close to the artificial offset of 900 Hz. This graph shows that the RF frequencies of the radios are correctly synchronized with the external clock signal.

### See Also

[SDRu Receiver](#) | [SDRu Transmitter](#) | [comm.SDRuReceiver](#) | [comm.SDRuTransmitter](#)

# Desired Vs. Actual Parameter Values

## In this section...

["Desired vs. Actual in the SDRu Blocks" on page 4-37](#)

["Desired vs. Actual in the SDRu System Objects" on page 4-38](#)

## Desired vs. Actual in the SDRu Blocks

When you set mask values for center frequency, gain, interpolation factor, and decimation factor, the block performs rudimentary checks that the values are scalar and real. Even if your values pass those checks, you can still provide values that are out of range for the USRP hardware. In that case, the hardware makes a best effort to set the requested value, and reports the actual value in the block mask.

If the SDRu block is connected to USRP hardware, you can check the block mask to find the acceptable ranges of center frequency and gain. The acceptable ranges of interpolation and decimation are integer values as follows:

Decimation factor for the SDRu receiver, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

Decimation Factor or Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
1	Valid	Not valid	Valid	Not valid when connected with TwinRX daughterboard
2	Valid	Acceptable when you use only the <code>int8</code> transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Valid

The radio uses the decimation factor when it downconverts the intermediate frequency (IF) signal to a complex baseband signal.

Interpolation factor for the SDRu transmitter, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

InterpolationFactor Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
1	Valid	Not valid	Valid	Valid
2	Valid	Acceptable when you use only the <code>int8</code> transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Not valid

The radio uses the interpolation factor when it upconverts the complex baseband signal to an intermediate frequency (IF) signal.

## Desired vs. Actual in the SDRu System Objects

The default parameter values for the SDRu System objects are as follows:

- `IPAddress`: 192.168.10.2
- `CenterFrequencySource`: Property
- `CenterFrequency`: 2450000000
- `ActualCenterFrequency`: 0
- `LocalOscillatorOffsetSource`: Property
- `LocalOscillatorOffset`: 0
- `ActualLocalOscillatorOffset`: 0
- `GainSource`: Property
- `Gain`: 8
- `ActualGain`: 0
- `InterpolationFactorSource`: Property
- `InterpolationFactor`: 512
- `ActualInterpolationFactor`: 0
- `UnderrunOutputPort`: False

- EnableBurstMode: False

## Supported Data Types

<b>Port</b>	<b>Supported Data Types</b>
Input	<ul style="list-style-type: none"><li>• Double-precision floating point</li><li>• Single-precision floating point</li><li>• 16-bit signed integers</li></ul>

# Performance Optimization

---

- “Burst-Mode Buffering” on page 5-2
- “Model Performance Optimization” on page 5-4
- “MATLAB Performance Improvements” on page 5-6
- “Computing Environment Optimization” on page 5-7

## Burst-Mode Buffering

### In this section...

- “What is Burst Mode?” on page 5-2
- “Determining If You Need Burst Mode” on page 5-2
- “Using Burst Mode” on page 5-2

### What is Burst Mode?

If the SDRu Receiver block or System object is not keeping up with the hardware, you are not processing data in real time. Burst mode allows you to buffer a minimum set of contiguous samples without losing samples.

You must first determine if you need burst mode.

### Determining If You Need Burst Mode

To determine if you should enable burst mode,

- For blocks monitor:
  - The `underrun` port when using SDRu Transmitter
  - The `overrun` port when using SDRu Receiver
- For System objects monitor:
  - The `underrun` output when you call `comm.SDRuTransmitter`
  - The `overrun` output when you call `comm.SDRuReceiver`

For more information, see “Detect Underruns and Overruns” on page 4-22.

If your model is not running in real time, you can:

- Use Burst mode
- Use vector-based processing
- Accelerate with code generation

Any combination of these techniques may be applied to approach or achieve real-time performance.

### Using Burst Mode

Use burst mode when your model is experiencing underruns or overruns because it can't keep up with the amount of data being sent or received in real time. Burst mode allows you to buffer a minimum set of contiguous samples without underruns or overruns.

---

**Tip** Overruns and underruns can still happen between bursts, especially with large burst sizes. Therefore, enabling the burst mode feature is recommended only if your model cannot keep up in real time.

---

The maximum burst size (in frames) is imposed by the operating system and the USRP device UHD. The maximum size imposed by the UHD is approximately 1 GB, or 256 megasamples. This maximum number of samples is enforced by our software. For example, with a frame size of 4000 samples, the maximum burst is approximately 67k frames. Depending on the memory constraints on a specific host, a lower limit may required. Exceeding the limit will be flagged by an 'unable to allocate memory' error.

The default burst size is 100 frames.

### Receiver Burst Mode Processing

When using burst mode for reception, the first SDRu receiver object call transfers a whole burst to the host computer memory and the SDRu receiver object processes the first frame. Subsequent SDRu receiver object calls process the rest of the burst, one frame at a time, from the host computer memory (not from the radio). When all the frames in the transferred data have been processed, the next SDRu receiver object call transfers another whole burst to the host computer memory and the first frame of data is processed by the SDRu receiver object. For example with `EnableBurstMode` set to `true`, if `NumFramesInBurst` is 10, and `SamplesPerFrame` is 375000. For the first SDRu receiver object call, a whole burst ( $375,000 \text{ samples/frame} * 10 \text{ frames/burst} = 3,750,000 \text{ samples/burst}$ ) is transferred to the host computer memory and the SDRu receiver object processes the first frame of data. For subsequent SDRu receiver object calls (second through tenth calls), one frame at a time, is processed from the host computer memory. After the whole burst has been processed, on the eleventh SDRu receiver object call, another whole burst is transferred from the radio to the host computer memory and the first frame of data is processed by the SDRu receiver object.

### Transmitter Burst Mode Processing

When using burst mode for transmission, data is not transferred to the radio until the SDRu transmitter object has been called `NumFramesInBurst` times. After the SDRu transmitter object has been called `NumFramesInBurst` times, the whole burst is transferred to the radio and transmitted.

- To set burst mode on the SDRu blocks, see SDRu Receiver and SDRu Transmitter.
- To set burst mode on the SDRu System objects, see `comm.SDRuReceiver` and `comm.SDRuTransmitter`.

## See Also

### Related Examples

- “Change Transport Data Rate” on page 4-29
- “Common Problems and Fixes” on page 2-2

# Model Performance Optimization

## In this section...

- “Acceleration” on page 5-4
- “Model Tuning” on page 5-4
- “Simulink Code Generation” on page 5-4

## Acceleration

Run your application with Accelerator or Rapid Accelerator instead of Normal mode. Be aware that some scopes do not plot data when run in Rapid Accelerator mode.

When using accelerator or rapid accelerator mode, on the **Model Configuration Parameters** dialog box, search for **Compiler optimization level**, then set the parameter value to **Optimizations on (faster runs)**.

## Model Tuning

- Use frame-based processing. With frame-based processing, the model processes multiple samples during a single execution call to a block. Consider using frame sizes from roughly 100 to several thousand.
- In **Model Configuration Parameters > Data Import/Export**, turn off all logging.
- The model should be single-rate. If the model requires resampling, then choose rational coefficients that will keep the model single-rate.
- Do not add any Buffer blocks to the model. If you want to create convenient frame sizes, do it in your data sources. Using a Buffer block typically degrades performance.
- Avoid feedback loops. Typically, such loops imply scalar processing, which will slow down the model considerably.
- Avoid using scopes. To visualize your data, send it to a workspace variable and post-process it.
- If your model has Constant blocks with values that do not change during simulation, make sure that the sample time is set to **inf** (default).

## Simulink Code Generation

- If you are generating code from the model, set the Solver setting to Fixed-step/ discrete. Set tasking mode to SingleTasking.
- You can generate a standalone executable for your Simulink model to improve performance. The generated code runs without Simulink in the loop. To perform any code generation, you must have an appropriate compiler installed. See <https://www.mathworks.com/support/compilers/> for a list of supported compilers.

You can generate generic real-time target (GRT) code if you have a Simulink Coder™ license. To do so, set **Model Configuration Parameters > Code Generation > System target file** to **grt.tlc** (Generic Real-Time Target).

- When you generate code for any target (not just GRT), uncheck the **Model Configuration Parameters > Hardware Implementation > Test hardware > Test hardware is the same as**

**production hardware** checkbox. After the checkbox is unchecked, set the Device type popup to MATLAB Host Computer.

- You can create generated code with a smaller stack than the GRT code if you have an Embedded Coder® license. To do so, set the **Model Configuration Parameters > Code Generation > System target file** to ert.tlc (Embedded Coder). Then, add the following lines to the **Model Configuration Parameters > Code Generation > Custom Code > Include custom C code in generated: > Source file** edit field:

```
#include <stdio.h>
#include <stdlib.h>
```

## See Also

## Related Examples

- “Common Problems and Fixes” on page 2-2

# MATLAB Performance Improvements

## In this section...

- “Vector-Based Processing” on page 5-6
- “MATLAB Code Generation” on page 5-6

## Vector-Based Processing

- Use vector-based processing. With vector-based processing, the program processes multiple samples during a single execution call to a System object. Consider using vectors from roughly 366 to several thousand. The default is 3660, which represents 10 Ethernet packets.
- Use large vectors of data to minimize function call overhead.

## MATLAB Code Generation

You can accelerate your MATLAB algorithms by generating a MEX function using the MATLAB Coder function `codegen`.

Use `codegen` to generate a MEX function from MATLAB code. The following example generates a MEX file called `sdruExMex` from the function `sdruExample`:

```
codegen sdruExample -args {ones(10,1)} -o sdruExMex -g -launchreport
```

---

**Caution** When you use this function, you must release the radio object before attempting to exit MATLAB, or MATLAB stops responding.

---

For a full list of syntax options and input parameters, see the `codegen` reference page.

---

**Note** `sdruExample` is used only for illustrative purposes; it is not a function shipped with the support package for USRP Radio. You must provide your own function for `codegen`.

---

## See Also

## Related Examples

- “Common Problems and Fixes” on page 2-2

# Computing Environment Optimization

Tune your computing environment to improve performance:

- Turn off antivirus and firewall software.
- Turn off all nonessential background processes on your computer.
- Disable all but IP4 in your network stack.

The following suggestions may also improve performance or reduce data loss:

- If you have a B-series radio, make sure that you use a USB 3 connection.
- See the recommended USB 3.0 controllers from Ettus Research: [https://kb.ettus.com/About\\_USRP\\_Bandwidths\\_and\\_Sampling\\_Rates](https://kb.ettus.com/About_USRP_Bandwidths_and_Sampling_Rates).
- With laptops, try connecting the laptop to a power supply. Most laptops are configured for better battery life when they are not connected to a power supply. This affects the performance negatively. For both battery mode and AC power supply mode, make sure you have a power setting corresponding to maximum performance. Some laptop manufacturers might also provide advanced power settings to help choose a plan for maximum CPU performance.

## See Also

### Related Examples

- “Common Problems and Fixes” on page 2-2



# FPGA Targeting

---

- “FPGA Targeting Overview” on page 6-2
- “Target FPGA with Custom Bitstream” on page 6-3

# FPGA Targeting Overview

## About FPGA Targeting

The FPGA Targeting workflow allows you to prototype baseband algorithms with Simulink on an FPGA with USRP N210 hardware, speeding up your simulations. FPGA Targeting can be used for over-the-air interface using the daughter card that corresponds to the N210 board. You can find a list of supported daughter cards in the Release Notes.

The software performs the following actions:

- Generates HDL code (with HDL Coder software).
- Carries out the necessary modifications to connect the generated HDL module with the output of digital down converter as implemented in the original USRP FPGA design. Therefore, the output from the FPGA to Simulink is now the output of the your algorithm instead of the original digital down converter output.

This workflow only applies to single-channel receivers that use the digital downconverter (DDC) already present in the FPGA within the USRP N210.

- Removes the logic from the transmit path and the second receive path from the original USRP FPGA design. Removing the logic creates more FPGA area for implementation of baseband algorithms from Simulink.

# Target FPGA with Custom Bitstream

## In this section...

- “Create Algorithm” on page 6-3
- “Set Tool Path” on page 6-4
- “Create FPGA Files with HDL Workflow Advisor” on page 6-4
- “Burn Custom FPGA File” on page 6-5
- “Verify FPGA Implementation” on page 6-5

## Create Algorithm

- “Design Guidelines” on page 6-3
- “Output Signals” on page 6-3
- “Scalar Mode” on page 6-3

### Design Guidelines

Create FPGA algorithm in Simulink, following recommended guidelines and limitations. As you develop your new FPGA algorithm in Simulink, consider the requirements for this workflow.

- You can use this workflow only when you configure the USRP hardware as a single channel receiver only.
- Output from your targeted subsystem must consist of two 16-bit signed signals to match the output of the existing digital down converter in the FPGA.
- Multiple input ports must all operate at the same rate as each other. Multiple output ports must also all operate at the same rate as each other.
- Use scalar mode for HDL Code generation.

The following sections offer suggestions for working within these requirements:

- “Output Signals” on page 6-3
- “Scalar Mode” on page 6-3

### Output Signals

Output from your targeted subsystem must consist of two 16-bit signed signals to match the output of the existing digital down converter (DDC) in the FPGA. If you need to convert signals, for example, you might create a subsystem that converts the output to 16-bit complex integers.

### Scalar Mode

Your system should operate in scalar mode for HDL Code generation. You can convert frame-based input signals to scalar by using the Unbuffer block. You can then convert the output back to frame signals using the Buffer block. Within this boundary of Unbuffer and Buffer blocks, the system operates in scalar mode, which is necessary for HDL Code generation.

## Set Tool Path

Set up your system environment for to access Xilinx® ISE from MATLAB with the function `hdlsetuptoolpath`. This function adds the necessary folders to the MATLAB search path using the Xilinx installation folder as its argument. For example:

```
sdrsetup('ToolName', 'Xilinx ISE', 'ToolPath', 'C:\Xilinx\14.6\ISE_DS\ISE\bin\nt64\ise.exe');  
Found and setup Communications Toolbox Support Package for USRP Radio.
```

This example assumes that the Xilinx ISE design suite is installed at C:\Xilinx\13.1\ISE\_DS\ISE.

## Create FPGA Files with HDL Workflow Advisor

- 1 Select the subsystem for HDL code generation. This is the subsystem that models the algorithm.
- 2 Launch HDL Workflow Advisor by selecting **Code > HDL Code > HDL Workflow Advisor**.
- 3 **Set Target** (HDL Workflow Advisor step 1).

At step 1.1, Set Target Device and Synthesis Tool, select the following:

- For **Target workflow**: Customization for the USRP Device
- For **Target platform**, select one of the following:
  - USRP N210 Board Rev 4.0
  - USRP N210 Board Rev 2.0

- 4 **Prepare Model for HDL Code Generation** (HDL Workflow Advisor step 2).

Run steps 2.1 through 2.5 to make sure your model is compatible with FPGA targeting.

---

**Note** At step 2.5, Check USRP Compatibility, the software checks that the algorithm does not violate any limitations; see “Create Algorithm” on page 6-3.

---

- 5 **HDL Code Generation** (HDL Workflow Advisor step 3).
  - a At step 3.1.2, Advanced Options, note the following settings but do not change them — they have been pre-set based on the FPGA Targeting workflow:
    - **Reset type**: Synchronous
    - **Reset asserted level**: Active-high
    - **Clock inputs**: Single
    - **Oversampling**: 1Setting these HDL options generates code that is compatible with the original Ettus Research™ USRP FPGA code.
  - b At step 3.2, Generate RTL Code and Testbench, select **Generate RTL Code**.
- 6 **Generate FPGA Implementation** (HDL Workflow Advisor step 4).

The USRP Source File Folder is already populated with the path to the USRP FPGA root directory. The relevant USRP FPGA code was downloaded during the installation from Ettus Research™ UHD Mirror site.
- 7 Click **Run This Task**.

The HDL Workflow Advisor creates a new Xilinx ISE project and adds the following:

- All the necessary files from the FPGA repository
- The generated HDL files for the selected subsystem and algorithm

If no errors are found during FPGA project generation and syntax checking, the FPGA programming file generation process starts. You can view this process in an external command shell and monitor its progress. When the process is finished, a message in the command window prompts you to close the window.

For additional instructions on using the HDL Workflow Advisor, see the HDL Coder documentation.

## Burn Custom FPGA File

You can burn the generated FPGA binary using `sdruload`, specifying the generated FPGA binary `usrp_n210_r4_fpga_mw.bin` (or `usrp_n210_r2_fpga_mw.bin`) from a project location such as `hdl_prj\usrp_prj`. For the firmware image, use the original firmware (`usrp_n210_fw_bin`) as no changes are made to firmware during FPGA targeting.

## Verify FPGA Implementation

To verify your FPGA implementation, first create a new targeted model based on your original model.

- 1 Copy your original model to a new model. This new model will be the retargeted model.
- 2 Remove the top-level subsystem that you specified during targeting. This subsystem is now programmed onto the FPGA.
- 3 If you earlier created a subsystem to create the two 16-bit signed signals required by the FPGA DDC, now put in place a method to reconvert your signals to match the expected output for your model (for example, a subsystem similar to the one you designed to create the signals).
- 4 Run simulation. The model now produces real-time data from your algorithm output.



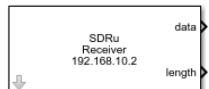
# Blocks

---

# SDRu Receiver

Receive data from USRP device

**Library:** Communications Toolbox Support Package for USRP Radio

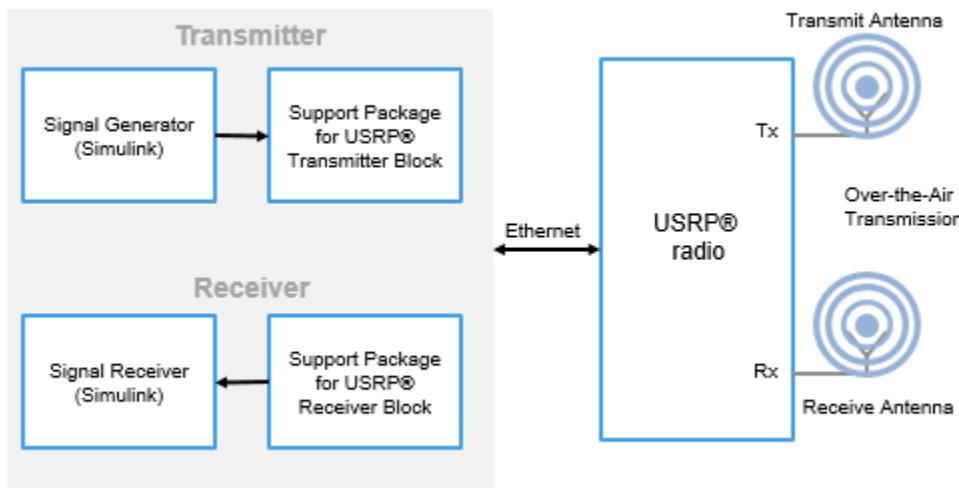


## Description

The SDRu Receiver block supports communication between Simulink and a Universal Software Radio Peripheral (USRP) device, enabling simulation and development for various software-defined radio applications. The SDRu Receiver block and the USRP board must be on the same Ethernet subnetwork.

The SDRu Receiver block receives signal and control data from a USRP board using the Universal Hardware Driver (UHD) from Ettus Research™. The SDRu Receiver block is a Simulink source that receives data from a USRP board and outputs a column vector or matrix signal with a fixed number of rows. The first call to this block can contain transient values, in this case the resulting packets contain undefined data.

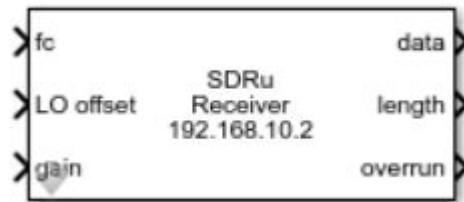
This block diagram illustrates how Simulink, the SDRu Transmitter and Receiver blocks, and the USRP hardware interface.



When this block is called, it is possible that the host has not yet received any data from the USRP hardware. The data length port, **length**, indicates when valid data is present. When the data length port contains a zero value, there is no data. To qualify the execution of part of the model, use the data length with an enabled subsystem.

If your computer is not connected to any USRP hardware, you can still use this block to develop a model that propagates sample time and data type information. To propagate this information, select **Simulation >> Update diagram**.

For information about the USRP hardware products that interface with this block, see the Ettus Research™ website.



This icon shows all ports, including optional ones:

## Ports

### Input

#### **fc — Center frequency setting**

positive scalar

Center frequency setting in Hz, specified as a positive scalar.

Example: 88.9e6 tunes the receiver to a center frequency of 88.9 MHz.

#### Dependencies

To enable this port, set **Source of center frequency** to Input Port.

Data Types: double

#### **LO offset — Local oscillator offset**

scalar

Local oscillator (LO) offset in Hz, specified as a scalar.

Example: 10 sets the LO offset to 10 Hz.

#### Dependencies

To enable this port, set **Source of LO offset** to Input Port.

Data Types: double

#### **gain — Receiver gain setting**

scalar | vector

Receiver gain setting in dB, specified as a scalar or vector. The valid gain range is from -4 dB to 71 dB and depends on the center frequency. An incompatible gain and center frequency combination returns an error from the radio hardware.

Example: 10 sets the receiver gain level to 10 dB.

#### Dependencies

To enable this port, set **Source of gain** to Input Port.

Data Types: double

## Output

### **data — Output data**

vector of complex values

Output data received from the radio hardware, returned as a vector of complex values. The complex output data values range from -1 to 1.

### Dependencies

To specify the base data type, use the **Output data type** parameter.

Data Types: int8 | int16 | single | double

Complex Number Support: Yes

### **length — Length of data received**

nonnegative integer

Length of the data received from the radio hardware, returned as a nonnegative integer. When this port contains a zero value, the received data is not valid. When this port contains a positive value, use the **length** port and an enabled subsystem driven by the **length** signal to qualify the execution of the model. For more information, see “Apply Conditional Execution” on page 4-27.

At the time of radio initialization, there is no valid data. In this case, the output data is set to all zeros and **length** port is set to a zero value. When the host has received enough data from radio, the receiver block outputs valid data and **length** port is set to a positive integer value.

### **overrun — Data discontinuity flag**

0 | 1

Data discontinuity flag, returned:

- 0 — indicates that no data samples were lost.
- 1 — indicates that data samples were lost.

Use this port as a diagnostic tool to determine real-time operation of the SDRu Receiver block. If your model is not running in real time, you can adjust parameters that reduce the number of transported samples. To approach or achieve real-time performance, you can decrease the baseband sampling rate or increase the decimation factor. For more information, see “Detect Underruns and Overruns” on page 4-22.

### Dependencies

To enable this port, select **Enable overrun output port**.

## Parameters

When you set block parameter values, the SDRu Receiver block first checks that the values have the correct data types. Even if the values pass those checks, the values can still be out of range for the radio hardware. In that case, the radio hardware sets the actual value as close to the specified value as possible. When you next synchronize the block with the radio hardware by clicking **Info**, a dialog box open to display the actual values along with other radio information.

If a parameter is listed as tunable, then you can change its value during simulation.

## Radio Connection

### **Platform — Radio to configure**

N200/N210/USRP2 (default) | N300 | N310 | N320/N321 | B200 | B210 | X300 | X310

Radio to configure, specified as one of USRP platforms listed.

### **IP address — IP address of radio hardware**

192.168.10.2 (default) | dotted quad expression

IP address of the radio hardware, specified as a dotted quad expression.

This parameter must match the physical IP address of the radio hardware assigned during hardware setup. See “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with an IP address other than the default, update **IP address** accordingly.

The **IP address** list displays IP addresses for USRP devices attached to the host computer. To specify another known dotted quad IP address, enter it directly into this field.

### Dependencies

To enable this parameter, set **Platform** to N200/N210/USRP2, N300, N310, N320/N321, X300, or X310.

### **Serial number — Serial number of radio hardware**

empty character vector (default) | character vector

Serial number of radio hardware, specified as a character vector.

This parameter must match the serial number of the radio hardware assigned during hardware setup. See “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with a serial number other than the default, update **Serial number** accordingly.

The **Serial number** list displays serial numbers for USRP devices attached to the host computer.

### Dependencies

To enable this parameter, set **Platform** to B200 or B210.

### **Refresh Device List — Refresh the list of connected devices**

button

Refresh the list of all connected devices to update the **IP address** or **Serial number** list. The updated list retains the value in focus before **Refresh Device List** is selected, even if the device with that setting was disconnected.

### **TwinRX Daughterboard — Option to enable TwinRX daughterboard**

off (default) | on

To enable the TwinRX daughterboard on an X-series radio, select this parameter.

When you select this parameter, you can use the **Enable TwinRX Phase Synchronization** parameter to provide phase synchronization between channels of the TwinRX daughterboard.

### Dependencies

To enable this property, set **Platform** to 'X300' or 'X310'.

### **Info — Provides information about device**

button

Provides information about the **Platform** associated with **IP address** or **Serial number**. **IP address** applies for N-series and X-series devices and **Serial number** applies for B-series devices. When you select the **Info** button, a new dialog box opens with information about the specified device.

### **Radio Properties**

#### **Enable TwinRX Phase Synchronization — Option to enable phase synchronization**

off (default) | on

When you select this parameter, the TwinRX daughterboard provides phase synchronization between all channels. In this case, the value of Center Frequency must be the same for all channels.

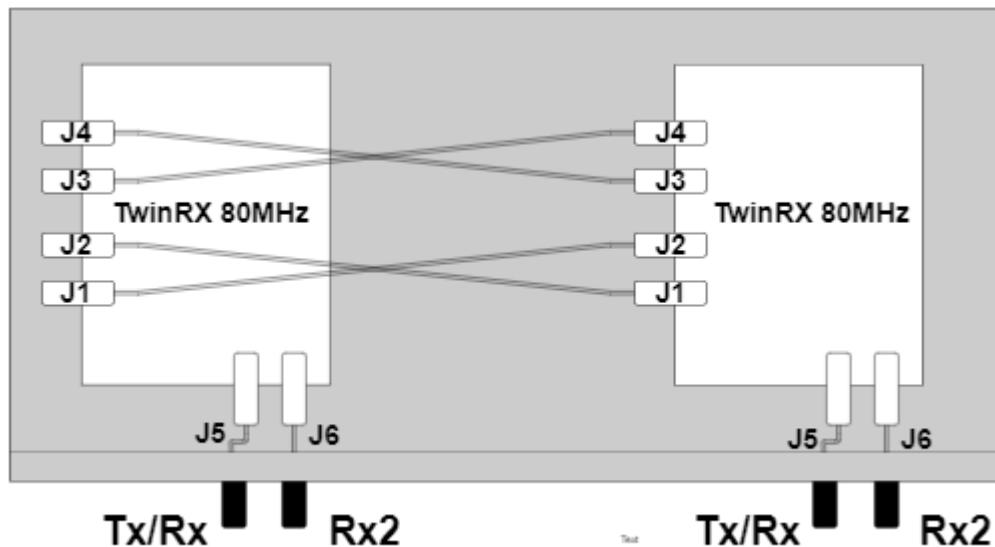
---

**Note** The Local oscillator (LO) source present on channel 1 is the master source to drive other LOs of the TwinRx daughterboard channels.

---

To share LOs between two TwinRx daughterboards, attach the four MMCX RA male cables on one daughterboard to the MMCX RA male cables on the other daughterboard by crisscrossing the cables between the two daughterboards. Make these cable connections, as also shown in the figure.

- J1 to J2
- J2 to J1
- J3 to J4
- J4 to J3



### **Dependencies**

To enable this property, set **Platform** to 'X300' or 'X310' and select **TwinRX Daughterboard**.

**Channel mapping — Channel output mapping for radio or bundled radios**

1 (default) | integer | vector

Channel output mapping for radio or bundled radios, specified as a positive integer scalar or vector. This table shows valid values for various radio platforms.

Platform Value	Channel mapping Value
N200/N210/USRP2	When <b>IP address</b> includes $N$ IP addresses: 1-by- $N$ row vector
N300	1, 2, or [1 2]
N310	Row vector of length [1, 4] with channel numbers as {1, 2, 3, 4}
N320/N321	1, 2, or [1 2]
B200	1
B210	1, 2, or [1 2]
X300 or X310 when <b>TwinRX Daughterboard</b> parameter is cleared	<ul style="list-style-type: none"> <li>When <b>IP address</b> includes one IP address, specify this parameter as 1, 2, or [1 2].</li> <li>When <b>IP address</b> includes <math>N</math> IP addresses, specify this parameter as 1-by-<math>2N</math> row vector.</li> </ul>
X300 or X310 when two TwinRX daughterboards are connected and <b>TwinRX Daughterboard</b> parameter is selected	<p>When you clear the <b>Enable TwinRX Phase Synchronization</b> parameter, specify this parameter as one of these values.</p> <ul style="list-style-type: none"> <li>[<math>N M</math>], where <math>N</math> and <math>M</math> are distinct integers from 1 to 4 — Channels <math>N</math> and <math>M</math> are in use.</li> <li>[<math>N M P</math>], where <math>N</math>, <math>M</math>, and <math>P</math> are distinct integers from 1 to 4 — Channels <math>N</math>, <math>M</math>, and <math>P</math> are in use.</li> <li>[1 2 3 4]</li> </ul> <p>When you select the <b>Enable TwinRX Phase Synchronization</b> parameter, specify this parameter as 1, [1 2], [1 2 3], or [1 2 3 4].</p>

When a scalar, 1 or 2 is specified, the device operates in SISO mode. When a vector is specified, the device operates in MIMO mode. When **IP address** includes multiple IP addresses, the channels defined by **Channel mapping** are sorted, first by the order in which the IP addresses appear in the list and then by the channel order within the same radio.

Example: If Platform is X300 and IP address contains 192.168.20.2, 192.168.10.3, then Channel mapping must be [1 2 3 4]. Channel mapping values 1 and 2 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.2. Channel mapping values 3 and 4 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.10.3.

Data Types: char

**Source of center frequency — Select source of center frequency**

Dialog (default) | Input port

Select source of center frequency, specified as:

- **Dialog** — Set the center frequency using the **Center frequency (Hz)** parameter.
- **Input port** — Set the center frequency using the **fc** input port.

**Center frequency (Hz) — RF center frequency**

2.45e9 (default) | nonnegative finite scalar | vector of nonnegative finite scalars

RF center frequency in Hz, specified as a nonnegative finite scalar or vector of nonnegative finite scalars. The valid range of this parameter depends on the RF daughterboard of the USRP device.

- For a single channel (SISO), specify the value for the center frequency as a scalar.
- For multiple channels (MIMO) that use the same center frequency, specify the center frequency value as a scalar. The center frequency is set by scalar expansion.
- For multiple channels (MIMO) that use different center frequencies, specify the values in a vector. The *i*th element of the vector is applied to the *i*th channel specified by **Channel mapping**.

---

**Note**

- The center frequency for B210 with MIMO must be a scalar. You cannot specify the frequencies as a vector.
  - The channels corresponding to the same RF daughterboard of N310 must have same center frequency value.
- 

When you select the **TwinRX Daughterboard** parameter:

- To tune all channels to the same frequency, select the **Enable TwinRX Phase Synchronization** parameter and set the center frequency as a scalar or row vector of the same values.
- To tune channels to different frequencies, clear the **Enable TwinRX Phase Synchronization** parameter and set the center frequency to a row vector. Each value in the row vector specifies the frequency of the corresponding channel.

---

**Note** When you select the **TwinRX Daughterboard** and **Enable TwinRX Phase Synchronization**, the LO source present on channel 1 is the master source to drive other LOs of the TwinRX daughterboard channels. In this case, the Center frequency value must be the same for all channels of the TwinRX daughterboard.

---

For more information, see “Enable TwinRX Phase Synchronization” on page 7-0 .

**Tunable:** Yes

**Dependencies**

To enable this parameter, set the **Source of center frequency** to **Dialog**.

Data Types: double

**Source of LO offset — Select source of LO offset**

Dialog (default) | Input port

Select source of LO offset, specified as:

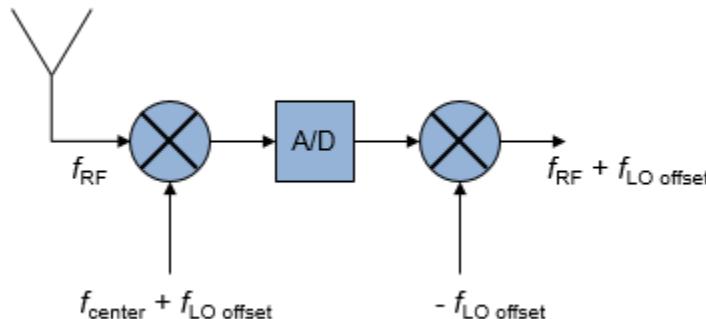
- **Dialog** — Set the LO offset using the **LO offset (Hz)** parameter.
- **Input port** — Set the LO offset using the **LO offset** input port.

#### **LO offset (Hz) — Offset frequency for local oscillator**

0 (default) | scalar | vector

Offset frequency for the local oscillator, specified as a scalar or vector. The valid range of this parameter depends on the RF daughterboard of the USRP device.

As shown in this illustration, the local oscillator offset affects the intermediate center frequency in the USRP hardware. It does not affect the received center frequency.



- $f_{RF}$  represents the received RF frequency.
- $f_{center}$  represents the center frequency specified by the block.
- $f_{LO\ offset}$  is the local oscillator offset frequency.
- Ideally,  $f_{RF} - f_{center} = 0$ .

To move the center frequency away from interference or harmonics generated by the USRP hardware, use the LO offset.

- For a single channel (SISO), specify the value for the LO offset as a scalar.
- For multiple channels (MIMO), the LO offset must be zero. This restriction is due to a UHD limitation. You can specify the LO offset as a scalar or as a vector.

#### **Dependencies**

To enable this parameter, set **Source of LO offset** to **Dialog**.

#### **Source of gain — Select source of gain**

Dialog (default) | Input port

Select source of gain, specified as:

- **Dialog** — Specify the gain using the **Gain (dB)** parameter.
- **Input port** — Specify the gain using the **gain** input port.

**Gain (dB) — Receiver gain**

32 (default) | scalar | vector

Receiver gain in dB, specified as a scalar or vector. The valid gain range is from -4 dB to 71 dB and depends on the center frequency. An incompatible gain and center frequency combination returns an error from the radio hardware.

Set the value of gain based on the **Channel Mapping** configuration:

- For a single channel (SISO), specify the gain as a scalar.
- For multiple channels (MIMO) that use the same gain value, specify the gain as a scalar. The gain is set by scalar expansion.
- For multiple channels (MIMO) that use different gains, specify the values in a row vector. The *i*th element of the vector is applied to the *i*th channel specified by **Channel Mapping**.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Source of gain** to Dialog.

Data Types: double

**PPS source — Pulse per second (PPS) signal source**

Internal (default) | External

Pulse per second (PPS) signal source, specified as:

- Internal — Uses the internal PPS signal of the USRP radio.
- External — Uses the PPS signal from an external signal generator.

To synchronize the time for all channels of the bundled radios, provide a common external PPS signal to all the bundled radios and set **PPS source** to External.

**Clock source — Clock source**

Internal (default) | External

Clock source, specified as:

- Internal — Uses the internal clock signal of the USRP radio.
- External — Uses the 10 MHz clock signal from an external clock generator.

For B-series radios, the external clock port is labeled "10 MHz". For N3xx, N2xx, USRP2, and X-series radios, the external clock port is labeled "REF IN".

To synchronize the frequency for all channels of the bundled radios, provide a common external 10 MHz clock signal to all the bundled radios and set **Clock source** to External.

**Master clock rate (Hz) — Master clock rate**

scalar

Master clock rate, specified as a scalar in Hz. The master clock rate is the A/D and D/A clock rate. The valid range of values for this property depends on the radio platform that is connected.

<b>Platform Value</b>	<b>Possible Master clock rate (Hz) Value</b>
N200/N210/USRP2	100e6 Hz. Read-only.
N300 or N310	122.88e6 Hz, 125e6 Hz, or 153.6e6 Hz Default value is 125e6 Hz.
N320/N321	200e6 Hz, 245.76e6 Hz, or 250e6 Hz Default value is 200e6 Hz.
B200 or B210	From 5e6 Hz to 56e6 Hz. When using B210 with multiple channels, the clock rate must be no higher than 30.72e6 Hz. This restriction is a hardware limitation for the B210 radios only when using two-channel operations. Default value is 32e6 Hz.
X300 or X310	184.32e6 Hz or 200e6 Hz 200e6 Hz — When TwinRX daughterboard is selected. Default value 200e6 Hz.

### Dependencies

To enable this parameter, set **Platform** to B200, B210, N300, N310, N320/N321, X300, or X310.

Data Types: double

### Decimation factor — Decimation factor for SDRu Receiver

512 (default) | integer

Decimation factor for the SDRu receiver, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

<b>DecimationFact or Property Value</b>	<b>B-Series</b>	<b>N2xx-Series</b>	<b>N3xx-Series</b>	<b>X-Series</b>
1	Valid	Not valid	Valid	Not valid when connected with TwinRX daughterboard
2	Valid	Acceptable when you use only the int8 transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid

DecimationFact or Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Valid

The radio uses the decimation factor when it downconverts the intermediate frequency (IF) signal to a complex baseband signal.

## Data

**Baseband sample rate (Hz) — Baseband sample rate of output signal**  
scalar

This parameter is read-only.

Baseband sample rate of output signal, in Hz, specified as a scalar.

This parameter displays the computed baseband sample rate derived from the **Master clock rate (Hz)** and **Decimation factor** parameter values. This computation uses the formula, **Baseband sample rate** = Master clock rate/Decimation factor. If you change the **Decimation factor** during simulation, then the block changes hardware data rate, but does not change the Simulink sample time.

To get more information on sample time, see “What Is Sample Time?” (Simulink).

Data Types: double

**Transport data type — Transport data type**  
int16 (default) | int8

Transport data type, specified as:

- **int16** — Uses 16-bit transport. Achieves higher precision than 8-bit transport.
- **int8** — Uses 8-bit transport. Uses a quantization step 256 times larger and achieves approximately two times faster transport data rate than 16-bit transport.

Specifying transport data rate data type as **int16**, assigns 16 bits for the in-phase component and 16 bits for the quadrature component, resulting in 32 bits for each complex sample of transport data.

**Output data type — Data type of output signal**

Same as transport data type (default) | double | single

Data type of the output signal, specified as **Same as transport data type**, **double**, or **single**. The complex output data values range from -1 to 1.

**Samples per frame — Number of samples per frame of output signal**

362 | positive integer

Number of samples per frame of the output signal that the object generates, specified as a positive integer scalar. This value optimally utilizes the underlying Ethernet packets, which have a size of 1500 8-bit bytes.

#### **Enable burst mode — Option to enable burst mode**

off (default) | on

Option to enable burst mode. To produce a set of contiguous frames without an overrun or underrun to the radio, select **Enable burst mode**. Enabling burst mode helps you simulate models that cannot run in real time.

When burst mode is enabled, specify the desired amount of contiguous data using the **Number of frames in burst** parameter. For more information, see “Detect Underruns and Overruns” on page 4-22.

#### **Number of frames in burst — Number of frames in contiguous burst**

100 (default) | integer

Number of frames in a contiguous burst, specified as an integer.

#### **Dependencies**

To enable this parameter, select **Enable burst mode**.

## **More About**

### **Single and Multiple Channel Output**

- N200, N210, USRP2, and B200 radios support a single channel that you can use to:
  - Send data with the SDRu Transmitter block. The SDRu Transmitter block receives a column vector signal of fixed length from Simulink.
  - Receive data with the SDRu Receiver block. The SDRu Receiver block outputs a column vector signal of fixed length to Simulink.
- B210, X300, and X310 radios support two channels that you can use to transmit and receive data with SDRu blocks. You can use both channels or only a single channel (either channel 1 or 2).
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.
  - The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.

---

**Note** When two TwinRX daughterboards are connected to X300 or X310 radio, the radio supports up to four channel reception.

---

- N300, N320, and N321 radios support two channels that you can use to transmit and receive data with SDRu blocks. You can use both channels or only a single channel (either channel 1 or 2).
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.
  - The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.

- N310 radio support four channels that you can use to transmit and receive data with SDRu blocks. You can use up to four channels.
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.
  - The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.

You can set the **Center frequency**, **LO offset**, and **Gain** block parameters independently for each channel or apply the same setting to all channels. All other block parameter values apply to all channels.

For more information, see “Single Channel Input and Output Operations” on page 4-8 and “Multiple Channel Input and Output Operations” on page 4-14.

## Compatibility Considerations

### **decim input port has been removed**

*Errors starting in R2020a*

Starting in R2020a, the SDRu Receiver block removed the **decim** input port. You can not set the decimation factor value for this block by using an input port. Instead use the **Decimation factor** parameter.

### **Sample time parameter has been removed**

*Warns starting in R2020a*

Starting in R2020a, the SDRu Receiver block removed the **Sample time** parameter. Use the read-only **Baseband sample rate (Hz)** parameter instead. This parameter displays the computed baseband sample rate derived from the **Master clock rate (Hz)** and **Decimation factor** parameters.

### **X3xx series radios no longer support 120 MHz master clock rate**

*Errors starting in R2020a*

Beginning with Ettus Research UHD version 003.014.000.000, X3xx series radios do not support a master clock rate value of 120 MHz. Consequently, starting in R2020a, which supports UHD version 003.015.000.000, Communications Toolbox Support Package for USRP Radio does not support a master clock rate value of 120 MHz for X3xx series radios.

For the SDRu Transmitter and SDRu Receiver blocks, when you select an X3xx series radio for the **Platform** parameter, you can no longer set the **Master clock rate (Hz)** parameter to 120e6.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## See Also

### **Blocks**

SDRu Transmitter

**Objects**

`comm.SDRuReceiver | comm.SDRuTransmitter`

**Topics**

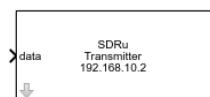
- “Single Channel Input and Output Operations” on page 4-8
- “Multiple Channel Input and Output Operations” on page 4-14
- “Apply Conditional Execution” on page 4-27
- “Detect Underruns and Overruns” on page 4-22
- “Burst-Mode Buffering” on page 5-2

**Introduced in R2011b**

# SDRu Transmitter

Send data to USRP device

**Library:** Communications Toolbox Support Package for USRP Radio

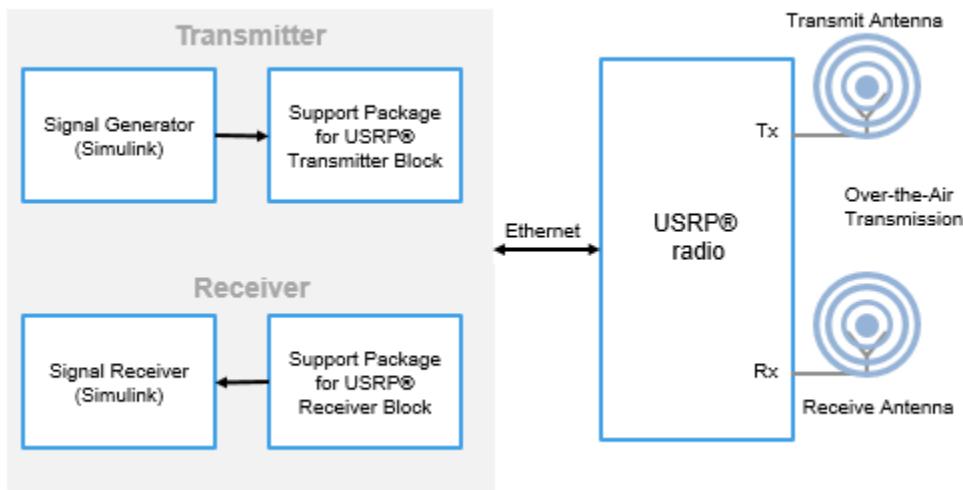


## Description

The SDRu Transmitter block supports communication between Simulink and a Universal Software Radio Peripheral (USRP) device, enabling simulation and development for various software-defined radio applications. The SDRu Transmitter block and the USRP board must be on the same Ethernet subnetwork.

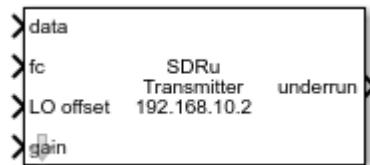
The SDRu Transmitter block accepts a column vector or matrix input signal from Simulink and transmits signal and control data to a USRP board using the Universal Hardware Driver (UHD) from Ettus Research™. The SDRu Transmitter block is a Simulink sink that takes data from a model and sends it to a USRP board. The first call to this block can contain transient values, in this case the resulting packets contain undefined data.

This block diagram illustrates how Simulink, SDRu Transmitter and Receiver blocks, and USRP hardware interface.



If your computer is not connected to any USRP hardware, you can still use this block to develop a model that propagates sample time and data type information. To propagate this information, select **Simulation >> Update diagram**.

For information about the USRP hardware products that interface with this block, see the Ettus Research™ website.



This icon shows all ports, including optional ones:

## Ports

### Input

#### **data — Input data**

column vector | matrix

Input data sent to the radio hardware, specified as a column vector or matrix. For a single channel radio, **data** is a column vector. For a multichannel radio, **data** is a or matrix, where each column contains a channel of data). The range of values for **data** is [ -1 1] for double or single precision data and [ -32768, 32767] for int16 data.

#### Dependencies

To specify the output data type, use the **Output data type** parameter.

Data Types: int16 | single | double

Complex Number Support: Yes

#### **fc — Center frequency setting**

positive scalar

Center frequency setting in Hz, specified as a positive scalar.

Example: 88.9e6 tunes the transmitter to a center frequency of 88.9 MHz.

#### Dependencies

To enable this port, set **Source of center frequency** to **Input Port**.

Data Types: double

#### **LO offset — Local oscillator offset**

scalar

Local oscillator (LO) offset in Hz, specified as a scalar.

Example: 10 sets the LO offset to 10 Hz.

#### Dependencies

To enable this port, set **Source of LO offset** to **Input Port**.

Data Types: double

#### **gain — Transmitter gain setting**

scalar | vector

Transmitter gain setting in dB, specified as a scalar or vector. The valid gain range is from 0 dB to 31.5 dB and depends on the center frequency. An incompatible gain and center frequency combination returns an error from the radio hardware.

Transmitter gain setting accounts combined analog and digital transmitter gains of the USRP hardware.

- For a single channel, specify the value for the gain as a scalar in dB.
- For multiple channels that use the same gain value, specify the gain as a scalar in dB. The gain is set by scalar expansion.
- For multiple channels that use different gains, specify the values in a vector, for example, [5 12]. The Nth element of the vector is applied to the Nth channel specified by **Channel mapping**.

Example: 10 sets the transmitter gain level to 10 dB.

#### Dependencies

To enable this port, set **Source of gain** to **Input Port**.

Data Types: double

#### Output

##### **underrun — Data discontinuity flag**

0 | 1

Data discontinuity flag, returned:

- 0 — indicates that no data samples were lost.
- 1 — indicates that data samples were lost.

Use this port as a diagnostic tool to determine real-time operation of the SDRu Transmitter block. If your model is not running in real time, you can adjust parameters that reduce the number of transported samples. To approach or achieve real-time performance, you can increase the interpolation factor. For more information, see “Detect Underruns and Overruns” on page 4-22.

#### Dependencies

To enable this port, select **Enable overrun output port**.

## Parameters

When you set block parameter values, the SDRu Transmitter block first checks that the values have the correct data types. Even if the values pass those checks, the values can still be out of range for the radio hardware. In that case, the radio hardware sets the actual value as close to the specified value as possible. When you next synchronize the block with the radio hardware by clicking **Info**, a dialog box open to display the actual values along with other radio information.

If a parameter is listed as tunable, then you can change its value during simulation.

#### Radio Connection

##### **Platform — Radio to configure**

N200/N210/USRP2 (default) | N300 | N310 | N320/N321 | B200 | B210 | X300 | X310

Radio to configure, specified as one of USRP platforms listed.

##### **IP address — IP address of radio hardware**

192.168.10.2 (default) | dotted quad expression

IP address of the radio hardware, specified as a dotted quad expression.

This parameter must match the physical IP address of the radio hardware assigned during hardware setup. See “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with an IP address other than the default, update **IP address** accordingly.

The **IP address** list displays IP addresses for USRP devices attached to the host computer. To specify another known dotted quad IP address, enter it directly into this field.

#### **Dependencies**

To enable this parameter, set **Platform** to N200/N210/USRP2, N300, N310, N320/N321, X300, or X310.

#### **Serial number — Serial number of radio hardware**

empty character vector (default) | character vector

Serial number of radio hardware, specified as a character vector.

This parameter must match the serial number of the radio hardware assigned during hardware setup. See “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with a serial number other than the default, update **Serial number** accordingly.

The **Serial number** list displays serial numbers for USRP devices attached to the host computer.

#### **Dependencies**

To enable this parameter, set **Platform** to B200 or B210.

#### **Refresh Device List — Refresh the list of connected devices**

button

Refresh the list of all connected devices to update the **IP address** or **Serial number** list. The updated list retains the value in focus before **Refresh Device List** is selected, even if the device with that setting was disconnected.

#### **Info — Provides information about device**

button

Provides information about the **Platform** associated with **IP address** or **Serial number**. **IP address** applies for N-series and X-series devices and **Serial number** applies for B-series devices. When you select the **Info** button, a new dialog box opens with information about the specified device.

### **Radio Properties**

#### **Channel mapping — Channel output mapping for radio or bundled radios**

1 (default) | integer | vector

Channel output mapping for radio or bundled radios, specified as a positive integer scalar or vector. This table shows valid values for various radio platforms.

Platform Value	Possible Channel mapping Value
N200/N210/USRP2	When <b>IP address</b> includes $N$ IP addresses: 1-by- $N$ row vector

Platform Value	Possible Channel mapping Value
N300	1, 2, or [1 2]
N310	Row vector of length [1, 4] with channel numbers as {1, 2, 3, 4}
N320/N321	1, 2, or [1 2]
B200	1
B210	1, 2, or [1 2]
X300 or X310	<ul style="list-style-type: none"> <li>• When <b>IP address</b> includes one IP address: 1, 2, or [1 2]</li> <li>• When <b>IP address</b> includes <math>N</math> IP addresses: 1-by-<math>2N</math> row vector</li> </ul>

When a scalar, 1 or 2 is specified, the device operates in SISO mode. When a vector is specified, the device operates in MIMO mode. When **IP address** includes multiple IP addresses, the channels defined by **Channel mapping** are sorted, first by the order in which the IP addresses appear in the list and then by the channel order within the same radio.

Example: If Platform is X300 and IP address contains 192.168.20.2, 192.168.10.3, then Channel mapping must be [1 2 3 4]. Channel mapping values 1 and 2 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.2. Channel mapping values 3 and 4 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.10.3.

Data Types: char

#### Source of center frequency — Select source of center frequency

Dialog (default) | Input port

Select source of center frequency, specified as:

- Dialog — Set the center frequency using the **Center frequency (Hz)** parameter.
- Input port — Set the center frequency using the **fc** input port.

#### Center frequency (Hz) — RF center frequency

2.45e9 (default) | nonnegative finite scalar | vector of nonnegative finite scalars

RF center frequency in Hz, specified as a nonnegative finite scalar or vector of nonnegative finite scalars. The valid range of this parameter depends on the RF daughterboard of the USRP device.

- For a single channel (SISO), specify the value for the center frequency as a scalar.
- For multiple channels (MIMO) that use the same center frequency, specify the center frequency value as a scalar. The center frequency is set by scalar expansion.
- For multiple channels (MIMO) that use different center frequencies, specify the values in a vector. The  $i$ th element of the vector is applied to the  $i$ th channel specified by **Channel mapping**.

---

#### Note

- The center frequency for B210 with MIMO must be a scalar. You cannot specify the frequencies as a vector.
  - The channels corresponding to the same RF daughterboard of N310 must have same center frequency value.
-

**Tunable:** Yes

#### Dependencies

To enable this parameter, set the **Source of center frequency** to Dialog.

Data Types: double

#### Source of LO offset — Select source of LO offset

Dialog (default) | Input port

Select source of LO offset, specified as:

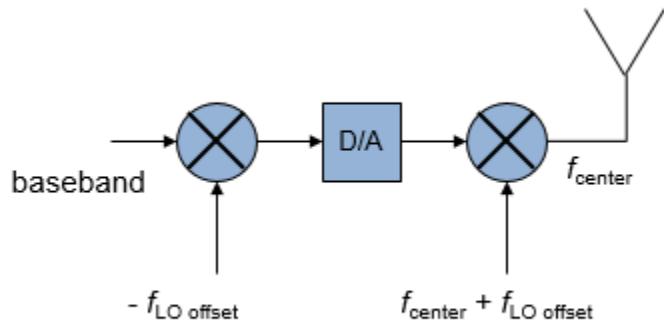
- Dialog — Set the LO offset using the **LO offset (Hz)** parameter.
- Input port — Set the LO offset using the **LO offset** input port.

#### LO offset (Hz) — Offset frequency for local oscillator

0 (default) | scalar | vector

Offset frequency for the local oscillator, specified as a scalar or vector. The valid range of this parameter depends on the RF daughterboard of the USRP device.

As shown in this illustration, the local oscillator offset affects the intermediate center frequency in the USRP hardware. It does not affect the transmitted center frequency.



- $f_{\text{LO offset}}$  is the local oscillator offset frequency.
- $f_{\text{center}}$  represents the center frequency specified by the block.

To move the center frequency away from interference or harmonics generated by the USRP hardware, use the LO offset.

- For a single channel (SISO), specify the value for the LO offset as a scalar.
- For multiple channels (MIMO), the LO offset must be zero. This restriction is due to a UHD limitation. You can specify the LO offset as a scalar or as a vector.

#### Dependencies

To enable this parameter, set **Source of LO offset** to Dialog.

**Source of gain — Select source of gain**

Dialog (default) | Input port

Select source of gain, specified as:

- Dialog — Specify the gain using the **Gain (dB)** parameter.
- Input port — Specify the gain using the **gain** input port.

**Gain (dB) — Transmitter gain**

8 (default) | scalar | vector

Transmitter gain in dB, specified as a scalar or vector. The valid gain range is from XX dB to YY dB and depends on the center frequency. An incompatible gain and center frequency combination returns an error from the radio hardware.

Set the value of gain based on the **Channel Mapping** configuration:

- For a single channel (SISO), specify the gain as a scalar.
- For multiple channels (MIMO) that use the same gain value, specify the gain as a scalar. The gain is set by scalar expansion.
- For multiple channels (MIMO) that use different gains, specify the values in a row vector. The *i*th element of the vector is applied to the *i*th channel specified by **Channel Mapping**.

**Tunable:** Yes

**Dependencies**

To enable this parameter, set **Source of gain** to **Dialog**.

Data Types: double

**PPS source — Pulse per second (PPS) signal source**

Internal (default) | External

Pulse per second (PPS) signal source, specified as:

- Internal — Uses the internal PPS signal of the USRP radio.
- External — Uses the PPS signal from an external signal generator.

To synchronize the time for all channels of the bundled radios, provide a common external PPS signal to all the bundled radios and set **PPS source** to **External**.

**Clock source — Clock source**

Internal (default) | External

Clock source, specified as:

- Internal — Uses the internal clock signal of the USRP radio.
- External — Uses the 10 MHz clock signal from an external clock generator.

For B-series radios, the external clock port is labeled "10 MHz". For N3xx, N2xx, USRP2, and X-series radios, the external clock port is labeled "REF IN".

To synchronize the frequency for all channels of the bundled radios, provide a common external 10 MHz clock signal to all the bundled radios and set **Clock source** to **External**.

### **Master clock rate (Hz) — Master clock rate**

scalar

Master clock rate, specified as a scalar in Hz. The master clock rate is the A/D and D/A clock rate. The valid range of values for this property depends on the radio platform that is connected.

Platform Value	Possible Master clock rate (Hz) Value
N200/N210/USR2	100e6 Hz. Read-only.
N300 or N310	122.88e6 Hz, 125e6 Hz, or 153.6e6 Hz Default value is 125e6 Hz.
N320/N321	200e6 Hz, 245.76e6 Hz, or 250e6 Hz Default value is 200e6 Hz.
B200 or B210	From 5e6 to 56e6 Hz. When using B210 with multiple channels, the clock rate must be no higher than 30.72e6 Hz. This restriction is a hardware limitation for the B210 radios only when using two-channel operations. Default value is 32e6.
X300 or X310	184.32e6 Hz or 200e6 Hz Default value is 200e6 Hz.

#### **Dependencies**

To enable this parameter, set **Platform** to B200, B210, N300, N310, N320/N321, X300, or X310.

Data Types: double

### **Interpolation factor — Interpolation factor for SDRu Transmitter**

512 (default) | integer

Interpolation factor for the SDRu transmitter, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

Interpolation Factor Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
1	Valid	Not valid	Valid	Valid
2	Valid	Acceptable when you use only the int8 transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid

Interpolation Factor Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Not valid

The radio uses the interpolation factor when it upconverts the complex baseband signal to an intermediate frequency (IF) signal.

## Data

**Baseband sample rate (Hz) — Baseband sample rate of output signal**  
scalar

This parameter is read-only.

Baseband sample rate of output signal, in Hz, specified as a scalar.

This parameter displays the computed baseband sample rate derived from the **Master clock rate (Hz)** and **Interpolation factor** parameter values. This computation uses the formula, **Baseband sample rate** = Master clock rate/Interpolation factor. If you change the **Interpolation factor** during simulation, then the block changes hardware data rate, but does not change the Simulink sample time.

To get more information on sample time, see “What Is Sample Time?” (Simulink).

Data Types: double

**Transport data type — Transport data type**

int16 (default) | int8

Transport data type, specified as:

- int16 — Uses 16-bit transport. Achieves higher precision than 8-bit transport.
- int8 — Uses 8-bit transport. Uses a quantization step 256 times larger and achieves approximately two times faster transport data rate than 16-bit transport.

Specifying transport data rate data type as int16, assigns 16 bits for the in-phase component and 16 bits for the quadrature component, resulting in 32 bits for each complex sample of transport data.

**underrun — Data discontinuity flag**

0 | 1

Data discontinuity flag, returned:

- 0 — indicates that no data samples were lost.
- 1 — indicates that data samples were lost.

Use this port as a diagnostic tool to determine real-time operation of the SDRu Transmitter block. If your model is not running in real time, you can adjust parameters that reduce the number of

transported samples. To approach or achieve real-time performance, you can decrease the interpolation factor. For more information, see “Detect Underruns and Overruns” on page 4-22.

#### **Dependencies**

To enable this port, select **Enable overrun output port**.

#### **Enable burst mode — Option to enable burst mode**

off (default) | on

Option to enable burst mode. To produce a set of contiguous frames without an overrun or underrun to the radio, select **Enable burst mode**. Enabling burst mode helps you simulate models that cannot run in real time.

When burst mode is enabled, specify the desired amount of contiguous data using the **Number of frames in burst** parameter. For more information, see “Detect Underruns and Overruns” on page 4-22.

#### **Number of frames in burst — Number of frames in contiguous burst**

100 (default) | integer

Number of frames in a contiguous burst, specified as an integer.

#### **Dependencies**

To enable this parameter, select **Enable burst mode**.

## **More About**

### **Single and Multiple Channel Output**

- N200, N210, USRP2, and B200 radios support a single channel that you can use to:
  - Send data with the SDRu Transmitter block. The SDRu Transmitter block receives a column vector signal of fixed length from Simulink.
  - Receive data with the SDRu Receiver block. The SDRu Receiver block outputs a column vector signal of fixed length to Simulink.
- B210, X300, and X310 radios support two channels that you can use to transmit and receive data with SDRu blocks. You can use both channels or only a single channel (either channel 1 or 2).
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.
  - The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.

---

**Note** When two TwinRX daughterboards are connected to X300 or X310 radio, the radio supports up to four channel reception.

- N300, N320, and N321 radios support two channels that you can use to transmit and receive data with SDRu blocks. You can use both channels or only a single channel (either channel 1 or 2).
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.

- The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.
- N310 radio support four channels that you can use to transmit and receive data with SDRu blocks. You can use up to four channels.
  - The SDRu Transmitter block receives a matrix signal, where each column is a channel of data of fixed length.
  - The SDRu Receiver block outputs a matrix signal, where each column is a channel of data of fixed length.

You can set the **Center frequency**, **LO offset**, and **Gain** block parameters independently for each channel or apply the same setting to all channels. All other block parameter values apply to all channels.

For more information, see “Single Channel Input and Output Operations” on page 4-8 and “Multiple Channel Input and Output Operations” on page 4-14.

## Compatibility Considerations

### **interp input port has been removed**

*Errors starting in R2020a*

Starting in R2020a, the SDRu Transmitter block removed the **interp** input port. You can not set the interpolation factor value for this block by using an input port. Instead use the **Interpolation factor** parameter.

### **Sample time parameter has been removed**

*Warns starting in R2020a*

Starting in R2020a, the SDRu Transmitter block removed the **Sample time** parameter. Use the read-only **Baseband sample rate (Hz)** parameter instead. This parameter displays the computed baseband sample rate derived from the **Master clock rate (Hz)** and **Interpolation factor** parameters.

### **X3xx series radios no longer support 120 MHz master clock rate**

*Errors starting in R2020a*

Beginning with Ettus Research UHD version 003.014.000.000, X3xx series radios do not support a master clock rate value of 120 MHz. Consequently, starting in R2020a, which supports UHD version 003.015.000.000, Communications Toolbox Support Package for USRP Radio does not support a master clock rate value of 120 MHz for X3xx series radios.

For the SDRu Transmitter and SDRu Receiver blocks, when you select an X3xx series radio for the **Platform** parameter, you can no longer set the **Master clock rate (Hz)** parameter to 120e6.

## Extended Capabilities

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

## See Also

### Blocks

SDRu Receiver

### Objects

`comm.SDRuReceiver` | `comm.SDRuTransmitter`

### Topics

- “Single Channel Input and Output Operations” on page 4-8
- “Multiple Channel Input and Output Operations” on page 4-14
- “Apply Conditional Execution” on page 4-27
- “Detect Underruns and Overruns” on page 4-22
- “Burst-Mode Buffering” on page 5-2

### Introduced in R2011b



# System Objects

---

## comm.SDRuReceiver

**Package:** comm

Receive data from USRP device

### Description

The SDRuReceiver System object receives data from a Universal Software Radio Peripheral (USRP) hardware device, enabling simulation and development for various software-defined radio applications. The object enables communication with a USRP board on the same Ethernet subnetwork or a USRP board via a USB connection. You can write a MATLAB application that uses the System object, or you can generate code for the System object without connecting to a USRP radio.

This object receives signal and control data from a USRP board using the Universal Hardware Driver (UHD) from Ettus Research. The System object receives data from a USRP board and outputs a column vector or matrix signal of a fixed number of rows.

To receive data from a USRP device:

- 1 Create the `comm.SDRuReceiver` object and set its properties.
- 2 Call the object as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#).

## Creation

### Syntax

```
tx = comm.SDRuReceiver  
tx = comm.SDRuReceiver(address)  
tx = comm.SDRuReceiver(___,Name,Value)
```

### Description

`tx = comm.SDRuReceiver` creates a default SDRu receiver System object.

`tx = comm.SDRuReceiver(address)` sets the `IPAddress` property to address of the connected USRP device.

`tx = comm.SDRuReceiver(___,Name,Value)` sets “Properties” on page 8-2 using one or more name-value pairs in addition to any input argument combination from previous syntaxes. Enclose each property name in quotes. For example, `'CenterFrequency',5e6` specifies the center frequency as 5 MHz.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

#### **Platform — Model number of radio**

'N200/N210/USRP2' (default) | 'N300' | 'N310' | 'N320/N321' | 'B200' | 'B210' | 'X300' | 'X310'

Model number of the radio, specified as one of these values.

- 'N200/N210/USRP2'
- 'N300'
- 'N310'
- 'N320/N321'
- 'B200'
- 'B210'
- 'X300'
- 'X310'

Data Types: char | string

#### **IPAddress — IP address of USRP device**

'192.168.10.2' (default) | dotted-quad character vector | dotted-quad string scalar

IP address of the USRP device, specified as a dotted-quad character vector or dotted-quad string scalar. When you specify more than one IP address, you must separate each address by commas or spaces.

This value must match the physical IP address of the radio hardware assigned during hardware setup. For more information, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with an IP address other than the default, update this property accordingly.

To find the logical network location of all connected USRP radios, use the **findsdr** function.

Example: '192.168.10.2, 192.168.10.5' or '192.168.10.2 192.168.10.5' specifies IP addresses for two devices.

#### **Dependencies**

To enable this property, set Platform to 'N200/N210/USRP2', 'N300', 'N310', 'N320/N321', 'X300', or 'X310'.

Data Types: char | string

#### **SerialNum — Serial number of radio**

'' (default) | character vector | string scalar

Serial number of the radio hardware, specified as a character vector or string scalar.

This property must match the serial number of the radio hardware assigned during hardware setup. For more information, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If

you configure the radio hardware with a serial number other than the default, update this property accordingly.

**Dependencies**

To enable this property, set `Platform` to 'B200' or 'B210'.

Data Types: `char` | `string`

**IsTwinRXDaughterboard — Option to enable TwinRX daughterboard**  
`false` or `0` (default) | `true` or `1`

Option to enable the TwinRX daughterboard, specified as a numeric or logical `0` (`false`) or `1` (`true`). To enable the TwinRX daughterboard on an X-series radio, set `IsTwinRXDaughterboard` to `1` (`true`).

When you enable the TwinRX daughterboard, you can use the `EnableTwinRXPhaseSynchronization` property to provide phase synchronization between channels of the TwinRX daughterboard.

**Dependencies**

To enable this property, set the `Platform` property to 'X300' or 'X310'.

Data Types: `logical` | `numeric`

**EnableTwinRXPhaseSynchronization — Option to enable phase synchronization**  
`false` or `0` (default) | `true` or `1`

Flag to enable phase synchronization between channels of the TwinRX daughterboard, specified as a numeric or logical `0` (`false`) or `1` (`true`). When you set this property to `1` (`true`), TwinRX daughterboard provides phase synchronization between all the channels. In this case, the value of the `CenterFrequency` property must be same for all the channels.

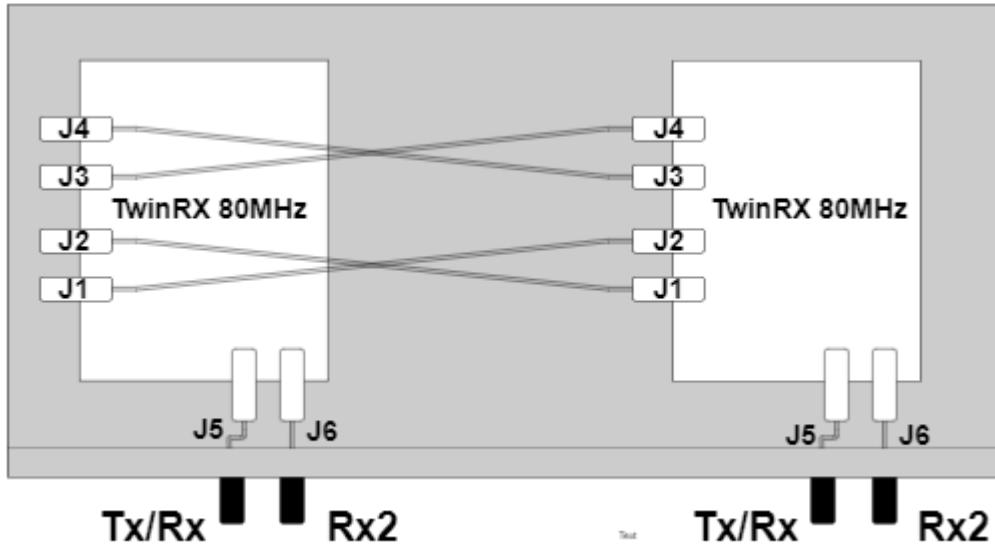
---

**Note** The local oscillator (LO) source present on the channel 1 is the master source to drive other LOs of the TwinRx daughterboard channels.

---

To share LOs between two TwinRx daughterboards, attach the four MMCX RA male cables on one daughterboard to the MMCX RA male cables on the other daughterboard by crisscrossing the cables between the two daughterboards. Make these cable connections, as also shown in the figure.

- J1 to J2
- J2 to J1
- J3 to J4
- J4 to J3



### Dependencies

To enable this property, set the `Platform` property to 'X300' or 'X310' and `IsTwinRXDaughterboard` property to 1 (true).

Data Types: logical | numeric

### ChannelMapping — Channel mapping for radio or bundled radios

1 (default) | nonnegative scalar | row vector of nonnegative values

Channel mapping for the radio or bundled radios, specified as a nonnegative scalar or a row vector of nonnegative values. This table shows the valid values for various radio platforms.

Platform Property Value	ChannelMapping Property Value
'N200/N210/USRP2'	1-by- $N$ row vector, where $N$ is the number of IP addresses included in the <code>IPAddress</code> property
'N300'	1, 2, or [1 2]
'N310'	1-, 2-, 3-, or 4-element row vector of channel numbers from the set {1, 2, 3, 4}
'N320/N321'	1, 2, or [1 2]
'B200'	1
'B210'	1, 2, or [1 2]
'X300' or 'X310' when the <code>IsTwinRXDaughterboard</code> property is 0 (false)	<ul style="list-style-type: none"> <li>when <code>IPAddress</code> includes one IP address, specify this property as 1, 2, or [1 2]</li> <li>When <code>IPAddress</code> includes <math>N</math> IP addresses, specify the property as 1-by-<math>2N</math> row vector. <math>N</math> is the number of IP addresses included in <code>IPAddress</code>.</li> </ul>

Platform Property Value	ChannelMapping Property Value
'X300' or 'X310' when two TwinRX daughterboards are connected and the IsTwinRXDaughterboard property is 1 (true)	<p>When the <code>EnableTwinRXPhaseSynchronization</code> property is 0 (false), specify this property as one of these values.</p> <ul style="list-style-type: none"> <li>• <math>[N M]</math>, where <math>N</math> and <math>M</math> are distinct integers from 1 to 4 — Channels <math>N</math> and <math>M</math> are in use.</li> <li>• <math>[N M P]</math>, where <math>N</math>, <math>M</math>, and <math>P</math> are distinct integers from 1 to 4 — Channels <math>N</math>, <math>M</math>, and <math>P</math> are in use.</li> <li>• <math>[1 2 3 4]</math></li> </ul> <p>When the <code>EnableTwinRXPhaseSynchronization</code> property is 1 (true), specify this property as 1, [1 2], [1 2 3], or [1 2 3 4].</p>

When `IPAddress` includes multiple IP addresses, the channels defined by `ChannelMapping` are ordered first by the order in which the IP addresses appear in the list and then by the channel order within the same radio.

Example: If `Platform` is 'X300' and `IPAddress` is '192.168.20.2, 192.168.10.3', then `ChannelMapping` must be [1 2 3 4]. Channels 1, 2, 3, and 4 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.2 and channels 1 and 2 of the radio with IP address 192.168.10.3.

Data Types: double

#### CenterFrequency — Center frequency

2.45e9 | nonnegative scalar | row vector of nonnegative values

Center frequency, specified as a nonnegative scalar or a row vector of nonnegative values. Units are in Hz. The valid range of values for this property depends on the RF daughter card of the USRP device.

To change the center frequency, specify the value according to these conditions.

- For a single channel (SISO), specify the value for the center frequency as a nonnegative scalar.
- For multiple channels (MIMO) that use the same center frequency, specify the center frequency as a nonnegative scalar. The center frequency is set by scalar expansion.
- For multiple channels (MIMO) that use different center frequencies, specify the values in a row vector (for example, [70e6 100e6]). The  $i$ th element of the vector is applied to the  $i$ th channel specified by `ChannelMapping`.

---

#### Note

- The center frequency for B210 with MIMO must be a scalar. You cannot specify the frequencies as a vector.
  - The channels corresponding to the same RF daughterboard of N310 must have same center frequency value as each other.
-

For any of these conditions, the `IsTwinRXDaughterboard` property is set to 0 (`false`).

When the `IsTwinRXDaughterboard` property is 1 (`true`), specify the center frequency according to these conditions.

- To tune all channels to the same frequency, specify the center frequency as a scalar or row vector of the same values and the `EnableTwinRXPhaseSynchronization` property as 1 (`true`).
- To tune channels to different frequencies, specify the center frequency as a row vector. Each value in the row vector specifies the frequency of the corresponding channel. Set `EnableTwinRXPhaseSynchronization` property to 0 (`false`).

---

**Note** When `IsTwinRXDaughterboard` and `EnableTwinRXPhaseSynchronization` are both set to 1 (`true`), the LO source present on channel 1 is the master source to drive other LOs of the TwinRX daughterboard channels. In this case, the `CenterFrequency` property value must be the same for all channels of the TwinRX daughterboard.

---

For more information, see `EnableTwinRXPhaseSynchronization`.

**Tunable:** Yes

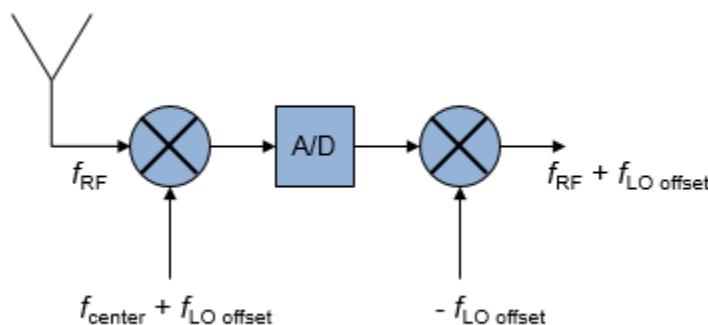
Data Types: double

#### **LocalOscillatorOffset — Local oscillator (LO) offset frequency**

0 | scalar | row vector

LO offset frequency, specified as a scalar or row vector. Units are in Hz. The valid range of this property depends on the RF daughterboard of the USRP device.

The LO offset does not affect the received center frequency. However, it does affect the intermediate center frequency in the USRP hardware, as shown in this diagram.



In this diagram:

- $f_{RF}$  is the received RF frequency.
- $f_{center}$  is the center frequency specified by the System object.

- $f_{\text{LO offset}}$  is the LO offset frequency.
- Ideally,  $f_{\text{RF}} - f_{\text{center}} = 0$ .

To move the center frequency away from interference or harmonics generated by the USRP hardware, use this property.

To change the LO offset, specify the value according to these conditions.

- For a single channel (SISO), specify the LO offset as a scalar.
- For multiple channels (MIMO), the LO offset must be zero. This restriction is due to a UHD limitation. In this case, you can specify the LO offset as scalar (0) or as a vector ([0 0]).

**Tunable:** Yes

Data Types: double

**Gain — Overall gain for USRP hardware receiver data path**

8 (default) | scalar | row vector

Overall gain for the USRP hardware receiver data path, including analog and digital components, specified as a scalar or row vector in dB. The valid range of this property depends on the RF daughterboard of the USRP device.

To change the gain, specify the value according to these conditions.

- For a single channel (SISO), specify the gain as a scalar.
- For multiple channels (MIMO) that use the same gain value, specify the gain as a scalar. The gain is set by scalar expansion.
- For multiple channels (MIMO) that use different gains, specify the values in a row vector (for example, [32 30]). The  $i$ th element of the vector is applied to the  $i$ th channel specified by `ChannelMapping`.

**Tunable:** Yes

Data Types: double

**PPSSource — Parts per second (PPS) signal source**

'Internal' (default) | 'External'

Parts per second (PPS) signal source, specified as one of these values.

- 'Internal' — Use the internal PPS signal of the USRP radio.
- 'External' — Use the PPS signal from an external signal generator.

To synchronize the time for all channels of the bundled radios, provide a common external PPS signal to all of the bundled radios and set this property to 'External'.

Data Types: char

**ClockSource — Clock source**

'Internal' (default) | 'External'

Clock source, specified as one of these values.

- Internal — Use the internal clock signal of the USRP radio.

- **External** — Use the 10-MHz clock signal from an external clock generator.

For B-series radios, the external clock port is labeled *10 MHz*. For N3xx, N2xx, USRP2, and X-series radios, the external clock port is labeled *REF IN*.

To synchronize the frequency for all channels of the bundled radios, provide a common external 10-MHz clock signal to all of the bundled radios and set this property to 'External'.

Data Types: char

#### **MasterClockRate — Master clock rate**

positive scalar

Master clock rate, specified as a positive scalar in Hz. The master clock rate is the A/D and D/A clock rate. The valid range of values for this property depends on the connected radio platform.

Platform Property Value	MasterClockRate Property Value (in Hz)
'N200/N210/USRP2'	100e6 (read-only)
'N300' or 'N310'	122.88e6, 125e6(default), or 153.6e6
'N320/N321'	200e6(default), 245.76e6, or 250e6
'B200' or 'B210'	Value in the range from 5e6 to 56e6. When using B210 with multiple channels, the clock rate must be less than or equal to 30.72e6. This restriction is a hardware limitation for the B210 radios when you use two-channel operations. The default value is 32e6.
'X300' or 'X310'	184.32e6 or 200e6(default)

#### **Dependencies**

To enable this property, set Platform to 'N300', 'N310', 'N320/N321', 'B200', 'B210', 'X300', or 'X310'.

Data Types: double

#### **DecimationFactor — Decimation factor for SDRu receiver**

512 (default) | integer from 1 to 1024

Decimation factor for the SDRu receiver, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

DecimationFact or Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
1	Valid	Not valid	Valid	Not valid when connected with TwinRX daughterboard

DecimationFactor or Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
2	Valid	Acceptable when you use only the <code>int8</code> transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Valid

The radio uses the decimation factor when it downconverts the intermediate frequency (IF) signal to a complex baseband signal.

Data Types: double

#### TransportDataType — Transport data type

'int16' (default) | 'int8'

Transport data type, specified as:

- 'int16' — Uses 16-bit transport to achieve higher precision.
- 'int8' — Uses 8-bit transport to achieve approximately two times faster transport data rate. The quantization step is 256 times larger than 16-bit transport.

The default transport data rate data type assigns the first 16 bits to the in-phase component and latter 16 bits to the quadrature component, resulting in 32 bits for each complex sample of transport data.

Data Types: char | string

#### OutputDataType — Data type of output signal

'Same as transport data type' (default) | 'double' | 'single'

Data type of the output signal, specified as one of these values.

- 'Same as transport data type' — The output data type is the same as the transport data type: either `int8` or `int16`.
  - When the transport data type is `int8`, the output values are raw 8-bit I and Q samples from the board in the range [-128, 127].
  - When the transport data type is '`int16`', the output values are raw 16-bit I and Q samples from the board in the range [-32,768, 32,767].

- 'single' — Single-precision floating point values scaled to the range of [-1, 1].
- 'double' — Double-precision floating point values scaled to the range of [-1, 1].

Data Types: char | string

#### **SamplesPerFrame — Number of samples per frame**

362 (default) | positive integer

Number of samples per frame of the output signal, specified as a positive integer. This value optimally utilizes the underlying Ethernet packets, which have a size of 1500 8-bit bytes.

Data Types: double

#### **EnableBurstMode — Option to enable burst mode**

0 or false (default) | 1 or true

Option to enable burst mode, specified as a numeric or logical value of 1 (true) or 0 (false). To produce a set of contiguous frames without an overrun or underrun to the radio, set this property to 1 (true). Enabling burst mode helps you simulate models that cannot run in real time.

When burst mode is enabled, specify the desired amount of contiguous data by using the NumFramesInBurst property. For more information, see "Detect Underruns and Overruns" on page 4-22.

Data Types: logical

#### **NumFramesInBurst — Number of frames in a contiguous burst**

100 (default) | nonnegative integer

Number of frames in a contiguous burst, specified as a nonnegative integer.

#### **Dependencies**

To enable this property, set EnableBurstMode to 1 (true).

Data Types: double

## **Usage**

### **Syntax**

```
data = rx()
[data,dataLen] = rx()
[data,dataLen,overrun] = rx()
[data,dataLen,overrun] = rx(fc,offset,gain)
```

#### **Description**

`data = rx()` receives data from a USRP device associated with the `comm.SDRuReceiver` System object, `rx`.

`[data,dataLen] = rx()` also returns `dataLen`, which indicates whether the object receives valid data from the radio hardware.

[`data`,`dataLen`,`overrun`] = `rx()` also returns an integer value that indicates data discontinuity in addition to the above syntaxes. If `overrun` is equal to or greater than 1, then `data` does not represent contiguous data.

[`data`,`dataLen`,`overrun`] = `rx(fc,offset,gain)` receives data from a USRP device associated with the `comm.SDRuReceiver` System object, `rx`. for given input center frequency, `fc`. The input `offset` is the local oscillator offset. `gain` is overall gain for USRP hardware receiver data path.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

## Input Arguments

### **fc — Center frequency**

2.45e9 (default) | nonnegative scalar | row vector of nonnegative values

Center frequency, specified as a nonnegative scalar or a row vector of nonnegative values. Units are in Hz. The valid range of values for this argument depends on the RF daughter card of the USRP device.

### **offset — Local oscillator offset**

0 (default) | scalar | row vector

LO offset frequency, specified as a scalar or row vector. Units are in Hz. The valid range of this argument depends on the RF daughterboard of the USRP device. For multiple channels (MIMO), the LO offset must be zero only. This restriction is due to a UHD limitation. In this case, you can specify the LO offset for multiple channels as a scalar (0) or as a vector ([0 0]).

### **gain — Overall gain**

8 (default) | scalar | row vector

Overall gain for the USRP hardware receiver data path, including analog and digital components, specified as a scalar or row vector in dB.

## Output Arguments

### **data — Output signal**

complex column vector | complex matrix

Output signal, returned as a column vector or matrix. For a single channel radio, this output is a column vector. For a multichannel radio, this output is a matrix. Each column in this matrix corresponds to a channel of complex data received on one channel.

Data Types: `int16` | `single` | `double`

### **dataLen — Data length and indication**

nonnegative integer

Data length and indication when valid data is present, returned as a nonnegative integer. When `dataLen` is 0, no data is present. If `dataLen` > 0, data is present, and you can use this data length to qualify the execution of part of the code. For more information, see “Apply Conditional Execution” on page 4-27.

At the time of radio initialization, no valid data exists. In this case, the output data is set to all zeros and `dataLen` is set to 0. When the host receives enough data from radio, the System object outputs valid data and `dataLen` is set to a positive integer.

#### **overrun — Data discontinuity flag**

integer

Data discontinuity flag, returned as an integer.

- When flag value is 0 — No overrun detected.
- When flag value is  $\geq 1$  — overrun detected. The output data does not represent contiguous data from the USRP radio to the host.

Although the reported value does not represent the actual number of packets dropped, as this value increases, the farther your execution of the object is from achieving real-time performance. You can use this value as a diagnostic tool to determine real-time execution of the object. For more information, see “Detect Underruns and Overruns” on page 4-22.

## **Object Functions**

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### **Specific to comm.SDRuReceiver**

info USRP radio information

### **Common to All System Objects**

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

## **Examples**

### **Receive Signals with B210 Radio and SDRu Receiver System Object**

Configure a B210 radio with serial number set to '31B92DD'. Set the radio to receive at 2.5 GHz, with the decimation factor of 256.

Create a SDRu Receiver System object to use for data reception.

```
rx = comm.SDRuReceiver( ...
    'Platform','B210',...
    'SerialNum','31B92DD',...
    'CenterFrequency',2.5e9,...
    'MasterClockRate',32e6,...
    'DecimationFactor',256);
```

Save the valid data using the `dsp.SignalSink` System object.

```
rxLog = dsp.SignalSink;
for counter = 1:20
    [data, len] = rx();
    if len>0
        rxLog(data);
    end
end
release(rx)
release(rxLog)
```

## Get Radio Information for Multi-Channel Radio

Create an SDRu receiver System object for a multi-channel radio configuration.

```
radio = comm.SDRuReceiver('Platform','X300','IPAddress','192.168.60.2');
radio.ChannelMapping = [1 2];
radio.CenterFrequency = [1.2 1.3]*1e9;
radio.Gain = [5 6];
```

Call the info method.

```
info(radio)

ans = struct with fields:
    Mboard: 'X300'
    RXSubdev: {'UBX RX' 'UBX RX'}
    TXSubdev: {'UBX TX' 'UBX TX'}
    MinimumCenterFrequency: [-70000000 -70000000]
    MaximumCenterFrequency: [6.0800e+09 6.0800e+09]
    MinimumGain: [0 0]
    MaximumGain: [37.5000 37.5000]
    GainStep: [0.5000 0.5000]
    CenterFrequency: [1.2000e+09 1.3000e+09]
    LocalOscillatorOffset: 0
    Gain: [5 6]
    MasterClockRate: 200000000
    DecimationFactor: 512
    BasebandSampleRate: 390625
```

## Capture and Save Receive Signals to Baseband File Writer

Configure a B210 radio with serial number set to '31B92DD'. Set the radio to receive at 1 GHz with an decimation factor of 512 and master clock rate of 56 MHz.

Create a SDRu Receiver System object to use for data reception. Calculate the baseband sample rate from master clock rate and decimation factor.

```
rx = comm.SDRuReceiver(...
    'Platform','B210',...
    'SerialNum','31B92DD',...
    'CenterFrequency',1e9,...
    'MasterClockRate', 56e6,...
```

```
'DecimationFactor',512);
sampleRate = rx.MasterClockRate/rx.DecimationFactor; % Calculate baseband sample rate
```

Create a baseband file writer object having center frequency of 1 GHz.

```
rxWriter = comm.BasebandFileWriter('b210_capture.bb', ...
    sampleRate, rx.CenterFrequency);
```

Write the valid baseband data to file 'b210\_capture.bb'.

```
for counter = 1:2000
    [data, len] = rx();
    if len>0
        rxWriter(data);
    end
end
```

Display information about received signal. Release the System objects.

```
info(rxWriter);
release(rx);
release(rxWriter);
```

## Detect Lost Samples using SDRu Receiver System Object

Configure a B210 radio with serial number set to '31B92DD'. Set the radio to receive at 2.5 GHz with an decimation factor of 125, output data type as 'double' and master clock rate of 56 MHz.

Create a USRP radio receiver System object to use for data reception.

```
rx = comm.SDRuReceiver('Platform','B210', ...
    'SerialNum','31B92DD', ...
    'CenterFrequency',2.5e9, ...
    'MasterClockRate',56e6, ...
    'DecimationFactor',125, ...
    'OutputDataType','double');
```

Capture signal data using comm.DPSKDemodulator System object.

```
demodulator = comm.DPSKDemodulator('BitOutput',true);
```

Inside a **for** loop, receive the data using the rx System object and return **overrun** as an output argument. Display the messages when receiver indicates **overrun** with data loss.

```
for frame = 1:2000
    [data, len, overrun] = rx();
    demodulator(data);
    if len>0
        if overrun~=0
            msg = ['Overrun detected in frame # ', int2str(frame)];
        end
    end
end
release(rx)
```

With SRDu receiver System objects, the `overrun` output indicates data loss. This output is a useful diagnostic tool for determining real-time operation of the System object.

### Burst-Mode Buffering to Overcome Overruns at Receiver

Configure a B210 radio with serial number set to '31B92DD'. Set the radio to receive at 2.5 GHz with a decimation factor of 125 and master clock rate of 56 MHz. Enable burst-mode buffering to overcome overrruns. Set number of frames in a burst to 20 and samples per frame to 37500.

Create a SDRu receiver System object to use for data reception.

```
rx = comm.SDRuReceiver(...
    'Platform', 'B210', ...
    'SerialNum', '31B92DD', ...
    'CenterFrequency', 2.5e9, ...
    'MasterClockRate', 56e6, ...
    'DecimationFactor', 125, ...
    'OutputDataType', 'double');

rx.EnableBurstMode = true;
rx.NumFramesInBurst = 20;
rx.SamplesPerFrame = 37500;
```

Capture signal data using `comm.DPSKDemodulaor` System object.

```
demodulator = comm.DPSKDemodulator('BitOutput', true);
```

Inside a for loop, transmit the data using the `rx` System object and return `overrun` as an output argument. Display the messages when receiver indicates overrun with data loss.

```
numFrames = 100;
for frame = 1:numFrames
    [data, len, overrun] = rx();
    if len>0
        demodulator(data);
        if (overrun)
            msg = ['Overrun detected in frame # ', int2str(frame)];
            disp(msg);
        end
    end
end

Overrun detected in frame # 1
Overrun detected in frame # 21
Overrun detected in frame # 41
Overrun detected in frame # 61
Overrun detected in frame # 81

release(rx)
```

Overruns are indicated at the start frame of transmission for each burst. With burst mode enabled, an overrun occurs in between bursts as the streaming resumes, because it is not possible to get continuous data when starting and stopping streaming.

## Receive Phase Synchronized Signals using TwinRX

Receive phase synchronized signals using the TwinRX daughterboard. The sinusoidal signals are transmitted with an N210 radio and reception is carried out on an X300 radio with two TwinRX daughterboards. The example requires two MATLAB sessions running on the host computer.

In the first MATLAB session, configure an N210 radio with IP address set to '192.168.10.2'. Set the radio to transmit at 2.45 GHz, an interpolation factor of 100, and a master clock rate of 100 MHz. Set a gain of 8 dB and transport data type of 'int16'.

```
tx = comm.SDRuTransmitter(...
    'Platform', 'N200/N210/USRP2', ...
    'IPAddress', '192.168.10.2', ...
    'MasterClockRate', 100e6, ...
    'InterpolationFactor', 100, ...
    'Gain', 8, ...
    'CenterFrequency', 2.45e9, ...
    'TransportDataType', 'int16');
```

Generate a sine wave of 30 kHz for transmission. The sample rate is calculated from the master clock rate and interpolation factor specified for an N210 radio System object configuration. Set the output data type of the sine wave as 'double'.

```
sinewave = dsp.SineWave(1,30e3);
sinewave.SampleRate = 100e6/100;
sinewave.SamplesPerFrame = 5e4;
sinewave.OutputDataType = 'double';
sinewave.ComplexOutput = true;
data = step(sinewave);
```

Set the frame duration for the sine wave to transmit based on the samples per frame and sample rate. Create time scope and frequency scope System objects to display time-domain and frequency-domain signals, respectively. Display a message when transmission starts.

```
frameDuration = (sinewave.SamplesPerFrame)/(sinewave.SampleRate);
time = 0;
timeScope = timescope('TimeSpanSource', 'Property', 'TimeSpan', 4/30e3, ...
    'SampleRate', 100e6/100);
spectrumScope = dsp.SpectrumAnalyzer('SampleRate', sinewave.SampleRate);
disp("Transmission Started");
timeScope(data);
spectrumScope(data);
```

Inside a `while` loop, transmit the sine wave using the `tx` System object. Display a message when transmission is complete. Release the radio System object.

```
while time<30
    tx(data);
    time = time+frameDuration;
end
disp("Transmission Stopped");
release(tx);
```

In the second MATLAB session, configure an X300 radio with IP address set to '192.168.20.2'. Set the radio to receive at 2.45 GHz with an decimation factor of 200 and a master clock rate of 200 MHz. Enable the TwinRX daughterboard and the TwinRX phase synchronization capability to receive phase synchronized signals. Set the channel mapping to [1 2 3 4]. Connect the power splitter from an N210 transmitter to four receiver channels of X300 radio for calibration.

```
rx = comm.SDRuReceiver(...  
    'Platform','X300', ...  
    'IPAddress','192.168.20.2', ...  
    'OutputDataType','double', ...  
    'IsTwinRXDaughterboard',true, ...  
    'EnableTwinRXPhaseSynchronization',true, ...  
    'ChannelMapping',[1 2 3 4]  
    'MasterClockRate',200e6, ...  
    'DecimationFactor',200, ...  
    'Gain',35, ...  
    'CenterFrequency',2.45e9, ...  
    'SamplesPerFrame',4000);
```

Set the frame duration for the sine wave to receive based on samples per frame and sample rate. Create the time scope and frequency scope System objects to display time-domain and frequency-domain signals, respectively. Display the message when reception starts.

```
frameduration = (rx.SamplesPerFrame)/(200e6/200);  
time = 0;  
timeScope = timescope('TimeSpanSource','Property','TimeSpan', ...  
    4/30e3,'SampleRate',200e6/200);  
spectrumScope = dsp.SpectrumAnalyzer('SampleRate',200e6/200);  
spectrumScope.ReducePlotRate = true;  
disp("Reception Started");
```

Inside a while loop, receive the sine wave using the rx System object. Normalize the signal with respect to amplitude for each receive channel. Compute the fast fourier transform (FFT) of each normalized signal. Calculate the phase difference between channel 1 and channel 2, channel 1 and channel 3, and channel 1 and channel 4.

```
while time < 10  
    [data,len] = step(rx);  
    if len > 0  
        amp(1) = max(abs(data(:,1)));  
        amp(2) = max(abs(data(:,2)));  
        amp(3) = max(abs(data(:,3)));  
        amp(4) = max(abs(data(:,4)));  
        maxAmp = max(amp);  
        if any(~amp)  
            NormalizedData = data;  
        else  
            NormalizedData(:,1) = maxAmp/amp(1)*data(:,1);  
            NormalizedData(:,2) = maxAmp/amp(2)*data(:,2);  
            NormalizedData(:,3) = maxAmp/amp(3)*data(:,3);  
            NormalizedData(:,4) = maxAmp/amp(4)*data(:,4);  
        end  
        freq0fFirst = fft(NormalizedData(:,1));  
        freq0fSecond = fft(NormalizedData(:,2));  
        freq0fThird = fft(NormalizedData(:,3));  
        freq0fFourth = fft(NormalizedData(:,4));  
        angle1 = rad2deg(angle(max(freq0fFirst)/max(freq0fSecond)));  
        angle2 = rad2deg(angle(max(freq0fFirst)/max(freq0fThird)));  
        angle3 = rad2deg(angle(max(freq0fFirst)/max(freq0fFourth)));  
        timeScope([real(NormalizedData),imag(NormalizedData)]);  
        spectrumScope(NormalizedData);  
    end  
    time = time + frameduration;  
end
```

Display the calculated phase difference between channel 1 and each of the other channels of TwinRX daughterboard.

```
disp([' Phase difference between channel 1 and 2: ', num2str(angle1)]);
disp([' Phase difference between channel 1 and 3: ', num2str(angle2)]);
disp([' Phase difference between channel 1 and 4: ', num2str(angle3)]);
disp("Reception Ended");
release(timeScope);
release(spectrumScope);
release(rx);

Phase difference between channel 1 and 2: -98.511
Phase difference between channel 1 and 3: -161.599
Phase difference between channel 1 and 4: -86.680
Reception Ended
```

## More About

### Single and Multiple Channel Output

- N200, N210, USRP2, and B200 radios support a single channel that you can use to:
  - Send data with the `comm.SDRuTransmitter` System object. The `comm.SDRuTransmitter` System object receives a column vector signal of fixed length.
  - Receive data with the `comm.SDRuReceiver` System object. The `comm.SDRuReceiver` System object outputs a column vector signal of fixed length.
- B210, X300, X310, N300, N320 and N321 radios support two channels that you can use to transmit and receive data with System objects. You can use both channels or only a single channel (either channel 1 or 2).
  - The `comm.SDRuTransmitter` System object receives a matrix signal, where each column is a channel of data of fixed length.
  - The `comm.SDRuReceiver` System object outputs a matrix signal, where each column is a channel of data of fixed length.

---

**Note** When two TwinRX daughterboards are connected to X300 or X310 radio, the radio supports up to four channel reception.

---

- N310 radio support four channels that you can use to transmit and receive data with System objects. You can use up to four channels.
  - The `comm.SDRuTransmitter` System object receives a matrix signal, where each column is a channel of data of fixed length.
  - The `comm.SDRuReceiver` System object outputs a matrix signal, where each column is a channel of data of fixed length.

You can set the `CenterFrequency`, `LocalOscillatorOffset`, and `Gain` properties independently for each channel. Alternatively, you can apply the same setting to all channels. All other System object property values apply to all channels.

For more information, see “Single Channel Input and Output Operations” on page 4-8 and “Multiple Channel Input and Output Operations” on page 4-14.

## Compatibility Considerations

### X3xx series radios no longer support 120 MHz master clock rate

*Errors starting in R2020a*

Beginning with Ettus Research UHD version 003.014.000.000, X3xx series radios do not support a master clock rate value of 120 MHz. Consequently, starting in R2020a, which supports UHD version 003.015.000.000, Communications Toolbox Support Package for USRP Radio does not support a master clock rate value of 120 MHz for X3xx series radios.

For the `comm.SDRuTransmitter` and `comm.SDRuReceiver` System objects, when you specify an X3xx series radio for the `Platform` property, you can no longer set the `MasterClockRate` property to `120e6`.

## See Also

### Objects

`comm.SDRuReceiver`

### Blocks

`SDRu Receiver`

### Topics

“Single Channel Input and Output Operations” on page 4-8

“Multiple Channel Input and Output Operations” on page 4-14

“Apply Conditional Execution” on page 4-27

“Detect Underruns and Overruns” on page 4-22

“Burst-Mode Buffering” on page 5-2

## Introduced in R2011b

# info

**Package:** comm

USRP radio information

## Syntax

```
radioSettings = info(radio)
```

## Description

`radioSettings = info(radio)` returns a structure array containing the current radio settings for the USRP radio hardware associated with the SDRu System object `radio`.

## Examples

### Get B210 Radio Information

Use the `info` object function to get information from the connected B210 radio. The actual values used in the radio are shown by `info` and can vary slightly from the values specified in the object.

```
radio = comm.SDRuTransmitter('Platform','B210','SerialNum','31B92DD');
radio.CenterFrequency = 912.3456e6;
radio.LocalOscillatorOffset = 1000;
radio.Gain = 8.3;
radio.MasterClockRate = 10.56789e6;
radio.InterpolationFactor = 510;
info(radio)

ans = struct with fields:
    Mboard: 'B210'
    RXSubdev: 'FE-RX2'
    TXSubdev: 'FE-TX2'
    MinimumCenterFrequency: 4.4716e+07
    MaximumCenterFrequency: 6.0053e+09
    MinimumGain: 0
    MaximumGain: 89.7500
    GainStep: 0.2500
    CenterFrequency: 9.1235e+08
    LocalOscillatorOffset: -999.7189
    Gain: 8.2500
    MasterClockRate: 1.0568e+07
    InterpolationFactor: 512
    BasebandSampleRate: 2.0640e+04
```

### Get Radio Information for Multi-Channel Radio

Create an SDRu receiver System object for a multi-channel radio configuration.

```
radio = comm.SDRuReceiver('Platform','X300','IPAddress','192.168.60.2');
radio.ChannelMapping = [1 2];
radio.CenterFrequency = [1.2 1.3]*1e9;
radio.Gain = [5 6];
```

Call the info method.

```
info(radio)
```

```
ans = struct with fields:
    Mboard: 'X300'
    RXSubdev: {'UBX RX' 'UBX RX'}
    TXSubdev: {'UBX TX' 'UBX TX'}
    MinimumCenterFrequency: [-70000000 -70000000]
    MaximumCenterFrequency: [6.0800e+09 6.0800e+09]
    MinimumGain: [0 0]
    MaximumGain: [37.5000 37.5000]
    GainStep: [0.5000 0.5000]
    CenterFrequency: [1.2000e+09 1.3000e+09]
    LocalOscillatorOffset: 0
    Gain: [5 6]
    MasterClockRate: 200000000
    DecimationFactor: 512
    BasebandSampleRate: 390625
```

## Input Arguments

### radio — USRP radio

comm.SDRuTransmitter System object | comm.SDRuReceiver System object

USRP radio, specified as a comm.SDRuTransmitter or comm.SDRuReceiver System object. This radio must be connected to the host computer.

## Output Arguments

### radioSettings — Synchronized radio settings information

structure array

Synchronized radio settings information between the System object and the associated USRP radio hardware, returned as a structure array.

If **radioSettings** has settings information, the returned radio information varies depending on the type of the USRP device.

If **radioSettings** has no settings information or it is not associated with a USRP radio attached to the host, the structure returned is empty or has its **Status** field set to 'No device found' in **findsdru** function.

## Tips

- The actual radio computed value and your specified setting can differ. To confirm that the actual radio computed value is close enough to your specified setting, use this **info** function with the System object at the input **radio**.

## See Also

### Objects

`comm.SDRuReceiver` | `comm.SDRuTransmitter`

### Blocks

`SDRu Receiver` | `SDRu Transmitter`

## Introduced in R2011b

# comm.SDRuTransmitter

**Package:** comm

Send data to USRP device

## Description

The SDRuTransmitter System object sends data to a Universal Software Radio Peripheral (USRP) hardware device, enabling simulation and development for various software-defined radio applications. The object enables communication with a USRP board on the same Ethernet subnetwork or a USRP board via a USB connection. You can write a MATLAB application that uses the System object, or you can generate code for the System object without connecting to a USRP radio.

This object accepts a column vector or matrix input signal from MATLAB and transmits signal and control data to a USRP board using the Universal Hardware Driver (UHD) from Ettus Research. The System object is a sink that sends the data it receives to a USRP board.

To send data from a USRP device:

- 1 Create the `comm.SDRuTransmitter` object and set its properties.
- 2 Call the object as if it were a function.

To learn more about how System objects work, see [What Are System Objects?](#).

## Creation

### Syntax

```
tx = comm.SDRuTransmitter  
tx = comm.SDRuTransmitter(address)  
tx = comm.SDRuTransmitter(___,Name,Value)
```

### Description

`tx = comm.SDRuTransmitter` creates a default SDRu transmitter System object.

`tx = comm.SDRuTransmitter(address)` sets the `IPAddress` property to address of the connected USRP device.

`tx = comm.SDRuTransmitter(___,Name,Value)` sets “Properties” on page 8-24 using one or more name-value pairs in addition to any input argument combination from previous syntaxes. Enclose each property name in quotes. For example, `'CenterFrequency', 5e6` specifies the center frequency as 5 MHz.

### Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

**Platform — Model number of radio**

'N200/N210/USRP2' (default) | 'N300' | 'N310' | 'N320/N321' | 'B200' | 'B210' | 'X300' | 'X310'

Model number of the radio, specified as one of these values.

- 'N200/N210/USRP2'
- 'N300'
- 'N310'
- 'N320/N321'
- 'B200'
- 'B210'
- 'X300'
- 'X310'

Data Types: char | string

**IPAddress — IP address of USRP device**

'192.168.10.2' (default) | dotted-quad character vector | dotted-quad string scalar

IP address of the USRP device, specified as a dotted-quad character vector or dotted-quad string scalar. When you specify more than one IP address, you must separate each address by commas or spaces.

This value must match the physical IP address of the radio hardware assigned during hardware setup. For more information, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If you configure the radio hardware with an IP address other than the default, update this property accordingly.

To find the logical network location of all connected USRP radios, use the **findsdr** function.

Example: '192.168.10.2, 192.168.10.5' or '192.168.10.2 192.168.10.5' specifies IP addresses for two devices.

**Dependencies**

To enable this property, set Platform to 'N200/N210/USRP2', 'N300', 'N310', 'N320/N321', 'X300', or 'X310'.

Data Types: char | string

**SerialNum — Serial number of radio**

'' (default) | character vector | string scalar

Serial number of the radio hardware, specified as a character vector or string scalar.

This property must match the serial number of the radio hardware assigned during hardware setup. For more information, see “Guided USRP Radio Support Package Hardware Setup” on page 1-5. If

you configure the radio hardware with a serial number other than the default, update this property accordingly.

#### Dependencies

To enable this property, set **Platform** to 'B200' or 'B210'.

Data Types: char | string

#### ChannelMapping — Channel mapping for radio or bundled radios

1 (default) | nonnegative scalar | row vector of nonnegative values

Channel mapping for the radio or bundled radios, specified as a nonnegative scalar or a row vector of nonnegative values. This table shows the valid values for various radio platforms.

Platform Property Value	ChannelMapping Property Value
'N200/N210/USRP2'	1-by- $N$ row vector, where $N$ is the number of IP addresses included in the <b>IPAddress</b> property
'N300'	1, 2, or [1 2]
'N310'	1-, 2-, 3-, or 4-element row vector of channel numbers from the set {1, 2, 3, 4}
'N320/N321'	1, 2, or [1 2]
'B200'	1
'B210'	1, 2, or [1 2]
'X300' or 'X310'	<ul style="list-style-type: none"><li>When the <b>IPAddress</b> property includes one IP address, specify this value as 1, 2, or [1 2].</li><li>When <b>IPAddress</b> includes <math>N</math> IP addresses, specify this value as 1-by-<math>2N</math> row vector, where <math>N</math> is the number of IP addresses included in <b>IPAddress</b>.</li></ul>

When **IPAddress** includes multiple IP addresses, the channels defined by **ChannelMapping** are ordered first by the order in which the IP addresses appear in the list and then by the channel order within the same radio.

Example: If **Platform** is 'X300' and **IPAddress** is '192.168.20.2, 192.168.10.3', then **ChannelMapping** must be [1 2 3 4]. Channels 1, 2, 3, and 4 of the bundled radio refer to channels 1 and 2 of the radio with IP address 192.168.20.2 and channels 1 and 2 of the radio with IP address 192.168.10.3.

Data Types: double

#### CenterFrequency — Center frequency

2.45e9 | nonnegative scalar | row vector of nonnegative values

Center frequency, specified as a nonnegative scalar or a row vector of nonnegative values. Units are in Hz. The valid range of values for this property depends on the RF daughter card of the USRP device.

To change the center frequency, specify the value according to these conditions.

- For a single channel (SISO), specify the value for the center frequency as a nonnegative scalar.
- For multiple channels (MIMO) that use the same center frequency, specify the center frequency as a nonnegative scalar. The center frequency is set by scalar expansion.
- For multiple channels (MIMO) that use different center frequencies, specify the values in a row vector (for example, [70e6 100e6]). The  $i$ th element of the vector is applied to the  $i$ th channel specified by the ChannelMapping property.

**Note**

- The center frequency for B210 with MIMO must be a scalar. You cannot specify the frequencies as a vector.
- The channels corresponding to the same RF daughterboard of N310 must have the same center frequency value as each other.

**Tunable:** Yes

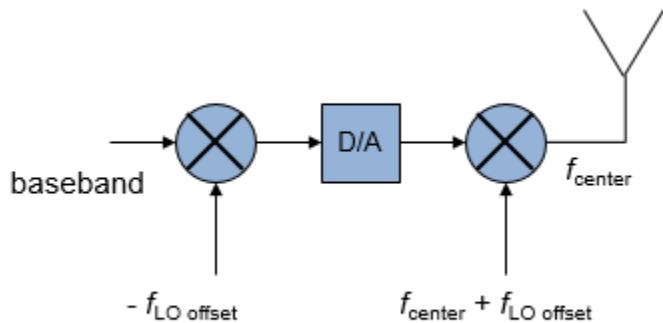
Data Types: double

**LocalOscillatorOffset — Local oscillator (LO) offset frequency**

0 | scalar | row vector

LO offset frequency, specified as a scalar or row vector. Units are in Hz. The valid range of this property depends on the RF daughterboard of the USRP device.

The LO offset does not affect the transmitted center frequency. However, it does affect the intermediate center frequency in the USRP hardware, as shown in this diagram.



In this diagram:

- $f_{\text{center}}$  is the center frequency specified by the System object.
- $f_{\text{LO offset}}$  is the local oscillator offset frequency.

To move the center frequency away from interference or harmonics generated by the USRP hardware, use this property.

To change the LO offset, specify the value according to these conditions.

- For a single channel (SISO), specify the LO offset as a scalar.
- For multiple channels (MIMO), the LO offset must be zero. This restriction is due to a UHD limitation. In this case, you can specify the LO offset as scalar (0) or as a vector ([0 0]).

**Tunable:** Yes

Data Types: double

**Gain — Overall gain for USRP hardware transmitter data path**

8 | scalar | row vector

Overall gain for the USRP hardware transmitter data path, including analog and digital components, specified as a scalar or row vector. Units are in dB. The valid range of this property depends on the RF daughterboard of the USRP device.

To change the gain, specify the value according to these conditions.

- For a single channel (SISO), specify the value for the gain as a scalar.
- For multiple channels (MIMO) that use the same gain value, specify the gain as a scalar. The gain is set by scalar expansion.
- For multiple channels (MIMO) that use different gains, specify the values in a row vector (for example, [32 30]). The *i*th element of the vector is applied to the *i*th channel specified by the **ChannelMapping** property.

**Tunable:** Yes

**PPSSource — Parts per second (PPS) signal source**

'Internal' (default) | 'External'

Parts per second (PPS) signal source, specified one of these values.

- 'Internal' — Use the internal PPS signal of the USRP radio.
- 'External' — Use the PPS signal from an external signal generator.

To synchronize the time for all channels of the bundled radios, provide a common external PPS signal to all of the bundled radios and set this property to 'External'.

Data Types: char

**ClockSource — Clock source**

'Internal' (default) | 'External'

Clock source, specified as one of these values.

- Internal — Use the internal clock signal of the USRP radio.
- External — Use the 10 MHz clock signal from an external clock generator.

For B-series radios, the external clock port is labeled *10 MHz*. For N3xx series, N2xx series, USRP2, and X-series radios, the external clock port is labeled *REF IN*.

To synchronize the frequency for all channels of the bundled radios, provide a common external 10 MHz clock signal to all of the bundled radios and set this property to 'External'.

Data Types: char

**MasterClockRate — Master clock rate**

positive scalar

Master clock rate, specified as a positive scalar. Units are in Hz. The master clock rate is the A/D and D/A clock rate. The valid range of values for this property depends on the connected radio platform.

Platform Property Value	MasterClockRate Property Value (in Hz)
'N200/N210/USRP2'	100e6 (read-only)
'N300' or 'N310'	122.88e6, 125e6(default), or 153.6e6
'N320/N321'	200e6(default), 245.76e6, or 250e6
'B200' or 'B210'	Value in the range from 5e6 to 56e6. When using B210 with multiple channels, the clock rate must be less than or equal to 30.72e6. This restriction is a hardware limitation for the B210 radios when you use two-channel operations. The default value is 32e6.
'X300' or 'X310'	184.32e6 or 200e6(default)

**Dependencies**

To enable this property, set Platform to 'B200', 'B210', 'N300', 'N310', 'N320/N321', 'X300', or 'X310'.

Data Types: double

**InterpolationFactor — Interpolation factor for SDRu transmitter**

512 (default) | integer from 1 to 1024

Interpolation factor for the SDRu transmitter, specified as an integer from 1 to 1024 with restrictions, based on the radio you use.

InterpolationFactor Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
1	Valid	Not valid	Valid	Valid
2	Valid	Acceptable when you use only the int8 transport data type	Valid	Valid
3	Valid	Not valid	Valid	Valid
Odd number from 4 to 128	Valid	Valid	Not valid	Valid
Even number from 4 to 128	Valid	Valid	Valid	Valid
Even number from 128 to 256	Valid	Valid	Valid	Valid
Multiple of 4 from 256 to 512	Valid	Valid	Valid	Valid

InterpolationFactor Property Value	B-Series	N2xx-Series	N3xx-Series	X-Series
Multiple of 8 from 512 to 1024	Not valid	Not valid	Valid	Not valid

The radio uses the interpolation factor when it upconverts the complex baseband signal to an intermediate frequency (IF) signal.

Data Types: double

#### **TransportDataType — Transport data type**

'int16' (default) | 'int8'

Transport data type, specified as one of these values:

- 'int16' — Use 16-bit transport to achieve higher precision.
- 'int8' — Use 8-bit transport to achieve an approximately two times faster transport data rate.  
The quantization step is 256 times larger than 16-bit transport.

The default transport data rate data type assigns the first 16 bits to the in-phase component and the latter 16 bits to the quadrature component, resulting in 32 bits for each complex sample of transport data.

Data Types: char | string

#### **EnableBurstMode — Option to enable burst mode**

0 or false (default) | 1 or true

Option to enable burst mode, specified as a numeric or logical value of 1 (true) or 0 (false). To produce a set of contiguous frames without an overrun or underrun to the radio, set this property to 1 (true). Enabling burst mode helps you simulate models that cannot run in real time.

When burst mode is enabled, specify the desired amount of contiguous data by using the NumFramesInBurst property. For more information, see "Detect Underruns and Overruns" on page 4-22.

Data Types: logical

#### **NumFramesInBurst — Number of frames in contiguous burst**

100 (default) | nonnegative integer

Number of frames in a contiguous burst, specified as a nonnegative integer.

#### **Dependencies**

To enable this property, set EnableBurstMode to 1 (true).

Data Types: double

## **Usage**

### **Syntax**

`tx(data)`

```
tx(data,fc)
underrun = tx(data,fc,offset,gain)
```

### Description

`tx(data)` sends data to a USRP device associated with the `comm.SDRuTransmitter` System object, `tx`.

`tx(data,fc)` sends data to a USRP device for the given input center frequency, `fc`.

`underrun = tx(data,fc,offset,gain)` returns an integer value that indicates data discontinuity for given input data, `data`, and center frequency, `fc`. The input `offset` is the local oscillator offset. `gain` is overall gain for USRP hardware transmitter data path.

---

**Note** Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

---

### Input Arguments

#### **input — Input signal**

complex column vector | complex matrix

Input signal sent to the USRP radio, specified as a complex column vector or complex matrix. The number of columns in the matrix depends on the number of channels in use, as specified by the `ChannelMapping` property. For a single channel radio, this input must be a column vector. For a multichannel radio, this input must be a matrix. Each column in this matrix corresponds to a channel of complex data sent on one channel.

The complex data type of the transmitted signal must be one of these data types:

- 16-bit signed integers — Complex values in the range of [-32768,32767]
- Single-precision floating point — Complex values in the range of [-1, 1]
- Double-precision floating point — Complex values in the range of [-1, 1]

Data Types: `double` | `single` | `int16`

#### **fc — Center frequency**

nonnegative scalar

Center frequency, specified as a nonnegative scalar. Units are in Hz. The valid range of values for this argument depends on the RF daughter card of the USRP device.

Data Types: `double`

#### **offset — LO offset**

scalar | row vector

LO offset, specified as a scalar or row vector. Units are in Hz. The valid range of values for this argument depends on the RF daughter card of the USRP device. For multiple channels (MIMO), the LO offset must be zero. This restriction is due to a UHD limitation. In this case, you can specify the LO offset as scalar (0) or as a vector ([0 0]).

Data Types: `double`

**gain — Overall gain**

scalar | row vector

Overall gain for the USRP hardware transmitter data path, including analog and digital components, specified as a scalar or row vector. Units are in dB.

Data Types: double

**Output Arguments****underrun — Data discontinuity flag**

integer

Data discontinuity flag, returned as an integer.

- When flag value is 0 — No underrun detected.
- When flag value is  $\geq 1$  — Underrun detected. The input data does not represent contiguous data from the host to the USRP radio.

Although the reported value does not represent the actual number of packets dropped, as this value increases, the farther your execution of the object is from achieving real-time performance. You can use this value as a diagnostic tool to determine real-time execution of the object. For more information, see "Detect Underruns and Overruns" on page 4-22.

## Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

### Specific to `comm.SDRuTransmitter`

`info` USRP radio information

### Common to All System Objects

`step` Run System object algorithm  
`release` Release resources and allow changes to System object property values and input characteristics  
`reset` Reset internal states of System object

## Examples

### Get B210 Radio Information

Use the `info` object function to get information from the connected B210 radio. The actual values used in the radio are shown by `info` and can vary slightly from the values specified in the object.

```
radio = comm.SDRuTransmitter('Platform','B210','SerialNum','31B92DD');
radio.CenterFrequency = 912.3456e6;
radio.LocalOscillatorOffset = 1000;
radio.Gain = 8.3;
radio.MasterClockRate = 10.56789e6;
```

```

radio.InterpolationFactor = 510;
info(radio)

ans = struct with fields:
    Mboard: 'B210'
    RXSubdev: 'FE-RX2'
    TXSubdev: 'FE-TX2'
    MinimumCenterFrequency: 4.4716e+07
    MaximumCenterFrequency: 6.0053e+09
    MinimumGain: 0
    MaximumGain: 89.7500
    GainStep: 0.2500
    CenterFrequency: 9.1235e+08
    LocalOscillatorOffset: -999.7189
    Gain: 8.2500
    MasterClockRate: 1.0568e+07
    InterpolationFactor: 512
    BasebandSampleRate: 2.0640e+04

```

## Transmit Signals with B210 Radio and SDRu Transmitter System Object

Configure a B210 radio with serial number set to '30F59A1'. Set the radio to transmit at 2.5 GHz with an interpolation factor of 256.

Create a SDRu Transmitter System object to use for data transmission.

```

tx = comm.SDRuTransmitter(
    'Platform','B210', ...
    'SerialNum','30F59A1', ...
    'CenterFrequency',2.5e9, ...
    'InterpolationFactor',256);

```

Create a DPSK modulator as the data source using `comm.DPSKModulator` System object. Inside a `for` loop, transmit the data using the `tx` System object.

```

mod = comm.DPSKModulator('BitInput',true);
for counter = 1:20
    data = randi([0 1],30,1);
    modSignal = mod(data);
    tx(modSignal);
end

```

## Detect Lost Samples using SDRuTransmitter System Object

Configure an B210 radio with serial number set to '30F59A1'. Set the radio to transmit at 2.5 GHz with an interpolation factor of 125 and master clock rate of 56 MHz.

Create a SDRu Transmitter System object to use for data transmission.

```

tx = comm.SDRuTransmitter(
    'Platform','B210', ...
    'SerialNum','30F59A1', ...

```

```
'CenterFrequency',2.5e9, ...
'InterpolationFactor',125, ...
'MasterClockRate', 56e6);
```

Create a DPSK modulator as the data source using `comm.DPSKModulator` System object.

```
modulator = comm.DPSKModulator('BitInput',true);
```

Inside a `for` loop, transmit the data using the `tx` System object and return `underrun` as an output argument. Display the messages when transmitter indicates `underrun` with data loss.

```
for frame = 1:20000
    data = randi([0 1], 30, 1);
    modSignal = modulator(data);
    underrun = tx(modSignal);
    if underrun~=0
        msg = ['Underrun detected in frame # ', int2str(frame)];
    end
end
release(tx)
```

With SRDu transmitter System objects, the `underrun` output indicates data loss. This output is a useful diagnostic tool for determining real-time operation of the System object.

### Burst-Mode Buffering to Overcome Underruns at Transmitter

Configure a B210 radio with serial number set to '`30F59A1`'. Set the radio to transmit at 2.5 GHz with an interpolation factor of 125 and master clock rate of 56 MHz. Enable burst-mode buffering to overcome underruns. Set number of frames in a burst to 20.

Create a SDRu Transmitter System object to use for data transmission.

```
tx = comm.SDRuTransmitter(
    'Platform','B210',...
    'SerialNum','30F59A1',...
    'CenterFrequency',2.5e9,...
    'InterpolationFactor',125,...
    'MasterClockRate', 56e6);
tx.EnableBurstMode = true;
tx.NumFramesInBurst = 20;
```

Create a DPSK modulator as the data source using `comm.DPSKModulator` System object.

```
modulator = comm.DPSKModulator('BitInput',true);
data = randi([0 1],37500,1);
modSignal = modulator(data);
```

Inside a `for` loop, transmit the data using the `tx` System object and return `underrun` as an output argument. Display the messages when transmitter indicates `underrun` with data loss.

```
numFrames = 100;
for frame = 1:numFrames
    underrun = tx(modSignal);
    if (underrun)
        msg = ['Underrun detected in frame # ', int2str(frame)];
```

```

    disp(msg)
end
no tx ack
Underrun detected in frame # 20
Underrun detected in frame # 40
Underrun detected in frame # 60
Underrun detected in frame # 80
Underrun detected in frame # 100

release(tx)

```

Underruns are indicated at the last frame of transmission for each burst. With burst mode enabled, an underrun occurs in between bursts as the streaming resumes, because it is not possible to get continuous data when starting and stopping streaming.

## More About

### Single and Multiple Channel Output

- N200, N210, USRP2, and B200 radios support a single channel that you can use to:
  - Send data with the `comm.SDRuTransmitter` System object. The `comm.SDRuTransmitter` System object receives a column vector signal of fixed length.
  - Receive data with the `comm.SDRuReceiver` System object. The `comm.SDRuReceiver` System object outputs a column vector signal of fixed length.
- B210, X300, X310, N300, N320 and N321 radios support two channels that you can use to transmit and receive data with System objects. You can use both channels or only a single channel (either channel 1 or 2).
  - The `comm.SDRuTransmitter` System object receives a matrix signal, where each column is a channel of data of fixed length.
  - The `comm.SDRuReceiver` System object outputs a matrix signal, where each column is a channel of data of fixed length.

---

**Note** When two TwinRX daughterboards are connected to X300 or X310 radio, the radio supports up to four channel reception.

- N310 radio support four channels that you can use to transmit and receive data with System objects. You can use up to four channels.
  - The `comm.SDRuTransmitter` System object receives a matrix signal, where each column is a channel of data of fixed length.
  - The `comm.SDRuReceiver` System object outputs a matrix signal, where each column is a channel of data of fixed length.

You can set the `CenterFrequency`, `LocalOscillatorOffset`, and `Gain` properties independently for each channel. Alternatively, you can apply the same setting to all channels. All other System object property values apply to all channels.

For more information, see “Single Channel Input and Output Operations” on page 4-8 and “Multiple Channel Input and Output Operations” on page 4-14.

## Compatibility Considerations

### X3xx series radios no longer support 120 MHz master clock rate

*Errors starting in R2020a*

Beginning with Ettus Research UHD version 003.014.000.000, X3xx series radios do not support a master clock rate value of 120 MHz. Consequently, starting in R2020a, which supports UHD version 003.015.000.000, Communications Toolbox Support Package for USRP Radio does not support a master clock rate value of 120 MHz for X3xx series radios.

For the `comm.SDRuTransmitter` and `comm.SDRuReceiver` System objects, when you specify an X3xx series radio for the `Platform` property, you can no longer set the `MasterClockRate` property to `120e6`.

## See Also

### Objects

`comm.SDRuTransmitter`

### Blocks

`SDRu Transmitter`

### Topics

“Single Channel Input and Output Operations” on page 4-8

“Multiple Channel Input and Output Operations” on page 4-14

“Apply Conditional Execution” on page 4-27

“Detect Underruns and Overruns” on page 4-22

“Burst-Mode Buffering” on page 5-2

## Introduced in R2011b

# Functions

---

## findsdru

Status of USRP radios connected to host computer

### Syntax

```
A = findsdru
A = findsdru(IPAddress)
A = findsdru(serialNum)
```

### Description

`A = findsdru` returns a structure that contains the model number, IP address, serial number, and status of each USRP radio that is connected to the host computer.

`A = findsdru(IPAddress)` returns information for only the USRP radio with the specified IP address.

`A = findsdru(serialNum)` returns information for only the USRP radio with the specified serial number.

### Examples

#### Report Status for All USRP Radios Connected to Host Computer

Before running this example, verify that your USRP radio is configured for host-radio communication by following the steps in “Guided USRP Radio Support Package Hardware Setup” on page 1-5.

Get the model number, IP address, serial number, and status of each USRP radio that is connected to the host computer.

```
A = findsdru
Checking radio connections...
A = struct with fields:
    Platform: 'B210'
    IPAddress: ''
    SerialNum: '31B92DD'
    Status: 'Success'
```

### Input Arguments

#### IPAddress — IP address of USRP radio

dotted-quad character vector | dotted-quad string scalar

IP address of the USRP radio that is connected to the host computer, specified as a dotted-quad character vector or dotted-quad string scalar.

Data Types: char | string

#### **serialNum — Serial number of USRP radio**

character vector | string scalar

Serial number of the USRP radio that is connected to the host computer, specified as a character vector or string scalar.

Data Types: char | string

## **Output Arguments**

### **A — Information about connected USRP devices**

structure

Information about connected USRP devices, returned as a structure containing these fields.

Fields	Description
<b>Platform</b>	Model number of the USRP device
<b>IPAddress</b>	IP address of the USRP device
<b>SerialNum</b>	Serial number of the USRP device
<b>Status</b>	Status information of the USRP device

The **Status** field of this structure displays one of the messages described in this table.

Status Value	Definition
<b>Success</b>	USRP device is available.
<b>No devices found</b>	No USRP devices found.
<b>Not compatible</b>	Device with incompatible firmware. To communicate with this device, you must update the firmware to the version returned by the function <code>getSDRDriverVersion</code> . For more information on firmware, see “USRP Radio Firmware Update” on page 1-53.

Status Value	Definition
<b>Not responding</b>	<p>Device is not responding because of any of these reasons.</p> <ul style="list-style-type: none"> <li>The device is not attached to the host computer. If you are having an Ethernet connectivity problem, see “Check Ethernet Configuration” on page 1-44.</li> <li>No device with the specified IP address exists.</li> <li>The subnet address of the host computer does not match the subnet address of the device. The subnet address is the third field of the IP address. Verify that the subnet value of the host and radio are the same. For example, your NIC has an IP address of 192.168.10.1, and the IP address of the USRP radio is 192.168.20.2. The IP addresses differ in the third octet. For more information on subnet addresses, see “Check Subnet Values on Host and Radio” on page 1-46.</li> </ul> <p><b>Note</b> For X3xx series radios, you will see this additional warning message that directs you to the hardware setup.</p> <p><b>Warning:</b> Device at '192.168.60.2' is not responding because subnet value of host network interface does not match with device subnet value. To configure host network interface <a href="#">click Here</a>.</p> <ul style="list-style-type: none"> <li>The specified serial number is invalid.</li> </ul>
<b>Busy</b>	Device is in use. The device is already owned by a block, a block dialog, or a System object.
<b>Unknown error</b>	Unknown problem.

## See Also

### Blocks

[SDRu Receiver](#) | [SDRu Transmitter](#)

### Functions

[getSDRuDriverVersion](#) | [setsdruip](#)

### Objects

[comm.SDRuReceiver](#) | [comm.SDRuTransmitter](#)

### Introduced in R2012b

# getSDRuDriverVersion

UHD version number of Communications Toolbox Support Package for USRP Radio

## Syntax

```
uhdVersion = getSDRuDriverVersion
```

## Description

`uhdVersion = getSDRuDriverVersion` returns the UHD version number of the installed Communications Toolbox Support Package for USRP Radio.

## Examples

### Get UHD Driver Version Number

Get the UHD version number of the installed Communications Toolbox™ Support Package for USRP™ Radio.

```
uhdVersion = getSDRuDriverVersion  
uhdVersion =  
'3.15.0.0-vendor'
```

## Output Arguments

### **uhdVersion — UHD version number**

character vector

UHD version number of the installed Communications Toolbox Support Package for USRP Radio, returned as a character vector.

## See Also

### Functions

`sdrupload`

### Topics

“Check Radio Firmware” on page 1-23

“USRP Radio Firmware Update” on page 1-53

## Introduced in R2016a

# probesdru

USRP radio information

## Syntax

```
info = probesdru
info = probesdru(IPAddress)
info = probesdru(serialNum)
[info,status] = probesdru(____)
```

## Description

`info = probesdru` returns information about the USRP radio connected to the host computer. If more than one USRP radio is connected to the host computer, the function returns radio information for the first discovered radio. This function calls the UHD application `uhd_usrp_probe`, provided by Ettus Research™, as a system command and returns the command output.

`info = probesdru(IPAddress)` returns detailed information on the USRP radio located at the specified IP address.

`info = probesdru(serialNum)` returns detailed information on the USRP radio with the specified serial number.

`[info,status] = probesdru(____)` also returns the status of the system command execution as the `Status` output. A nonzero `Status` value indicates an error. Specify an input argument combination from any of the previous syntaxes.

## Examples

### Get USRP Radio Information

Get detailed information about the USRP radio connected to the host computer. This function syntax returns radio information for the first radio that the function finds. If more than one radio is connected to the host computer, use the `probesdru(IPAddress)` or `probesdru(serialNum)` syntax instead.

```
I = probesdru
I =
' [0;32m[INFO] [UHD] [0;39mlinux; GNU C++ version 6.3.0; Boost_107000; UHD_3.15.0.0-vendor
[0;32m[INFO] [B200] [0;39mDetected Device: B210
[0;32m[INFO] [B200] [0;39mOperating over USB 3.
[0;32m[INFO] [B200] [0;39mInitialize CODEC control...
[0;32m[INFO] [B200] [0;39mInitialize Radio control...
[0;32m[INFO] [B200] [0;39mPerforming register loopback test...
[0;32m[INFO] [B200] [0;39mRegister loopback test passed
[0;32m[INFO] [B200] [0;39mPerforming register loopback test...
[0;32m[INFO] [B200] [0;39mRegister loopback test passed
[0;32m[INFO] [B200] [0;39mSetting master clock rate selection to 'automatic'.
[0;32m[INFO] [B200] [0;39mAsking for clock rate 16.000000 MHz...'
```

```
[0;32m[INFO] [B200] [0;39mActually got clock rate 16.000000 MHz.
```

```
/ Device: B-Series Device  
|  
| Mboard: B210  
| serial: 31B92DD  
| name: MyB210  
| product: 2  
| revision: 4  
| FW Version: 8.0  
| FPGA Version: 16.0  
|  
| Time sources: none, internal, external, gpsdo  
| Clock sources: internal, external, gpsdo  
| Sensors: ref_locked  
|  
| / RX DSP: 0  
| | Freq range: -8.000 to 8.000 MHz  
| |  
| | / RX DSP: 1  
| | | Freq range: -8.000 to 8.000 MHz  
| | |  
| | / RX Dboard: A  
| | |  
| | | / RX Frontend: A  
| | | | Name: FE-RX2  
| | | | Antennas: TX/RX, RX2  
| | | | Sensors: temp, rss, lo_locked  
| | | | Freq range: 50.000 to 6000.000 MHz  
| | | | Gain range PGA: 0.0 to 76.0 step 1.0 dB  
| | | | Bandwidth range: 200000.0 to 56000000.0 step 0.0 Hz  
| | | | Connection Type: IQ  
| | | | Uses LO offset: No  
| | |  
| | | / RX Frontend: B  
| | | | Name: FE-RX1  
| | | | Antennas: TX/RX, RX2  
| | | | Sensors: temp, rss, lo_locked  
| | | | Freq range: 50.000 to 6000.000 MHz  
| | | | Gain range PGA: 0.0 to 76.0 step 1.0 dB  
| | | | Bandwidth range: 200000.0 to 56000000.0 step 0.0 Hz  
| | | | Connection Type: IQ  
| | | | Uses LO offset: No  
| | |  
| | | / RX Codec: A  
| | | | Name: B210 RX dual ADC  
| | | | Gain Elements: None
```

```
/ TX DSP: 0
| Freq range: -8.000 to 8.000 MHz
|
/ TX DSP: 1
| Freq range: -8.000 to 8.000 MHz
|
/ TX Dboard: A
|
/ TX Frontend: A
| Name: FE-TX2
| Antennas: TX/RX
| Sensors: temp, lo_locked
| Freq range: 50.000 to 6000.000 MHz
| Gain range PGA: 0.0 to 89.8 step 0.2 dB
| Bandwidth range: 200000.0 to 56000000.0 step 0.0 Hz
| Connection Type: IQ
| Uses L0 offset: No
|
/ TX Frontend: B
| Name: FE-TX1
| Antennas: TX/RX
| Sensors: temp, lo_locked
| Freq range: 50.000 to 6000.000 MHz
| Gain range PGA: 0.0 to 89.8 step 0.2 dB
| Bandwidth range: 200000.0 to 56000000.0 step 0.0 Hz
| Connection Type: IQ
| Uses L0 offset: No
|
/ TX Codec: A
| Name: B210 TX dual DAC
| Gain Elements: None
```

### Get Information About USRP Radio with Specified Serial Number

Get detailed information about the USRP radio with serial number '31B92DD'.

```
[info,status] = probesdru('31B92DD');
disp(info)

[0;32m[INFO] [UHD] [0;39mlinux; GNU C++ version 6.3.0; Boost_107000; UHD_3.15.0.0-vendor
[0;32m[INFO] [B200] [0;39mDetected Device: B210
[0;32m[INFO] [B200] [0;39mOperating over USB 3.
[0;32m[INFO] [B200] [0;39mInitialize CODEC control...
[0;32m[INFO] [B200] [0;39mInitialize Radio control...
```

```
[0;32m[INFO] [B200] [0;39mPerforming register loopback test...
[0;32m[INFO] [B200] [0;39mRegister loopback test passed
[0;32m[INFO] [B200] [0;39mPerforming register loopback test...
[0;32m[INFO] [B200] [0;39mRegister loopback test passed
[0;32m[INFO] [B200] [0;39mSetting master clock rate selection to 'automatic'.
[0;32m[INFO] [B200] [0;39mAsking for clock rate 16.000000 MHz...
[0;32m[INFO] [B200] [0;39mActually got clock rate 16.000000 MHz.
```

```
Device: B-Series Device
```

```
Mboard: B210
serial: 31B92DD
name: MyB210
product: 2
revision: 4
FW Version: 8.0
FPGA Version: 16.0
```

```
Time sources: none, internal, external, gpsdo
Clock sources: internal, external, gpsdo
Sensors: ref_locked
```

```
RX DSP: 0
```

```
Freq range: -8.000 to 8.000 MHz
```

```
RX DSP: 1
```

```
Freq range: -8.000 to 8.000 MHz
```

```
RX Dboard: A
```

```
RX Frontend: A
Name: FE-RX2
Antennas: TX/RX, RX2
Sensors: temp, rss, lo_locked
Freq range: 50.000 to 6000.000 MHz
Gain range PGA: 0.0 to 76.0 step 1.0 dB
Bandwidth range: 200000.0 to 5600000.0 step 0.0 Hz
Connection Type: IQ
Uses LO offset: No
```

```
RX Frontend: B
```

```
Name: FE-RX1
Antennas: TX/RX, RX2
Sensors: temp, rss, lo_locked
Freq range: 50.000 to 6000.000 MHz
Gain range PGA: 0.0 to 76.0 step 1.0 dB
Bandwidth range: 200000.0 to 5600000.0 step 0.0 Hz
Connection Type: IQ
Uses LO offset: No
```

```
    / RX Codec: A
    | Name: B210 RX dual ADC
    | Gain Elements: None
    |
    / TX DSP: 0
    | Freq range: -8.000 to 8.000 MHz
    |
    / TX DSP: 1
    | Freq range: -8.000 to 8.000 MHz
    |
    / TX Dboard: A
    | TX Frontend: A
    | Name: FE-TX2
    | Antennas: TX/RX
    | Sensors: temp, lo_locked
    | Freq range: 50.000 to 6000.000 MHz
    | Gain range PGA: 0.0 to 89.8 step 0.2 dB
    | Bandwidth range: 200000.0 to 5600000.0 step 0.0 Hz
    | Connection Type: IQ
    | Uses LO offset: No
    |
    / TX Frontend: B
    | Name: FE-TX1
    | Antennas: TX/RX
    | Sensors: temp, lo_locked
    | Freq range: 50.000 to 6000.000 MHz
    | Gain range PGA: 0.0 to 89.8 step 0.2 dB
    | Bandwidth range: 200000.0 to 5600000.0 step 0.0 Hz
    | Connection Type: IQ
    | Uses LO offset: No
    |
    / TX Codec: A
    | Name: B210 TX dual DAC
    | Gain Elements: None
    |
    disp(status)
    0
```

## Input Arguments

### IPAddress — IP address of USRP radio

dotted-quad character vector | dotted-quad string scalar

IP address of the USRP radio that is connected to the host computer, specified as a dotted-quad character vector or dotted-quad string scalar.

Data Types: char | string

**serialNum — Serial number of USRP radio**

character vector | string scalar

Serial number of the USRP radio that is connected to the host computer, specified as a character vector or string scalar.

Data Types: char | string

## Output Arguments

**info — USRP radio information**

character vector | string scalar

USRP radio information, returned as a character vector or string scalar. If you connect more than one USRP radio to the host computer, the function returns radio information for the first discovered radio.

**status — Status of system command execution**

nonnegative integer

Status of system command execution, returned as a nonnegative integer. A nonzero value indicates an error.

## See Also

**Blocks**

SDRu Receiver | SDRu Transmitter

**Functions**

findsdr | getSDRuDriverVersion | setsdrui

**Objects**

comm.SDRuReceiver | comm.SDRuTransmitter

**Introduced in R2013a**

# sdruload

Load FPGA and firmware images for USRP radio

## Syntax

```
sdruload(Name,Value)
STATUS = sdruload(____)
```

## Description

`sdruload(Name,Value)` loads the default FPGA and UHD firmware images for device specified. For example, `sdruload('Device','USRP2')`. The `Name,Value` pair specifying the device is required, other `Name,Value` pairs are optional.

- Firmware images are the UHD versions compatible with the Communications Toolbox Support Package for USRP Radio.
  - If the device is an N2xx or X3xx series radio, this syntax loads the default images to the radio at IP address 192.168.10.2.
  - If the device is a USRP2 radio, this syntax writes the images to an SD card.

You can obtain the compatible UHD version number by entering `getSDRDriverVersion` at the MATLAB command prompt.

`STATUS = sdruload(____)` returns the status information of the call to `sdruload`.

---

**Note** `sdruload` uses the `uhd_image_loader` utility or the `usrp2_card_burner.py` Python script provided by Ettus Research for burning firmware images to the device.

---

**Warning** When burning images with the card burner, it is possible for you to overwrite your hard drive. To avoid accidentally overwriting the wrong drive, when using the card burner script, carefully select the correct drive for the radio.

---

## Examples

### Load Custom FPGA to N210 Device

Load a custom FPGA image to an N210 device and return the status of the operation.

```
status = sdruload('Device','N210','IPAddress','192.168.30.8',...
'FPGAIImage','c:\sdruload\images\usrp_n210_r4_fpga_ex.bin')

Checking radio connections...
Ready to write FPGA image
    usrp_n210_r4_fpga_ex.bin
and default firmware image
to n210 device at 192.168.30.8. Would you like to continue? [yes/no]: yes

Writing images using uhd_image_loader ...

==== Start messages from third party application ====
```

```

linux; GNU C++ version 4.9.3; Boost_105600; UHD_003.009.004-vendor

Unit: USRP N210 r4 (ECR16TEUP, 192.168.30.8)
Firmware image: c:\sdru\uhdapps\images\usrp_n210_fw.bin
-- Erasing firmware image...successful.

-- Writing firmware image (0%)
-- Writing firmware image (1%)
.
.
.
-- Writing firmware image (98%)
-- Writing firmware image...successful.

-- Verifying firmware image (0%)
-- Verifying firmware image (1%)
.
.
.
-- Verifying firmware image (98%)
-- Verifying firmware image...successful.
FPGA image: c:\sdru\uhdapps\images\usrp_n210_r4_fpga_ex.bin
-- Erasing FPGA image...successful.

-- Writing FPGA image (0%)
...
-- Writing FPGA image (99%)
-- Writing FPGA image...successful.

-- Verifying FPGA image (0%)
.
.
.
-- Verifying FPGA image (99%)
-- Verifying FPGA image...successful.
-- Resetting device...successful.
===== End messages from third party application =====

status =
logical
1

```

## Input Arguments

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`,`Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1`,`Value1`, ..., `NameN`,`ValueN`.

Example: `sdruload('Device','USRP2')`

#### **Device — USRP radio**

`'USRP2' | 'n200' | 'n210' | 'x300' | 'x310'`

USRP radio, specified as a comma-separated pair consisting of '`Device`' and a USRP radio device from this set {`'USRP2'`, `'n200'`, `'n210'`, `'x300'`, or `'x310'`}. The '`Device`' `Name`,`Value` is required.

- If you specify an invalid device, MATLAB responds displaying the list of valid devices.
- If the device you specify does not match the device connected, the error message returned by MATLAB lists the connected device name and prompts you to specify the connected device or to connect another device.
- Additional software must be installed prior to executing `sdruload` to load firmware for an '`USRP2`' radio. For more information, see "Update USRP2 Radio Firmware" on page 1-53.

Example: 'Device','n200'

**IPAddress — IP address where the N2xx or X3xx series radio is located**  
character vector

IP address where the N2xx or X3xx series radio is located, specified as a comma-separated pair consisting of 'IPAddress' and a valid IP address.

Example: 'IPAddress','192.168.10.2'

**Drive — SD card drive**

character vector

Valid SD card drive for USRP2 device, specified as a comma-separated pair consisting of 'Drive' and a valid SD card drive. When Device is specified as 'USRP2', sdruload loads the images for a USRP2 radio to an SD card at the SD card drive specified. If you do not specify a value for 'Drive', the function searches for possible SD card drives and prompts you to select one.

This option uses the uhd\_image\_loader utility provided by Ettus Research.

Example: 'Drive','S:'

**FPGAIImage — FPGA image**

character vector

FPGA image, specified as a comma-separated pair consisting of 'FPGAIImage' and a valid FPGA image file. Use this option to load the FPGA image that is compatible with the UHD version supported by MATLAB and Simulink.

You can also use this option to load custom FPGA images, including images you generate using the HDL workflow advisor. For more information, see HDL Coder.

---

**Note** Custom FPGA image uploading is not supported for X300 or X310.

---

Example: 'FPGAIImage','c:\work\fpga\usrp\_n210\_r4\_fpga.bin'

**FirmwareImage — Firmware image**

character vector

Firmware image, specified as a comma-separated pair consisting of 'FirmwareImage' and a valid firmware image file. Use this option to load the UHD firmware image that is compatible with the UHD version supported by MATLAB and Simulink.

---

**Note** Custom firmware image updating is not supported for X300 or X310.

---

Example: 'FirmwareImage','c:\work\fpga\usrp\_n210\_fw.bin'

## Output Arguments

**STATUS — Status of call to sdruload**  
logical

Status of call to `sdruload`, returned as `true` if the operation was successful.

## See Also

### Topics

- “USRP Radio Firmware Update” on page 1-53
- “FPGA Targeting Overview” on page 6-2
- “Target FPGA with Custom Bitstream” on page 6-3

**Introduced in R2013b**

# setsdruip

Set IP address for USRP N2xx or X3xx series radio

## Syntax

```
info = setsdruip(currentIP,newIP)
[info,status] = setsdruip(currentIP,newIP)
```

## Description

`info = setsdruip(currentIP,newIP)` sets the IP address of a USRP N2xx or X3xx series radio. The current IP address, `CurrentIP`, is replaced with new IP address, `NewIP`. The function calls the UHD application `usrp_burn_mb_eeprom`, provided by Ettus Research™, as a system command and returns the command output in `I`. You must power-cycle the radio for the IP address change to take effect. This function does not support N3xx series radios.

`[info,status] = setsdruip(currentIP,newIP)` returns the status of the system command execution as output. A nonzero `S` value indicates an error.

## Examples

### Set IP Address of USRP Radio

Set the IP address of a USRP radio at IP address 192.168.30.22 to 192.168.30.20. This example assumes that you have a USRP radio at IP address 192.168.30.22.

```
I = setsdruip('192.168.30.22','192.168.30.20')
I =
'192.168.30.20'
```

## Input Arguments

### **currentIP — Current IP address**

dotted-quad character vector | dotted-quad string scalar

Current IP address of a USRP N2xx or X3xx series radio, specified as dotted-quad character vector or a dotted-quad string scalar.

Data Types: `char` | `string`

### **newIP — New IP address**

dotted-quad character vector | dotted-quad string scalar

New IP address to be assigned to the USRP N2xx or X3xx series radio, specified as dotted-quad character vector or a dotted-quad string scalar. The current IP address of the radio, `CurrentIP`, is replaced with new IP address, `NewIP`.

---

**Note** If you change the subnet address of the radio (the subnet address is the third number in the IP character vector), you must reconfigure the Ethernet port of your host computer with the same subnet address of the radio so that the host computer can communicate with the radio.

---

Data Types: char | string

## Output Arguments

### **info — New IP address of USRP radio**

character vector | string scalar

New IP address of the USRP radio, returned as a character vector or string scalar.

Data Types: char | string

### **status — Status of system command execution**

nonnegative integer

Status of the system command execution, returned as a nonnegative integer. A nonzero value indicates an error.

## See Also

### **Blocks**

SDRu Receiver | SDRu Transmitter

### **Functions**

findsdr | getSDRDriverVersion | probesdr

### **Objects**

comm.SDRuReceiver | comm.SDRuTransmitter

## Introduced in R2013a



# Communications Toolbox Support Package for USRP Radio Examples

---

## Packetized Modem with Data Link Layer

This example shows you how to implement a packetized modem with Data Link Layer [ 1 ] using MATLAB® and Communications Toolbox™. The modem features a packet-based physical layer and an ALOHA-based Data Link Layer. You can either simulate the system or run with radios using the Communications Toolbox Support Package for USRP® Radio.

### Required Hardware and Software

To simulate the system performance, you need the following software:

- Communications Toolbox™

To measure system performance with radios, you also need the following hardware:

- USRP® radios (B2xx, N2xx, or X3xx)

and the following software

- Communications Toolbox Support Package for USRP® Radio

For a full list of Communications Toolbox supported SDR platforms, refer to the Supported Hardware section of Software Defined Radio (SDR) discovery page.

### Introduction

Packetized wireless modems are communications systems that transmit information in bursts called packets through a wireless channel. Each modem, also called a node, features a physical layer where packets are modulated, transmitted and received on a shared frequency band, and demodulated. Since the same frequency band is used by all nodes, a medium access control (MAC) algorithm is required to reduce the packet loss due to collisions (i.e. simultaneous transmissions). Data Link Layer includes a MAC sublayer and a logical link control sublayer to share the same channel and provides an error-free link between two nodes. Data Link Layer is also called Layer 2 and sits between Network Layer (Layer 3) and Physical Layer (Layer 1).

### Run the Example

To run the example, type `PacketizedModemNetworkExample` in the MATLAB Command Window or click the link. The example code creates three packetized modem node objects and connects them through a channel object. Each node can send packets to the other two nodes. `ACKTimeout` determines the timeout duration before a node decides the DATA packet transmission was not successful. `ACKTimeout` must be greater than the round trip duration for a DATA-ACK exchange, which is 0.21 seconds for this example. The simulation is time-based and simulates the full physical layer processing together with the data link layer.

```
% Set simulation parameters
runDuration = 10; % Seconds
numPayloadBits = 19530; % Bits
packetArrivalRate = 0.2; % Packets per second
ackTimeOut = 0.25; % ACK time out in seconds
maxBackoffTime = 10; % Maximum backoff time in ackTimeOut durations
mMaxDataRetries = 5; % Maximum DATA retries
queueSize = 10; % Data Link Layer queue size in packets
samplesPerFrame = 2000; % Number of samples processed every iteration
verbose = true; % Print packet activity to command line
sampleRate = 200e3;
```

```
% Fix random number generation seed for repeatable simulations
rng(12345)

% Create packetized modem nodes
node1 = helperPacketizedModemNode(
    'Address', 1, ...
    'DestinationList', [2, 3], ...
    'NumPayloadBits', numPayloadBits, ...
    'PacketArrivalRate', packetArrivalRate, ...
    'ACKTimeOut', ackTimeOut, ...
    'MaxBackoffTime', maxBackoffTime, ...
    'MaxDataRetries', mMaxDataRetries, ...
    'QueueSize', queueSize, ...
    'CarrierDetectorThreshold', 1e-5, ...
    'AGCMaxPowerGain', 65, ...
    'SamplesPerFrame', samplesPerFrame, ...
    'Verbose', verbose, ...
    'SampleRate', sampleRate);
node2 = helperPacketizedModemNode(
    'Address', 2, ...
    'DestinationList', [1 3], ...
    'NumPayloadBits', numPayloadBits, ...
    'PacketArrivalRate', packetArrivalRate, ...
    'ACKTimeOut', ackTimeOut, ...
    'MaxBackoffTime', maxBackoffTime, ...
    'MaxDataRetries', mMaxDataRetries, ...
    'QueueSize', queueSize, ...
    'CarrierDetectorThreshold', 1e-5, ...
    'AGCMaxPowerGain', 65, ...
    'SamplesPerFrame', samplesPerFrame, ...
    'Verbose', verbose, ...
    'SampleRate', sampleRate);
node3 = helperPacketizedModemNode(
    'Address', 3, ...
    'DestinationList', [1 2], ...
    'NumPayloadBits', numPayloadBits, ...
    'PacketArrivalRate', packetArrivalRate, ...
    'ACKTimeOut', ackTimeOut, ...
    'MaxBackoffTime', maxBackoffTime, ...
    'MaxDataRetries', mMaxDataRetries, ...
    'QueueSize', queueSize, ...
    'CarrierDetectorThreshold', 1e-5, ...
    'AGCMaxPowerGain', 65, ...
    'SamplesPerFrame', samplesPerFrame, ...
    'Verbose', verbose, ...
    'SampleRate', sampleRate);

% Setup channel
channel = helperMultiUserChannel(
    'NumNodes', 3, ...
    'EnableTimingSkew', true, ...
    'DelayType', 'Triangle', ...
    'TimingError', 20, ...
    'EnableFrequencyOffset', true, ...
        'PhaseOffset', 47, ...
    'FrequencyOffset', 2000, ...
    'EnableAWGN', true, ...
```

```
'EbNo', 25, ...
'BitsPerSymbol', 2, ...
'SamplesPerSymbol', 4, ...
'EnableRicianMultipath', true, ...
'PathDelays', [0 node1.SamplesPerSymbol/node1.SampleRate], ...
'AveragePathGains', [15 0], ...
'KFactor', 15, ...
'MaximumDopplerShift', 10, ...
'SampleRate', node1.SampleRate);

% Main loop
radioTime = 0;
nodeInfo = info(node1);
frameDuration = node1.SamplesPerFrame/node1.SampleRate;
[rcvd1,rcvd2,rcvd3] = deal(complex(zeros(node1.SamplesPerFrame,1)));
while radioTime < runDuration
    trans1 = node1(rcvd1, radioTime);
    trans2 = node2(rcvd2, radioTime);
    trans3 = node3(rcvd3, radioTime);

    % Multi-user channel
    [rcvd1,rcvd2,rcvd3] = channel(trans1,trans2,trans3);

    % Update radio time.
    radioTime = radioTime + frameDuration;
end
```

Time	Link	Action	Seq #	Backoff
4.46000 s	3 ->> 1	DATA	#	0
4.67000 s	1 <<- 3	DATA	#	0
4.67000 s	1 ->> 3	ACK	#	0
4.68000 s	3 <<- 1	ACK	#	0
5.04000 s	1 ->> 3	DATA	#	0
5.16000 s	2 ->> 3	DATA	#	0
5.30000 s	1 ->> 3	Back Off	#	0   1.00000 s
5.42000 s	2 ->> 3	Back Off	#	0   1.00000 s
6.31000 s	1 ->> 3	DATA	#	0
6.43000 s	2 ->> 3	DATA	#	0
6.57000 s	1 ->> 3	Back Off	#	0   2.25000 s
6.69000 s	2 ->> 3	Back Off	#	0   1.75000 s
8.45000 s	2 ->> 3	DATA	#	0
8.66000 s	3 <<- 2	DATA	#	0
8.66000 s	3 ->> 2	ACK	#	0
8.67000 s	2 <<- 3	ACK	#	0
8.83000 s	1 ->> 3	DATA	#	0
9.09000 s	1 ->> 3	Back Off	#	0   2.25000 s
9.52000 s	3 ->> 2	DATA	#	1
9.73000 s	2 <<- 3	DATA	#	1
9.73000 s	2 ->> 3	ACK	#	1
9.74000 s	3 <<- 2	ACK	#	1

## Results

The Node object collect statistics on the performance of the Data Link Layer algorithm. Call the `info` method of the Node object to access these statistics. Following shows a sample result for a 10 second simulated time with 0.2 packets/second packet arrival rate. Each data packet is 200 msec long.

```
% Display statistics
nodeInfo(1) = info(node1);
nodeInfo(2) = info(node2);
nodeInfo(3) = info(node3);

for p=1:length(nodeInfo)
    fprintf('\nNode %d:\n', p);
    fprintf('\tNumGeneratedPackets: %d\n', nodeInfo(p).NumGeneratedPackets)
    fprintf('\tNumReceivedPackets: %d\n', nodeInfo(p).NumReceivedPackets)
    fprintf('\tAverageRetries: %f\n', nodeInfo(p).Layer2.AverageRetries)
    fprintf('\tAverageRoundTripTime: %f\n', nodeInfo(p).Layer2.AverageRoundTripTime)
    fprintf('\tNumDroppedPackets: %d\n', nodeInfo(p).Layer2.NumDroppedPackets)
    fprintf('\tNumDroppedPackets (Max retries): %d\n', nodeInfo(p).Layer2.NumDroppedPacketsDueToRetries)
    fprintf('\tThroughput: %d\n', numPayloadBits / nodeInfo(p).Layer2.AverageRoundTripTime)
    fprintf('\tLatency: %d\n', nodeInfo(p).Layer2.AverageLatency)
end
```

Node 1:

```
NumGeneratedPackets: 2
NumReceivedPackets: 1
AverageRetries: NaN
AverageRoundTripTime: NaN
NumDroppedPackets: 0
NumDroppedPackets (Max retries): 0
Throughput: NaN
Latency: Inf
```

Node 2:

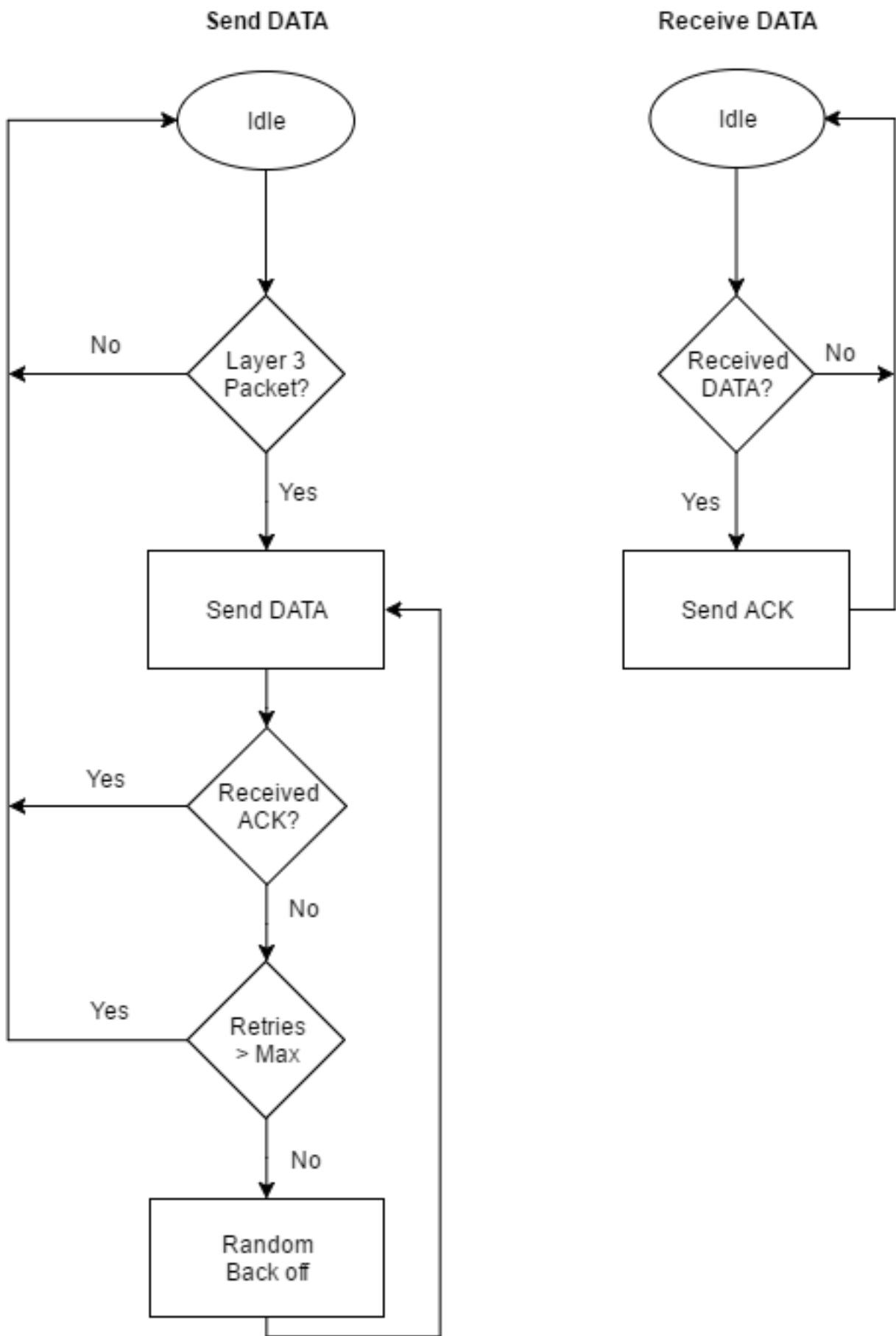
```
NumGeneratedPackets: 1
NumReceivedPackets: 1
AverageRetries: 2.000000
AverageRoundTripTime: 3.509844
NumDroppedPackets: 0
NumDroppedPackets (Max retries): 0
Throughput: 5.564350e+03
Latency: 2.104687e-01
```

Node 3:

```
NumGeneratedPackets: 2
NumReceivedPackets: 1
AverageRetries: 0.000000
AverageRoundTripTime: 0.220254
NumDroppedPackets: 0
NumDroppedPackets (Max retries): 0
Throughput: 8.867039e+04
Latency: 1.749922e+00
```

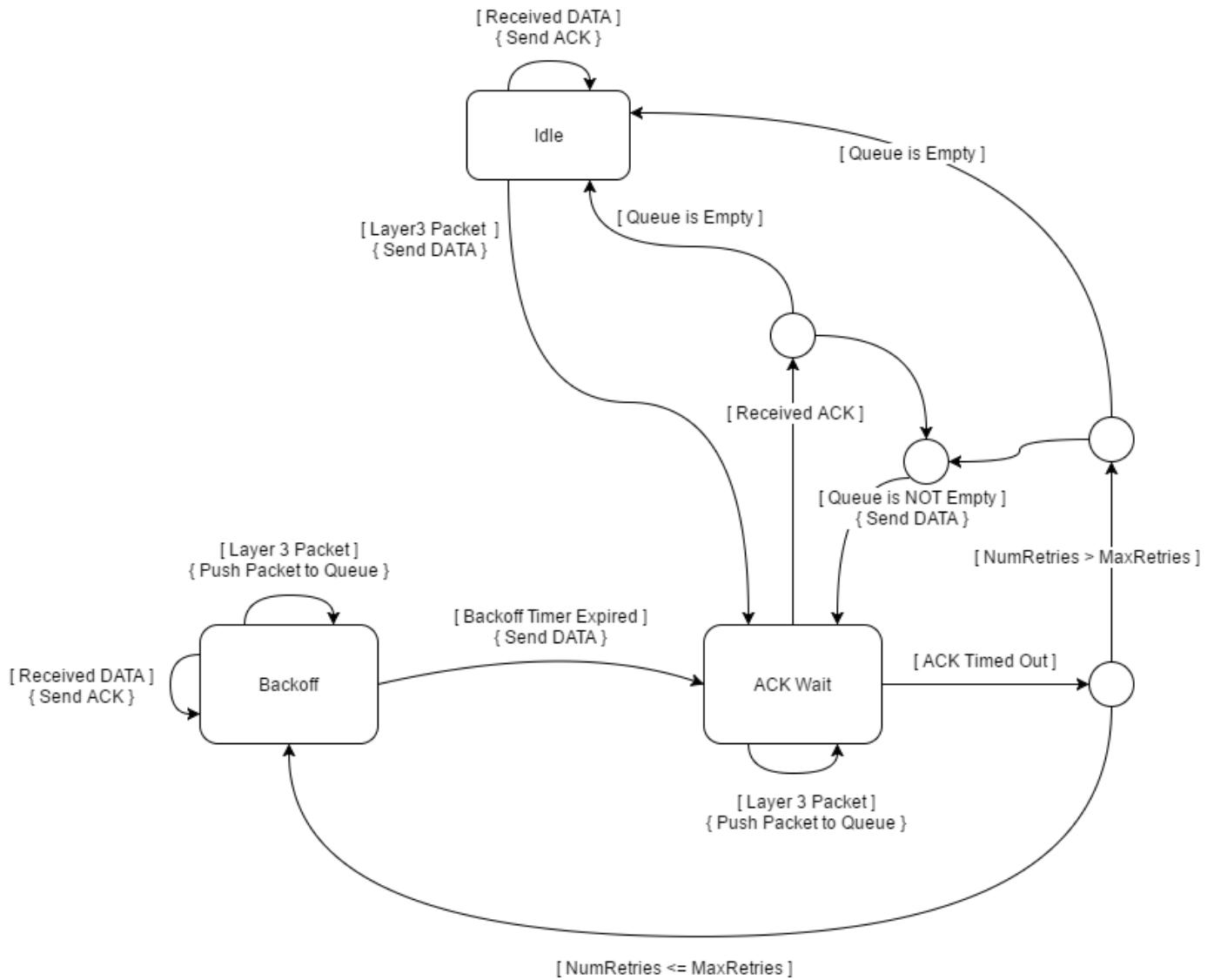
## Data Link Layer (Layer 2)

This example implements a Data Link Layer based on the ALOHA random access protocol [ 2 ]. The following flow diagram shows how the ALOHA protocol transmits and receives data packets.



When Data Link Layer has a Layer 3 packet to transmit, it starts a new session and sends the packet right away using a DATA packet. The algorithm waits for an acknowledgment (ACK) packet. If an ACK is not received before the timeout period, it backs off a random amount of time and sends the DATA packet again. If it fails to receive an ACK after a number of retries, it drops the packet. If during this session, a new Layer 3 packet is received, the Layer 3 packet is put in a first-in-first-out (FIFO) queue. If the FIFO queue is full, packet is dropped.

The algorithm is implemented in the helperPacketizedModemDataLinkLayer System object™. The helperPacketizedModemDataLinkLayer System object defines a state machine with three states: IDLE, ACK\_WAIT, and BACKOFF. The following state machine describes how the data link layer algorithm is implemented in this object. Statements in brackets, [...], and curly braces, {...}, are conditions and actions, respectively. Small circles are passthrough states used to represent multiple conditions.



The original ALOHA protocol uses a hub/star topology. The uplink and downlink utilizes two separate frequency bands. The following example employs a mesh network topology where nodes transmit and receive using the same frequency band.

### **Modem Structure**

The following presents the modem code structure. The processing has six main parts and runs in the following order:

- 1** Source Controller
- 2** Message Generator
- 3** PHY Decoder
- 4** Data Link Layer
- 5** Message Parser
- 6** PHY Encoder

The Data Link Layer processes outputs of the Message Generator and PHY Decoder, so it must run after those two operations. The Message Parser and PHY Encoder process outputs of the Data Link Layer. This sequence ensures that the modem can receive packets and respond to them in the same time interval. The `helperPacketizedModemNode` object implements the modem.

### **Source Controller**

The Source Controller generates an enable signal and a random destination address based on the user-selected packet arrival distribution.

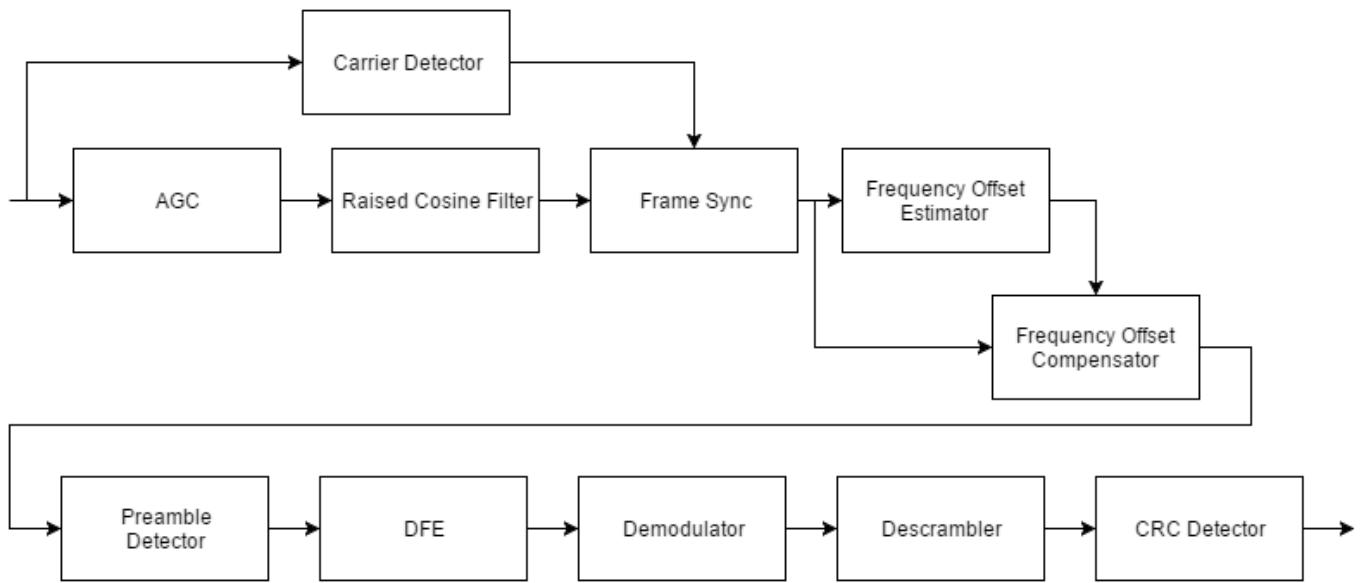
### **Message Generator**

The Message Generator starts creating layer 3 data packets when enabled by the source controller. The packets contain a digitized text message. If the message does not fit into one packet, the generator creates multiple packets. The packet structure is as follows:

- To Address: 8 bits
- From Address: 8 bits
- Packet Number: 16 bits
- Payload: M bits

### **PHY Decoder**

The PHY Decoder receives baseband I/Q samples and creates layer 2 packets. PHY Decoder can correct for amplitude variations using an AGC, frequency offsets with a frequency offset estimator and compensator, and timing skews and multipath using a fractionally spaced decision feedback equalizer (DFE). The block diagram of the physical layer (Layer 1) receiver is as follows:



When data payload size is set to 19530 bits, the total packet length of the modem is 39956 samples. The modem processes `SamplesPerFrame` samples, which is 2000 samples for this example, at each iteration. A smaller `SamplesPerFrame` results in smaller latency but increases the overhead of the modem algorithm. An increased overhead may increase the processing time such that the modem does not run in real-time anymore.

## Data Link Layer

Data Link Layer provides a link between two neighboring nodes. It employs the ALOHA-based protocol described in the [Data Link Layer \(Layer 2\)](#) section. The packet structure is as follows:

- Type: 4 bits
- Version: 2 bits
- Reserved: 2 bits
- To Address: 8 bits
- From Address: 8 bits
- Sequence Number: 8 bits
- Time stamp: 32 bits
- Payload: N (= M+32) bits

Data Link Layer also collects following statistics:

- Number of successful packet transfers, which is defined as the number of successfully received ACK packets
- Average retries
- Average round trip time in seconds
- Number of dropped packets due to layer 3 packet queue being full
- Number of dropped packets due to retries
- Throughput defined as successful data delivery rate in bits per second

- Average latency in seconds defined as the time between the generation of the layer 3 data packet and reception of it at the destination node

### Message Parser

The Message Parser parses received layer 2 payload and creates Layer 3 packet. It also collects the following statistics:

- Number of received packets
- Number of received duplicate packets

### PHY Encoder

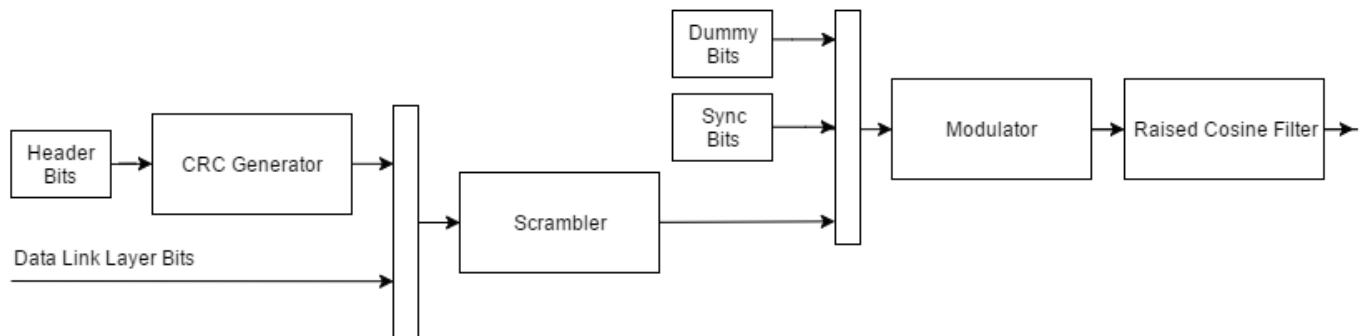
PHY Encoder creates physical layer packets by modulating the layer 2 packets into baseband I/Q samples. The packet structure is as follows:

Dummy Symbols	Synchronization Symbols	Payload Length	CRC	Payload Symbols
---------------	-------------------------	----------------	-----	-----------------

The dummy symbols are used to train the AGC and for carrier detection. The synchronization symbols are a modulated PN-sequence. The header has following fields:

- Payload length: 16 bits
- CRC: 16 bits

The block diagram of the physical layer (Layer 1) transmitter is as follows:

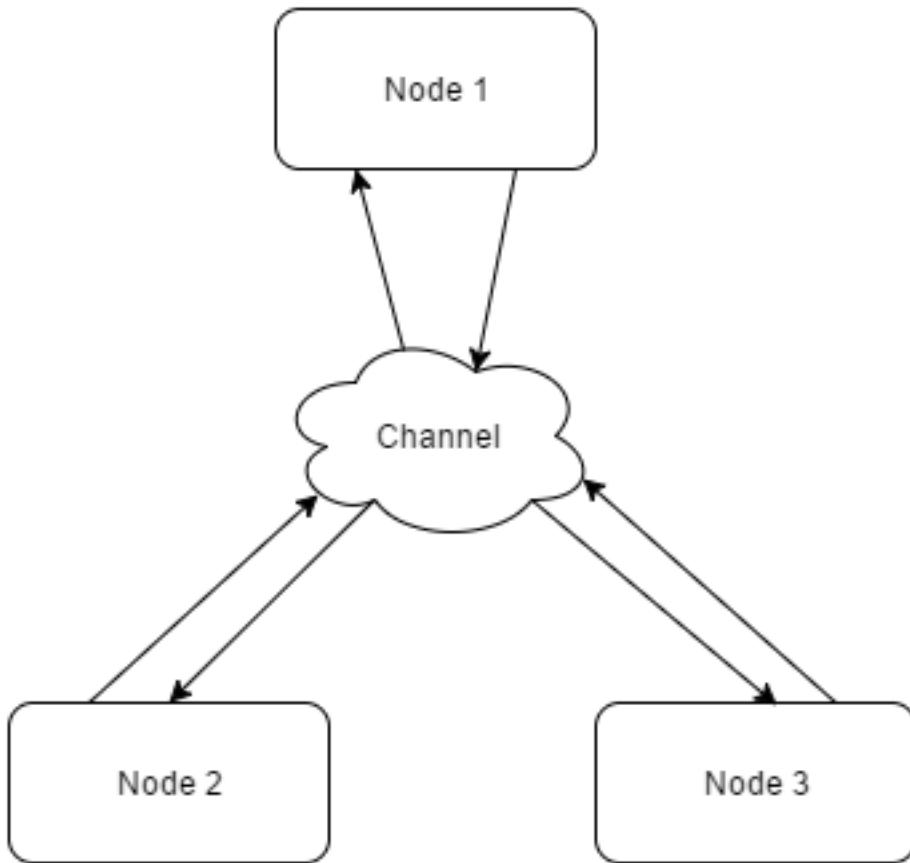


### Channel Model

This example simulates a three-node network but any number of nodes can be simulated. The output of each node is passed to the channel simulator. The channel adds baseband signals from all three nodes after imposing the following channel impairments:

- Timing skew
- Frequency offset
- Rician multipath
- AWGN

In addition to these impairments, the signals from neighboring nodes are applied a path loss of 20 dB, while the self-interference is added directly.



### Running Using Radios

You can also run this example using radios instead of a simulated channel. The combination of an SDR hardware and a host computer that runs a MATLAB session comprises a node. The following steps show you how to set up a three-node network. This example uses USRP® B200 and B210 radios.

- 1) Connect a USRP® radio to host computer A, which we will call Node 1. Follow the instruction in “Installation and Setup” to install and setup your host computer for use with USRP® radios. Start a MATLAB session.
- 2) Set up Node 1 as a transmitter for initialization. Run `helperPacketizedModemInitializeRadio('tx', PLATFORM, ADDRESS, FC, RT)`, where PLATFORM is the type of the USRP® radio and ADDRESS is the serial number or IP address, FC is the center frequency, and RT is run time in seconds. This example uses 915 MHz center frequency. Assuming that your radio is a B200 with serial number 'ABCDE', the function call will be `helperPacketizedModemInitializeRadio('tx', 'B200', 'ABCDE', 915e6, 120)`. This function will run the transmitter for 120 seconds. If you need more time to finish the initialization, rerun the command with a longer run time.
- 3) Repeat step 1 for a second radio and host computer and call this node Node 2.
- 4) Set up Node 2 as a receiver for initialization. Run `[CDT, MAXGAIN, RXGAIN] = helperPacketizedModemInitializeRadio('rx', PLATFORM, ADDRESS, FC, RT)`. Assuming that your radio is a B210 with serial number '12345', the function call will be `[CDT1, MAXGAIN1, RXGAIN1] = helperPacketizedModemInitializeRadio('rx', 'B210', '12345', 915e6, 120)`. The function will run until it

determines the best values for carrier detector threshold (CDT), maximum AGC gain (MAXGAIN), and radio receive gain (RXGAIN) or until RT seconds have elapsed. If the initialization algorithm cannot determine suitable parameters, it may suggest increasing or decreasing the transmitter power and retrying the initialization.

5) Run the same experiment with Node 1 as the receiver and Node 2 as the transmitter to determine best receiver parameters for Node 1. In most cases the channel should be dual and the parameters will be very close.

6) Repeat steps 1-5 for all other pairs of radios, i.e. Node 1 and Node 3, Node 3 and Node 2. Obtain CDT, MAXGAIN, and RXGAIN values for each node. If you get different values for the same node while initializing for different links, choose the maximum values for MAXGAIN and RXGAIN, and minimum of CDT.

7) Start Node 1 by running the function

`helperPacketizedModemRadio(P,RA,NA,DA,FC,CDT,MAXG,RGAIN,D)`, where P is platform, RA is radio address, NA is node address, DA is destination address list, FC is center frequency, CDT is carrier detection threshold, MAXG is maximum AGC gain, RGAIN is radio receiver gain, and D is duration. For example, run as `helperPacketizedModemRadio('B200', 'ABCDE', 1, [2 3], 915e6, CDT1, MAXGAIN1, RXGAIN1, 120)`.

8) Start Node 2 by running `helperPacketizedModemRadio('B210', '12345', 2, [1 3], 915e6, CDT2, MAXGAIN2, RXGAIN2, 120)`.

9) Start Node 3 by running `helperPacketizedModemRadio('B200', 'A1B2C', 3, [1 2], 915e6, CDT3, MAXGAIN3, RXGAIN3, 120)`.

10) Once the session ends, each node prints out its statistics.

A three network setup operated for two hours. Each node generated packets at a rate of 0.2 packets/second according to a Poisson distribution. The nodes were placed approximately equal distance. One of the links had line-of-sight while other two did not. The following are the results collected on all three nodes. Since the round trip time of a DATA-ACK exchange using the B2xx radios connected over USB can be as high as 800 msec, the average round trip time of the network is greater than 3 sec. The algorithm minimizes packet loss and provides a fair access to the shared channel to all nodes.

Node 1:

```
NumGeneratedPackets: 1440
NumReceivedPackets: 1389
AverageRetries: 0.533738
AverageRoundTripTime: 3.725093
NumDroppedPackets: 95
NumDroppedPackets (Max retries): 23
Throughput: 5.242823e+03
```

Node 2:

```
NumGeneratedPackets: 1440
NumReceivedPackets: 1340
AverageRetries: 0.473157
AverageRoundTripTime: 3.290775
NumDroppedPackets: 31
NumDroppedPackets (Max retries): 9
Throughput: 5.934772e+03
```

Node 3:

```
NumGeneratedPackets: 1440
```

```

NumReceivedPackets: 1385
AverageRetries: 0.516129
AverageRoundTripTime: 3.558408
NumDroppedPackets: 107
NumDroppedPackets (Max retries): 29
Throughput: 5.488410e+03

```

## Discussions

The simulation code from previous sections and the helperPacketizedModemRadio.m function both utilize the helperPacketizedModemNode.m System object to implement the modem node. In the current example, the same code is used to evaluate a system, first using a simulated channel, then using SDR hardware and over-the-air channels.

Even though the code using simulated channels is time-based, the modem node object could be used to run an event-based simulation. This example does not provide an event-based simulation kernel.

## Further Exploration

You can vary the following parameters to investigate their effect on data link layer performance:

- PacketArrivalRate
- ACKTimeOut
- MaxBackoffTime
- MaxDataRetries
- QueueSize

You can also explore the following functions for details of the implementation of the algorithms:

- helperPacketizedModemNode.m
- helperPacketizedSourceController.m
- helperPacketizedModemMessageGenerator.m
- helperPacketizedModemDataLinkLayer.m
- helperPacketizedModemPHYEncoder.m
- helperPacketizedModemPHYDecoder.m
- helperPacketizedModemMessageParser.m
- helperMultiUserChannel.m

You can examine the physical layer only performance using the PacketizedModemPhysicalLayerTxRxExample script.

## Selected Bibliography

- 1 Data Link Layer
- 2 ALOHANet

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments® Corp.

# Family Radio Service (FRS) Full-Duplex Transceiver with USRP® Hardware

This example shows how to use two Universal Software Radio Peripheral® devices exploiting SDRu (Software Defined Radio USRP® System objects to perform full duplex communication by transmitting/receiving recorded audio simultaneously using MATLAB® and the FRS protocol. For a more detailed description of FRS processing, see [sdruFRSGMRSTransmitter](#) and [sdruFRSGMRSReceiver](#).

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter and SDRu Receiver System objects™.

In order to run this example, you need two host computers connected to two USRP® radios with daughterboards (e.g. SBX or WBX) that support full-duplex operation and the FRS band. Using this script, `sdruFRSFullDuplexTransceiver`, the first USRP® radio sends one audio signal at 467.6125 MHz, and simultaneously receives another audio signal at 467.6625 MHz. The second USRP® radio sends at 467.6625 MHz and receives at 467.6125 MHz using the MATLAB script `sdruFRSFullDuplexTransceiver_2`. Each host computer plays the received audio signal on its audio device.

## Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdr` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdr;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    radioFound = true;
    platform = connectedRadios(1).Platform;
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
        case {'N200/N210/USRP2','X300','X310'}
            address = connectedRadios(1).IPAddress;
    end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end

Checking radio connections...
```

## Initialization

The `configureFDTx` and `configureFDRx` MATLAB functions initialize simulation parameters and generate parameter structures for the FRS transmitter and receiver systems.

The frequency calibration compensation value, `rfRxFreqCorrection`, can be determined for your USRP® environment by running the MATLAB script `sdruFrequencyCalibrationReceiver` while sending a sine wave at the RF receiver center frequency, `rfRxFreq`, with MATLAB script `sdruFrequencyCalibrationTransmitter`.

```
% Transmitter Initialization
rfTxFreq = 467.6125e6; % RF Transmitter Center Frequency (Hz)
frsFDTxParams = configureFDTx(platform, rfTxFreq);

% Receiver Initialization
rfRxFreq = 467.6625e6; % RF Receiver Center Frequency (Hz)
rfRxFreqCorrection = -4e3; % Frequency calibration compensation value (Hz)
rfRxFreqActual = rfRxFreq + rfRxFreqCorrection;
frsFDRxParams = configureFDRx(platform, rfRxFreqActual);
```

### Transmit One Audio File Using FRS Waveform

This example uses a source object to generate data signals for the transmitter. The source signal is a multimedia audio file. When using a multimedia file, the sampling rate needs to be converted to 8 kHz; therefore, the FRSGMRSDemoAudioSource object employs a rate conversion filter to convert the 22.5 kHz signal to an 8 kHz signal.

```
% Create a data source to transmit the contents of a sound file at a
% sampling frequency of 8 kHz.
source = FRSGMRSDemoSource('Sound file', frsFDTxParams.SourceSampleRate);
source.AudioFileName = 'speech_dft.avi';

% The Continuous Tone-Coded Squelch System (CTCSS) filters out
% undesired communication or interference from these other users by
% generating a tone between 67 Hz and 250 Hz and transmitting it along with
% the source signal.
ctcss = dsp.SineWave(frsFDTxParams.CTCSSAmplitude, ...
    frsFDTxParams.CTCSSToneFrequencies(frsFDTxParams.CTCSSCode), ...
    'SampleRate', frsFDTxParams.SourceSampleRate, ...
    'SamplesPerFrame', frsFDTxParams.SourceFrameLength, ...
    'OutputDataType', 'single');

% The interpolator and FM modulator convert the sampling rate of the sum of
% the modulating signal and the CTCSS tone to match the USRP(R) hardware
% sampling rate of 200 kHz.
interpolator = dsp.FIRInterpolator(frsFDTxParams.InterpolationFactor, ...
    frsFDTxParams.InterpolationNumerator);

fmMod = comm.FMModulator('SampleRate', frsFDTxParams.RadioSampleRate, ...
    'FrequencyDeviation', frsFDTxParams.FrequencyDeviation);

% The SDRu transmitter sets the interpolation factor and master clock rate
% so that the example uses round numbers to convert the sampling rate from
% 8 kHz to 200 kHz. B200 and B210 series USRP(R) radios are addressed using
% a serial number while USRP2, N200, N210, X300 and X310 radios are
% addressed using an IP address.

% Set up transmitter radio object to use the found radio
switch platform
    case {'B200','B210'}
        radioTx = comm.SDRuTransmitter('Platform', platform, ...
            'SerialNum', address, ...
            'MasterClockRate', frsFDTxParams.RadioMasterClockRate, ...
            'CenterFrequency', frsFDTxParams.CenterFrequency, ...
            'Gain', frsFDTxParams.RadioGain, ...
            'InterpolationFactor', frsFDTxParams.RadioInterpolationFactor)
    case {'X300','X310'}
        radioTx = comm.SDRuTransmitter('Platform', platform, ...
```

```
'IPAddress', address, ...
'MasterClockRate', frsFDTxParams.RadioMasterClockRate, ...
'CenterFrequency', frsFDTxParams.CenterFrequency, ...
'Gain', frsFDTxParams.RadioGain, ...
'InterpolationFactor', frsFDTxParams.RadioInterpolationFactor)
case {'N200/N210/USRP2'}
    radioTx = comm.SDRuTransmitter('Platform', platform, ...
        'IPAddress', address, ...
        'CenterFrequency', frsFDTxParams.CenterFrequency, ...
        'Gain', frsFDTxParams.RadioGain, ...
        'InterpolationFactor', frsFDTxParams.RadioInterpolationFactor)
end

radioTx =
System: comm.SDRuTransmitter

Properties:
    Platform: 'B210'
    SerialNum: 'F5BA51'
    ChannelMapping: 1
    CenterFrequency: 467612500
    LocalOscillatorOffset: 0
        Gain: 15
        PPSSource: 'Internal'
        ClockSource: 'Internal'
        MasterClockRate: 20000000
    InterpolationFactor: 100
    TransportDataType: 'int16'
    EnableBurstMode: false
```

### Receive Another Audio File Using FRS Demodulation

Create objects to resample the input to 200 kHz, perform automatic gain control, perform channel selectivity filtering, FM demodulate, resample to an 8 kHz audio output, perform CTCSS decoding, filter out the CTCSS tones, then send the signal to an audio output device.

```
% Set up transmitter radio object to use the found radio
switch platform
    case {'B200','B210'}
        radioRx = comm.SDRuReceiver('Platform', platform, ...
            'SerialNum', address, ...
            'MasterClockRate', frsFDRxParams.RadioMasterClockRate, ...
            'CenterFrequency', frsFDRxParams.CenterFrequency, ...
            'Gain', frsFDRxParams.RadioGain, ...
            'DecimationFactor', frsFDRxParams.RadioDecimationFactor, ...
            'SamplesPerFrame', frsFDRxParams.RadioFrameLength, ...
            'OutputDataType', 'single')
    case {'X300','X310'}
        radioRx = comm.SDRuReceiver('Platform', platform, ...
            'IPAddress', address, ...
            'MasterClockRate', frsFDRxParams.RadioMasterClockRate, ...
            'CenterFrequency', frsFDRxParams.CenterFrequency, ...
            'Gain', frsFDRxParams.RadioGain, ...
            'DecimationFactor', frsFDRxParams.RadioDecimationFactor, ...
            'SamplesPerFrame', frsFDRxParams.RadioFrameLength, ...
```

```

    'OutputDataType', 'single')
case {'N200/N210/USRP2'}
    radioRx = comm.SDRuReceiver('Platform', platform, ...
        'IPAddress', address, ...
        'CenterFrequency', frsFDRxParams.CenterFrequency, ...
        'Gain', frsFDRxParams.RadioGain, ...
        'DecimationFactor', frsFDRxParams.RadioDecimationFactor, ...
        'SamplesPerFrame', frsFDRxParams.RadioFrameLength, ...
        'OutputDataType', 'single')
end

% AGC
agc = comm.AGC;

% Low pass filter for channel separation
channelFilter = frsFDRxParams.ChannelFilter;

% FM demodulator
fmDemod = comm.FMDemodulator('SampleRate', frsFDRxParams.RadioSampleRate, ...
    'FrequencyDeviation', frsFDRxParams.FrequencyDeviation);

% Decimation filter to resample to 8 kHz
decimator = dsp.FIRDecimator(frsFDRxParams.DecimationFactor, ...
    frsFDRxParams.DecimationNumerator);

% The CTCSS decoder compares the estimated received code with the
% preselected code and then sends the signal to the audio device if the two
% codes match.
decoder = FRSGMRSDemoCTCSSDecoder(...
    'MinimumBlockLength', frsFDRxParams.CTCSSDecodeBlockLength, ...
    'SampleRate', frsFDRxParams.AudioSampleRate);

% High pass filter to filter out CTCSS tones
audioFilter = frsFDRxParams.AudioFilter;

% Audio device writer
audioPlayer = audioDeviceWriter(frsFDRxParams.AudioSampleRate);

radioRx =
    System: comm.SDRuReceiver
Properties:
    Platform: 'B210'
    SerialNum: 'F5BA51'
    ChannelMapping: 1
    CenterFrequency: 467658500
    LocalOscillatorOffset: 0
        Gain: 5
    MasterClockRate: 20000000
    DecimationFactor: 100
    TransportDataType: 'int16'
        OutputDataType: 'single'
    SamplesPerFrame: 4000
    EnableBurstMode: false

```

## Stream Processing Loop

```
% Perform stream processing if a radio is found.
if radioFound
    % Loop until the example reaches the target stop time.
    timeCounter = 0;

    while timeCounter < frsFDTxParams.StopTime
        % Transmitter stream processing
        %
        dataTx = step(source); % Generate audio waveform
        dataWTone = dataTx + step(ctcss); % Add CTCSS tones

        % Interpolation FM modulation
        outResamp = step(interpolator, dataWTone); % Resample to 200 kHz
        outMod = step(fmMod, outResamp);
        step(radioTx, outMod); % Transmit to USRP(R) radio

        % Receiver stream processing
        %
        [dataRx, lenRx] = step(radioRx);
        if lenRx > 0
            outAGC = step(agc, dataRx); % AGC
            outChanFilt = step(channelFilter, outAGC); % Adjacent channel filtering
            rxAmp = mean(abs(outChanFilt));
            if rxAmp > frsFDRxParams.DetectionThreshold
                outThreshold = outChanFilt;
            else
                outThreshold = complex(single(zeros(frsFDRxParams.RadioFrameLength, 1)));
            end

            % FM demodulation and decimation
            outFMDemod = step(fmDemod, outThreshold); % FM demodulate
            outDecim = step(decimator, outFMDemod); % Resample to 8 kHz

            % CTCSS decode and conditionally send to audio output
            rcvdCode = step(decoder, outDecim);
            if (rcvdCode == frsFDRxParams.CTCSSCode) || (frsFDRxParams.CTCSSCode == 0)
                rcvdSig = outDecim;
            else
                rcvdSig = single(zeros(frsFDRxParams.AudioFrameLength, 1));
            end

            audioSig = step(audioFilter, rcvdSig); % Filter out CTCSS tones
            step(audioPlayer, audioSig); % Audio output
            timeCounter = timeCounter + frsFDRxParams.RadioFrameTime;
        end
    end
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end

% Release all SDRu and audio resources, FM Modulator and Demodulator
release(fmMod)
release(radioTx)
release(fmDemod)
release(radioRx)
release(audioPlayer)
```

## Conclusion

In this example, you used Communications Toolbox™ System objects to perform full duplex transmission and reception using FRS waveforms and two USRP® radios.

## Appendix

The following functions are used in this example.

- `configureFDTx.m`
- `configureFDRx.m`
- `sdruFRSFullDuplexTransceiver_2.m`

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## FM Receiver with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® (USRP®) device with MATLAB® to build an FM broadcast receiver.

In order to run this example, you need a USRP® board with an appropriate receiver daughterboard that supports the FM band (e.g., TVRX or WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object™.

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdrdru` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdrdru;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    radioFound = true;
    platform = connectedRadios(1).Platform;
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
        case {'N200/N210/USRP2','X300','X310','N300','N310','N320/N321'}
            address = connectedRadios(1).IPAddress;
    end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end
```

### Initialization

The `getParamsSdruFMEExamples.m` function initialize some simulation parameters and generates a structure, `fmRxParams`. The fields of this structure are the parameters of the FM receiver system at hand.

```
fmRxParams = getParamsSdruFMEExamples(platform)
```

### Configuration of Receiver Object

The script communicates with the USRP® board using the SDRu System object. B200 and B210 series USRP® radios are addressed using a serial number while USRP2, N200, N210, X300 and X310 radios are addressed using an IP address. Master clock rate is configurable for B2xx and X3xx series radios.

```
% Set up radio object to use the found radio
switch platform
    case {'B200','B210'}
        radio = comm.SDRuReceiver(...
            'Platform', platform, ...
            'SerialNum', address, ...
            'MasterClockRate', fmRxParams.RadioMasterClockRate);
    case {'X300','X310','N300','N310','N320/N321'}
```

```

radio = comm.SDRuReceiver(...
    'Platform', platform, ...
    'IPAddress', address, ...
    'MasterClockRate', fmRxParams.RadioMasterClockRate);
case {'N200/N210/USRP2'}
    radio = comm.SDRuReceiver(...
        'Platform', platform, ...
        'IPAddress', address);
end

```

Set the center frequency to 102.5 MHz and the gain to 30 dB. Set the master clock rate and decimation factor to obtain a sample rate of 200 kHz at the output of the SDRu receiver object. For example, for a B210 radio, set MasterClockRate to 20 MHz and DecimationRate to 100. For N200, N210, and USRP2 radios master clock rate is fixed at 100 MHz. The 200 kHz sample rate is not aliased the audio signal, however, it enables real time operation on desktop. Set the frame length to 4000 samples. Select the output data type as single to reduce the required memory and speed up execution.

```

radio.CenterFrequency = 102.5e6;
radio.Gain = fmRxParams.RadioGain;
radio.DecimationFactor = fmRxParams.RadioDecimationFactor;
radio.SamplesPerFrame = fmRxParams.RadioFrameLength;
radio.OutputDataType = 'single'

```

You can obtain information about the daughterboard using the info method of the object. This method returns a structure with fields that specify the valid range of SDRu properties. You can verify that the daughterboard covers the FM broadcast frequency range, which is 88 MHz to 108 MHz.

```
hwInfo = info(radio)
```

## FM Demodulation

This example uses the FM Broadcast Demodulator Baseband System object to demodulate the received signal. The block also converts the sampling rate of 240 kHz to 48 kHz, a native sampling rate for your host computer's audio device. According to the FM broadcast standard in the United States, the deemphasis lowpass filter time constant is set to 75 microseconds. Set up the demodulator to process stereo signals. The demodulator can also process the signals in a mono fashion.

```

fmBroadcastDemod = comm.FMBroadcastDemodulator(...
    'SampleRate', fmRxParams.RadioSampleRate, ...
    'FrequencyDeviation', fmRxParams.FrequencyDeviation, ...
    'FilterTimeConstant', fmRxParams.FilterTimeConstant, ...
    'AudioSampleRate', fmRxParams.AudioSampleRate, ...
    'PlaySound', true, ...
    'BufferSize', fmRxParams.BufferSize, ...
    'Stereo', true);

```

To perform stereo decoding, the FM Broadcast Demodulator Baseband block uses a peaking filter which picks out the 19 kHz pilot tone from which the 38 kHz carrier is created. Using the obtained carrier signal, the FM Broadcast Demodulator Baseband block downconverts the L-R signal, centered at 38 kHz, to baseband. Afterwards, the L-R and L+R signals pass through a 75 microsecond deemphasis filter. The FM Broadcast Demodulator Baseband block separates the L and R signals and converts them to the 48 kHz audio signal.

## Stream Processing Loop

Capture FM signals and apply FM demodulation for 10 seconds, which is specified by fmRxParams.StopTime. The SDRu object returns a column vector, x. Because the MATLAB script may

run faster than the hardware, the object also returns the actual size of the valid data in `x` using the second output argument, `len`. If `len` is zero, then there is no new data for the demodulator code to process. The demodulator downconverts the sampling rate to 152 kHz and then performs FM demodulation. The stereo decoder extracts the 19 kHz pilot signal and generates a 38 kHz reference tone to convert the L-R channel to baseband. After compensating for the loss in the L-R channel, both L-R and L+R channels pass through a rate converter and deemphasis filter. The resulting 48 kHz signal is separated into the left and right channels and sent to the audio device.

Check for the status of the USRP® radio

```
if radioFound
    % Loop until the example reaches the target stop time, which is 10
    % seconds.
    timeCounter = 0;
    while timeCounter < fmRxParams.StopTime
        [x, len] = step(radio);
        if len > 0
            % FM demodulation
            step(fmBroadcastDemod, x);
            % Update counter
            timeCounter = timeCounter + fmRxParams.AudioFrameTime;
        end
    end
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end
```

Release the FM Broadcast Demodulator Baseband and USRP® resources.

```
release(fmBroadcastDemod)
release(radio)
```

## Conclusion

In this example, you used Communications Toolbox™ System objects to build an FM receiver utilizing the USRP® device. The example showed that the MATLAB script can process signals captured by the USRP® device in real time.

## Further Exploration

To further explore the example, you can vary the center frequency of the USRP® device and listen to other radio stations.

If you have your own FM transmitter that can transmit .wma files, you can duplicate the test that shows the channel separation result above. Load the `sdruFMStereoTestSignal.wma` file into your transmitter. The channel separation can be easily observed from the spectrum and heard from the audio device. You can also adjust the gain compensation to see its effect on stereo separation.

You can set the `Stereo` property of the FM demodulator object to `false` to process the signals in a mono fashion and compare the sound quality.

## Appendix

The following function is used in this example.

- `getParamsSdruFMExamples.m`

### **Selected Bibliography**

- FM broadcasting on Wikipedia

### **Copyright Notice**

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## Frequency Offset Calibration Receiver with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® devices exploiting SDRu (Software Defined Radio USRP®) System objects to measure and calibrate for transmitter/receiver frequency offset at the receiver using MATLAB®.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object.

The USRP® Transmitter sends a sine wave at 100Hz with the MATLAB script, `sdrufreqcalibTransmitter.m`, to the USRP® receiver. The USRP® Receiver monitors received signals, calculates the transmitter/receiver frequency offset and displays it in the MATLAB command window for calibration with the MATLAB script, `sdrufreqcalibReceiver.m`.

### Introduction

The example provides the following information about the USRP® transmitter/receiver link:

- The quantitative value of the frequency offset
- A graphical view of the spur-free dynamic range of the receiver
- A graphical view of the qualitative SNR level of the received signal

To calibrate the frequency offset between two USRP® devices, run `sdrufreqcalibTransmitter.m` on one USRP® radio, and while simultaneously running `sdrufreqcalibReceiver.m` on another USRP® radio. The `CenterFrequency` property of the SDRu transmitter and receiver System objects should have the same value.

To compensate for a transmitter/receiver frequency offset, add the displayed frequency offset to the Center Frequency of the SDRu Receiver System object. Be sure to use the sign of the offset in your addition. Once you've done that, the spectrum displayed by the receiver's spectrum analyzer System object should have its maximum amplitude at roughly 0 Hz.

Please refer to the Simulink® model `sdrufreqcalib_rx.mdl` for a block diagram view of the system.

### Hardware Requirements

To run this example, ensure that the center frequency of the SDRu Transmitter and Receiver System objects is within the acceptable range of your USRP® daughter board and the antennas you are using.

### Code Architecture

The Frequency Offset Calibration Receiver MATLAB script, `sdrufreqcalibReceiver.m`, uses three System objects: `comm.SDRuReceiver`, a coarse frequency offset object, and a `dsp.SpectrumAnalyzer` to show the power spectral density of the received signal.

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdrdru` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdrdru;
if strncmp(connectedRadios(1).Status, 'Success', 7)
```

```

radioFound = true;
switch connectedRadios(1).Platform
    case {'B200','B210'}
        address = connectedRadios(1).SerialNum;
        platform = connectedRadios(1).Platform;
    case {'N200/N210/USRP2'}
        address = connectedRadios(1).IPAddress;
        platform = 'N200/N210/USRP2';
    case {'X300','X310'}
        address = connectedRadios(1).IPAddress;
        platform = connectedRadios(1).Platform;
    end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end

```

## Initialization

Baseband and RF configuration

```

rfRxFreq           = 1.85e9; % Nominal RF receive center frequency
bbRxFreq          = 100;    % Received baseband sine wave frequency

prmFreqCalibRx = configureFreqCalibRx(platform, rfRxFreq, bbRxFreq);

% This example communicates with the USRP(R) board using the SDRu receiver
% System object. B200 and B210 series USRP(R) radios are addressed using a
% serial number while USRP2, N200, N210, X300 and X310 radios are addressed
% using an IP address. The parameter structure, prmFreqCalibRx, sets the
% CenterFrequency, Gain, InterpolationFactor, and SamplesPerFrame
% arguments.
switch platform
    case {'B200','B210'}
        radio = comm.SDRuReceiver(...,
            'Platform',      platform, ...
            'SerialNum',     address, ...
            'MasterClockRate', prmFreqCalibRx.MasterClockRate, ...
            'CenterFrequency', prmFreqCalibRx.RxCenterFrequency, ...
            'Gain',          prmFreqCalibRx.Gain, ...
            'DecimationFactor', prmFreqCalibRx.DecimationFactor, ...
            'SamplesPerFrame',   prmFreqCalibRx.FrameLength, ...
            'OutputDataType',  prmFreqCalibRx.OutputDataType)
    case {'X300','X310'}
        radio = comm.SDRuReceiver(...,
            'Platform',      platform, ...
            'IPAddress',     address, ...
            'MasterClockRate', prmFreqCalibRx.MasterClockRate, ...
            'CenterFrequency', prmFreqCalibRx.RxCenterFrequency, ...
            'Gain',          prmFreqCalibRx.Gain, ...
            'DecimationFactor', prmFreqCalibRx.DecimationFactor, ...
            'SamplesPerFrame',   prmFreqCalibRx.FrameLength, ...
            'OutputDataType',  prmFreqCalibRx.OutputDataType)
    case {'N200/N210/USRP2'}
        radio = comm.SDRuReceiver(...,
            'Platform',      platform, ...
            'IPAddress',     address, ...

```

```
'MasterClockRate', prmFreqCalibRx.MasterClockRate, ...
'CenterFrequency', prmFreqCalibRx.RxCenterFrequency, ...
'Gain', prmFreqCalibRx.Gain, ...
'DecimationFactor', prmFreqCalibRx.DecimationFactor, ...
'SamplesPerFrame', prmFreqCalibRx.FrameLength, ...
'OutputDataType', prmFreqCalibRx.OutputDataType)
end

% Create a coarse frequency offset compensation System object to calculate
% the offset. The System object performs an FFT on its input signal and
% finds the frequency of maximum power. This quantity is the frequency
% offset.
CF0 = comm.CoarseFrequencyCompensator(
    'FrequencyResolution', 25, ...
    'SampleRate', prmFreqCalibRx.Fs);

spectrumAnalyzer = dsp.SpectrumAnalyzer(
    'Name', 'Actual Frequency Offset', ...
    'Title', 'Actual Frequency Offset', ...
    'SpectrumType', 'Power density', ...
    'FrequencySpan', 'Full', ...
    'SampleRate', prmFreqCalibRx.Fs, ...
    'YLimits', [-130, -20], ...
    'SpectralAverages', 50, ...
    'FrequencySpan', 'Start and stop frequencies', ...
    'StartFrequency', -100e3, ...
    'StopFrequency', 100e3, ...
    'Position', figposition([50 30 30 40]));

radio =
System: comm.SDRuReceiver

Properties:
    Platform: 'B210'
    SerialNum: 'F5BA6A'
    ChannelMapping: 1
    CenterFrequency: 1850000000
    LocalOscillatorOffset: 0
        Gain: 30
        PPSSource: 'Internal'
        ClockSource: 'Internal'
        MasterClockRate: 20000000
        DecimationFactor: 100
        TransportDataType: 'int16'
        OutputDataType: 'double'
        SamplesPerFrame: 4096
        EnableBurstMode: false
```

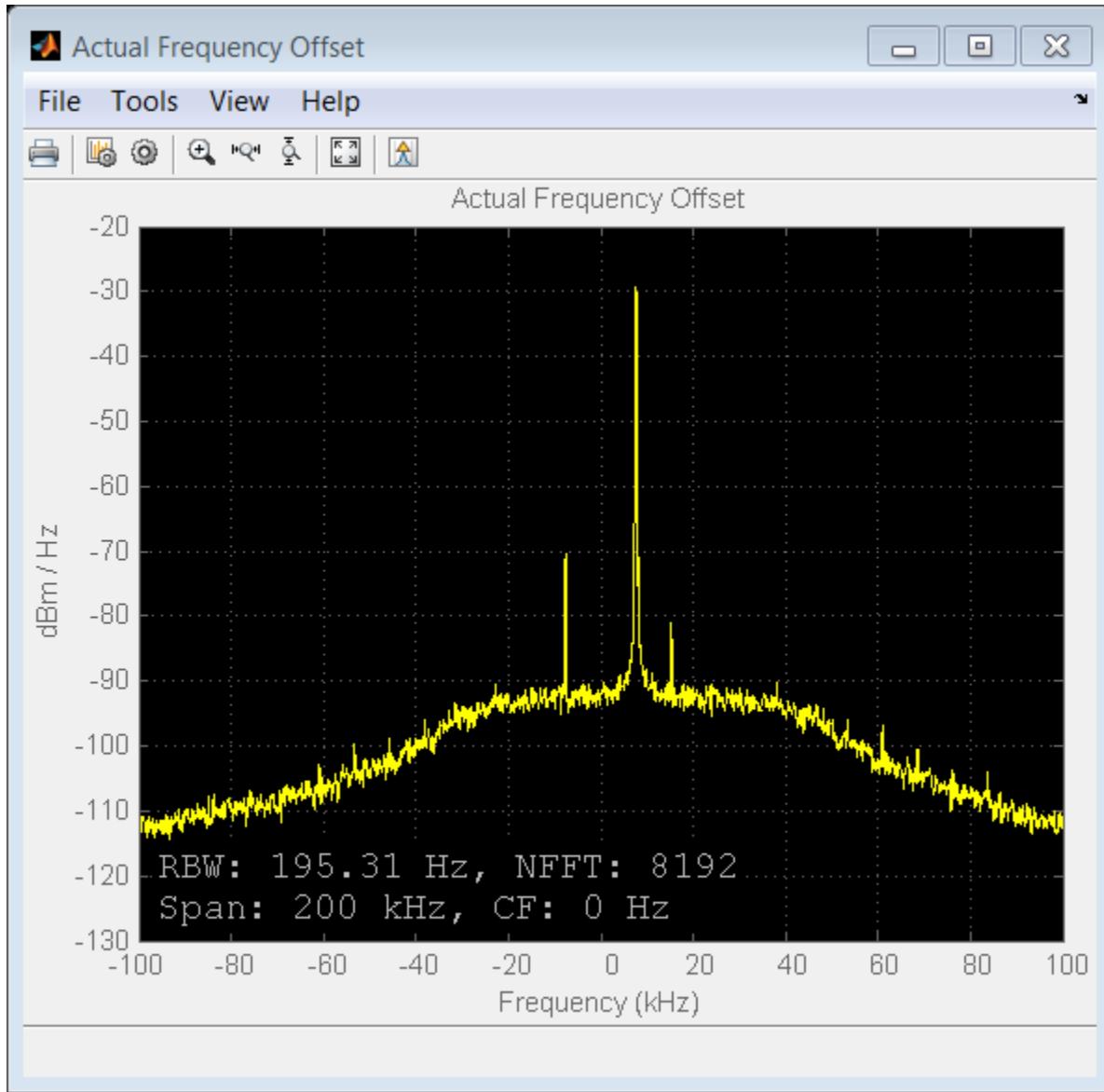
## Stream Processing

Loop until the example reaches the target number of frames.

```
% Check for the status of the USRP(R) radio
if radioFound
    for iFrame = 1 : prmFreqCalibRx.TotalFrames
        [rxSig, len] = radio();
```

```
if len > 0
    % Display received frequency spectrum.
    spectrumAnalyzer(rxSig);
    % Compute the frequency offset.
    [~, offset] = CF0(rxSig);
    % Print the frequency offset compensation value in MATLAB command
    % window.
    offsetCompensationValue = -offset
end
end
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end

% Release all System objects
release(radio);
clear radio
release(CF0);
```



## Conclusion

In this example, you used Communications Toolbox™ System objects to build a receiver that calculates the relative frequency offset between a USRP® transmitter and a USRP® receiver.

## Appendix

The following scripts are used in this example.

- `configureFreqCalibRx.m`

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

# Frequency Offset Calibration Transmitter with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® devices exploiting SDRu (Software Defined Radio USRP®) System objects to measure and calibrate for transmitter/receiver frequency offset at the receiver using MATLAB®.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter System object.

The USRP® Transmitter sends a sine wave at 100Hz with the MATLAB script, `sdrufreqcalibTransmitter.m`, to the USRP® receiver. The USRP® Receiver monitors received signals, calculates the transmitter/receiver frequency offset and displays it in the MATLAB command window for calibration with the MATLAB script, `sdrufreqcalibReceiver.m`.

## Introduction

The example provides the following information about the USRP® transmitter/receiver link:

- The quantitative value of the frequency offset
- A graphical view of the spur-free dynamic range of the receiver
- A graphical view of the qualitative SNR level of the received signal

To calibrate the frequency offset between two USRP® devices, run `sdrufreqcalibTransmitter.m` on one USRP® radio, while simultaneously running `sdrufreqcalibReceiver.m` on another USRP® radio. The `CenterFrequency` property of the SDRu transmitter and receiver System objects should have the same value.

To compensate for a transmitter/receiver frequency offset, add the displayed frequency offset value to the Center Frequency of the SDRu Receiver System object. Be sure to use the sign of the offset in your addition. Once you've done that, the spectrum displayed by the receiver's spectrum analyzer System object should have its maximum amplitude at roughly 0 Hz.

Please refer to the Simulink® model `sdrufreqcalib.mdl` for a block diagram view of the system.

## Hardware Requirements

To run this example, ensure that the center frequency of the SDRu Transmitter and Receiver System objects is within the acceptable range of your USRP® daughter board and the antennas you are using.

## Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdrus` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdrus;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    radioFound = true;
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
```

```
platform = connectedRadios(1).Platform;
case {'N200/N210/USRP2'}
    address = connectedRadios(1).IPAddress;
    platform = 'N200/N210/USRP2';
case {'X300','X310'}
    address = connectedRadios(1).IPAddress;
    platform = connectedRadios(1).Platform;
end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end
```

## Initialization

Set the properties of the sine wave source, the SDRu transmitter, and the spectrum analyzer System object.

```
bbTxFreq = 100; % Transmitted baseband frequency
rfTxFreq = 1.85e9; % Nominal RF transmit center frequency

prmFreqCalibTx = configureFreqCalibTx(platform, rfTxFreq, bbTxFreq);

hSineSource = dsp.SineWave ( ...
    'Frequency', prmFreqCalibTx.SineFrequency, ...
    'Amplitude', prmFreqCalibTx.SineAmplitude, ...
    'ComplexOutput', prmFreqCalibTx.SineComplexOutput, ...
    'SampleRate', prmFreqCalibTx.Fs, ...
    'SamplesPerFrame', prmFreqCalibTx.SineFrameLength, ...
    'OutputDataType', prmFreqCalibTx.SineOutputDataType);

% The host computer communicates with the USRP(R) radio using the SDRu
% transmitter System object. B200 and B210 series USRP(R) radios are
% addressed using a serial number while USRP2, N200, N210, X300 and X310
% radios are addressed using an IP address. The parameter structure,
% prmFreqCalibTx, sets the CenterFrequency, Gain, and InterpolationFactor
% arguments.

% Set up radio object to use the found radio
switch platform
case {'B200','B210'}
    radio = comm.SDRuTransmitter( ...
        'Platform', platform, ...
        'SerialNum', address, ...
        'MasterClockRate', prmFreqCalibTx.MasterClockRate, ...
        'CenterFrequency', prmFreqCalibTx.USRPtxCenterFrequency, ...
        'Gain', prmFreqCalibTx.USRPGain, ...
        'InterpolationFactor', prmFreqCalibTx.USRPInterpolationFactor)
case {'X300','X310'}
    radio = comm.SDRuTransmitter( ...
        'Platform', platform, ...
        'IPAddress', address, ...
        'MasterClockRate', prmFreqCalibTx.MasterClockRate, ...
        'CenterFrequency', prmFreqCalibTx.USRPtxCenterFrequency, ...
        'Gain', prmFreqCalibTx.USRPGain, ...
        'InterpolationFactor', prmFreqCalibTx.USRPInterpolationFactor)
```

```

case {'N200/N210/USRP2'}
radio = comm.SDRuTransmitter( ...
    'Platform', platform, ...
    'IPAddress', address, ...
    'CenterFrequency', prmFreqCalibTx.USRPTxCenterFrequency, ...
    'Gain', prmFreqCalibTx.USRPGain, ...
    'InterpolationFactor', prmFreqCalibTx.USRPInterpolationFactor)
end

% Use dsp.SpectrumAnalyzer to display the spectrum of the transmitted
% signal.
hSpectrumAnalyzer = dsp.SpectrumAnalyzer(...
    'Name', 'Frequency of the Sine waveform sent out',...
    'Title', 'Frequency of the Sine waveform sent out',...
    'FrequencySpan', 'Full', ...
    'SampleRate', prmFreqCalibTx.Fs, ...
    'YLimits', [-70,30],...
    'SpectralAverages', 50, ...
    'FrequencySpan', 'Start and stop frequencies', ...
    'StartFrequency', -100e3, ...
    'StopFrequency', 100e3, ...
    'Position', figposition([50 30 30 40]));

```

radio =

System: comm.SDRuTransmitter

Properties:

- Platform: 'B210'
- SerialNum: 'F5BA6A'
- ChannelMapping: 1
- CenterFrequency: 1850000000
- LocalOscillatorOffset: 0
- Gain: 23
- PPSSource: 'Internal'
- ClockSource: 'Internal'
- MasterClockRate: 20000000
- InterpolationFactor: 100
- TransportDataType: 'int16'
- EnableBurstMode: false

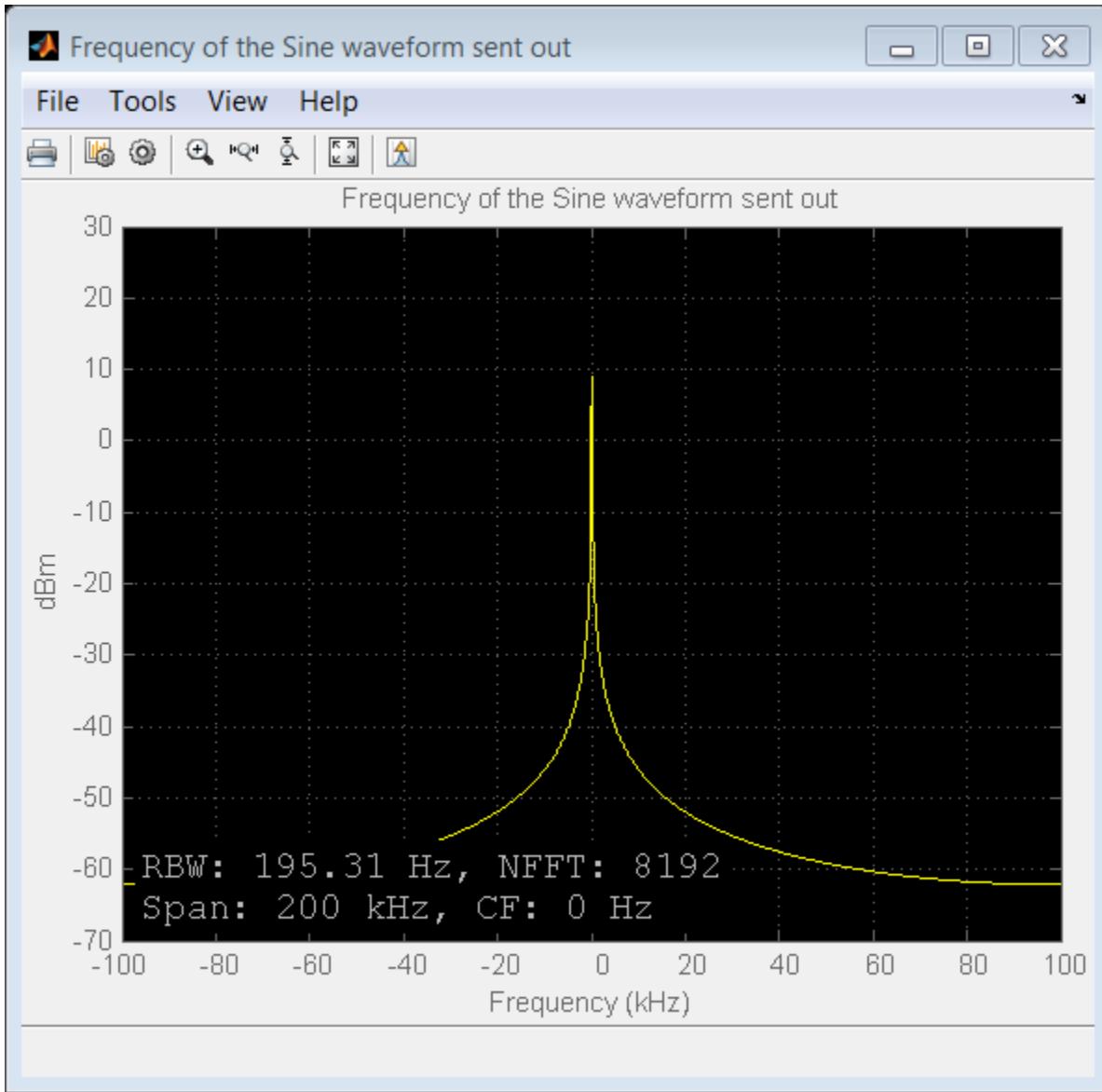
## Stream Processing

Loop until the example reaches the target number of frames.

```

% Check for the status of the USRP(R) radio
if radioFound
    for iFrame = 1: prmFreqCalibTx.TotalFrames
        sinewave = step(hSineSource); % generate sine wave
        step(radio, sinewave); % transmit to USRP(R) radio
    end
    % Display the spectrum after the simulation.
    step(hSpectrumAnalyzer, sinewave);
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end

```



### Release System Objects

```
release (hSineSource);
release (radio);
clear radio
```

### Conclusion

In this example, you used Communications Toolbox™ System objects to build a signal source to send a reference tone at 100 Hz. This signal is to be used as a calibration signal for a USRP® receiver.

### Appendix

The following scripts are used in this example.

- configureFreqCalibTx.m

### **Copyright Notice**

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## FRS/GMRS Walkie-Talkie Receiver with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® (USRP®) device with MATLAB® to implement a walkie-talkie receiver. The specific radio standard that this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System). You can transmit a signal to the implemented receiver using a commercial walkie-talkie device.

In order to run this example, you need a USRP® board with an appropriate receiver daughterboard that supports the UHF 462-467 MHz band (for example, WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object™.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found at [ 1 ] and [ 2 ]. Operation in other countries may or may not work.

### Overview

Please refer to the FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware example for general information and overview details. Note that all the information in that section applies to this example, except that this example is designed to receive signals instead of transmit them.

Also, please refer to the Simulink® model in the FRS/GMRS Walkie-Talkie Receiver with USRP® Hardware example for a block diagram view of the system.

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdr` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdr;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    radioFound = true;
    platform = connectedRadios(1).Platform;
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
        case {'N200/N210/USRP2','X300','X310'}
            address = connectedRadios(1).IPAddress;
    end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end
```

Checking radio connections...

### Initialization

The `getParamsSdrFRSGMRSRxDemo.m` function initializes some simulation parameters and generates a structure, `frsRx`. The fields of this structure are the parameters of the FRS/GMRS receiver system at hand.

```
% Configure the example to receive on channel 12 with the CTCSS code 5 with
% a detection threshold of 0.1.
channel = 12;
CTCSSCode = 5;
detectionThreshold = 0.1;

frsRxParams = getParamsSdruFRSGMRSRxDemo(platform, channel, CTCSSCode, ...
    detectionThreshold)

frsRxParams =
    RadioMasterClockRate: 20000000
        RadioSampleRate: 200000
            RadioGain: 5
    RadioDecimationFactor: 100
        RadioFrameLength: 4000
            ChannelFilter: [1x1 dsp.FIRFilter]
    ChannelFilterNumerator: [1x33 double]
        FrequencyDeviation: 2500
            DecimationFactor: 25
    DecimationNumerator: [1x600 double]
        AudioSampleRate: 8000
        AudioFrameLength: 160
            Channel: 12
            CTCSSCode: 5
        DetectionThreshold: 0.1000
    CTCSSDecodeBlockLength: 4000
    CTCSSSToneFrequencies: [38x1 double]
    GoertzelCoefficients: [38x1 double]
        AudioFilter: [1x1 dsp.FIRFilter]
        StopTime: 10
    RadioFrameTime: 0.0200
```

## FRS/GMRS Receiver

The FRS/GMRS receiver example tunes the USRP® board to receive at the center frequency specified by the channel selection. The script applies automatic gain control (AGC) and FM demodulates the received signal. A CTCSS tone decoder passes the demodulated signal to an audio device if the received code matches the selected code.

### AGC and Channel Selectivity Filter

Automatic gain controller applies a variable gain to the received signal to assure that the received signal amplitude is at a known level. In this example, the walkie-talkie transmitter is likely nearby the USRP® board, which implies that the received signal should not suffer from fading, and the received signal-to-noise ratio (SNR) should be high. In practice, the received signals will likely suffer from fading and low SNR.

```
agc = comm.AGC;
```

This script uses a low pass channel separation filter to reduce the signals from an adjacent channel. The gap between adjacent channels is 25 kHz, which means the baseband bandwidth is at most 12.5 kHz. Thus, we choose the cutoff frequency to be 10 kHz. Create a digital filter System object that implements an FIR transfer function and set the Numerator property to the value specified in the frsRx structure.

```
channelFilter = frsRxParams.ChannelFilter;
```

Next, a channel selector computes the average power of the filtered signal. If it is greater than a threshold (set to a default of 10%), the channel selector determines that the received signal is from the correct channel and it allows the signal to pass through. In the case of an out-of-band signal, although the channel separation filter reduces its magnitude, it is still FM modulated and the modulating signal will be present after FM demodulation. To reject such a signal completely, the channel selector outputs zero.

### FM Demodulation and Decimation

This example uses the FM Demodulator Basesband System object whose sample rate and maximum frequency deviation are set to 200 kHz and 2.5 kHz, respectively.

```
fmDemod = comm.FMDemodulator('SampleRate', frsRxParams.RadioSampleRate, ...
    'FrequencyDeviation', frsRxParams.FrequencyDeviation);
```

A decimation filter converts the sampling rate to 8 kHz. This rate is one of the native sampling rates of your host computer's output audio device. Use an FIR decimator System object to convert the 200 kHz signal to an 8 kHz signal. Set the decimation factor to 25, and the numerator to the value specified in the frsRx structure.

```
decimator = dsp.FIRDecimator(frsRxParams.DecimationFactor, ...
    frsRxParams.DecimationNumerator);
```

### Continuous Tone-Coded Squelch System (CTCSS)

The CTCSS [ 3 ] decoder computes the power at each CTCSS tone frequency using the Goertzel algorithm [ 4 ] and outputs the code with the largest power. The Goertzel algorithm provides an efficient method to compute the frequency components at predetermined frequencies, i.e., the tone code frequencies used by FRS/GMRS.

The script compares the estimated received code with the preselected code and then sends the signal to the audio device if the two codes match. When the preselected code is zero, it indicates no squelch system is used and the decision block passes the signal at the channel to the audio device no matter which code is used.

```
decoder = FRSGMRSDemoCTCSSDecoder(...
    'MinimumBlockLength', frsRxParams.CTCSSDecodeBlockLength, ...
    'SampleRate', frsRxParams.AudioSampleRate);
```

### Audio Output

A high pass filter with a cutoff frequency of 260 Hz filters out the CTCSS tones, which have a maximum frequency of 250 Hz. Use an audio device writer System object to play the received signals through your computer's speakers. If you do not hear any sound, please select another device using the Device property of the audio player object, audioPlayer.

```
audioFilter = frsRxParams.AudioFilter;
audioPlayer = audioDeviceWriter(frsRxParams.AudioSampleRate);
```

### Configuration of Receiver Object

The script communicates with the USRP® board using the SDRu receiver System object. B200 and B210 series USRP® radios are addressed using a serial number while USRP2, N200, N210, X300 and X310 radios are addressed using an IP address.

```
% Setup radio object to use the found radio
switch platform
    case {'B200','B210'}
        radio = comm.SDRuReceiver('Platform', platform, ...
            'SerialNum', address, ...
            'MasterClockRate', frsRxParams.RadioMasterClockRate);
    case {'X300','X310'}
        radio = comm.SDRuReceiver('Platform', platform, ...
            'IPAddress', address, ...
            'MasterClockRate', frsRxParams.RadioMasterClockRate);
    case {'N200/N210/USRP2'}
        radio = comm.SDRuReceiver('Platform', platform, ...
            'IPAddress', address);
end
```

Set the master clock rate and decimation factor to obtain a sample rate of 200 kHz at the output of the SDRu receiver object. For example, for a B210 radio, set MasterClockRate to 20 MHz and DecimationRate to 100. For N200, N210, and USRP2 radios master clock rate is fixed at 100 MHz. The 200 kHz sample rate enables us to use a simple decimation filter to convert the sampling rate from 200 kHz to 8 kHz. Frame length controls the number of samples at the output of the SDRu receiver, which is the input to the AGC. The frame length must be an integer multiple of the decimation factor, which is 25. Set the frame length to 4000 samples. Select the output data type as single to reduce the required memory and speed up execution. Set the center frequency source to input port and the gain to 5 dB. Note that, for X3xx series radios, these numbers may vary to match the valid radio hardware values.

```
radio.CenterFrequencySource = 'Input port';
radio.Gain = frsRxParams.RadioGain;
radio.DecimationFactor = frsRxParams.RadioDecimationFactor;
radio.SamplesPerFrame = frsRxParams.RadioFrameLength;
radio.OutputDataType = 'single';

% Display SDRu receiver object
radio

radio =
System: comm.SDRuReceiver

Properties:
    Platform: 'B210'
    SerialNum: 'F5BA51'
    ChannelMapping: 1
    LocalOscillatorOffset: 0
    Gain: 5
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 20000000
    DecimationFactor: 100
    TransportDataType: 'int16'
    OutputDataType: 'single'
    SamplesPerFrame: 4000
    EnableBurstMode: false
```

## Running the Example

Turn on your walkie-talkie, set the channel to 12 and the private code to 5. The center frequency is a function of the selected channel number.

```
% Get the carrier frequency for the selected channel
fc = convertChan2FreqFRSGMRSDemo(frsRxParams.Channel);
```

## Stream Processing Loop

Capture FRS/GMRS signals and demodulate for 10 seconds, which is specified by frsRx.StopTime. The SDRu object returns a column vector, x. Because the MATLAB script may run faster than the hardware, the object also returns the actual size of the valid data in x using the second output argument, len. If len is zero, then there is no new data for the demodulator code to process.

Check for the status of the USRP® radio

```
if radioFound
    % Loop until the example reaches the target stop time.
    timeCounter = 0;
    while timeCounter < frsRxParams.StopTime

        [data, len] = step(radio, fc);
        if len > 0

            % AGC and channel selectivity
            outAGC = step(agc, data);

            outChanFilt = step(channelFilter, outAGC);
            rxAmp = mean(abs(outChanFilt));
            if rxAmp > frsRxParams.DetectionThreshold
                x = outChanFilt;
            else
                x = complex(single(zeros(frsRxParams.RadioFrameLength, 1)));
            end

            % FM demodulator and decimation
            y = step(fmDemod, x);
            outRC = step(decimator, y);

            % CTCSS decoder
            rcvCode = step(decoder, outRC);
            if (rcvCode == frsRxParams.CTCSSCode) || (frsRxParams.CTCSSCode == 0)
                rcvSig = outRC;
            else
                rcvSig = single(zeros(frsRxParams.AudioFrameLength, 1));
            end

            % Output to audio device
            audioSig = step(audioFilter, rcvSig);
            step(audioPlayer, audioSig);

            timeCounter = timeCounter + frsRxParams.RadioFrameTime;
        end
    end
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end
```

Release the audio , FM Demodulator Baseband and USRP® resources.

```
release(fmDemod);
release(audioPlayer);
release(radio);
```

## **Conclusion**

In this example, you used Communications Toolbox™ System objects to build an FRS/GMRS receiver utilizing the USRP® device. The example showed that the MATLAB script can process signals captured by the USRP® device in real time.

## **Further Exploration**

The CTCSS decoding computes the DFT (Discrete-Time Fourier Transform) of the incoming signal using the Goertzel algorithm and computes the power at the tone frequencies. Since the tone frequencies are very close to each other (only 3-4 Hz apart) the block length of the DFT should be large enough to provide enough resolution for the frequency analysis. However, long block lengths cause decoding delay. For example, a block length of 16000 will cause 2 seconds of delay because the CTCSS decoder operates at an 8 kHz sampling rate. This creates a trade-off between detection performance and processing latency. The optimal block length may depend on the quality of the transmitter and receiver, the distance between the transmitter and receiver, and other factors. You are encouraged to change the block length in the initialization function by navigating to the `getParamsSdruFRSGMRSRxDemo` function and changing the value of the `CTCSSDecodeBlockLength` field. This will enable you to observe the trade-off and find the optimal value for your transmitter/receiver pair.

## **Appendix**

The following function is used in this example.

- `getParamsSdruFRSGMRSRxDemo.m`

## **References**

- Family Radio Service on Wikipedia
- General Mobile Radio Service on Wikipedia
- Continuous Tone-Coded Squelch System on Wikipedia
- Goertzel Algorithm on Wikipedia

## **Copyright Notice**

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® (USRP®) device with MATLAB® to implement a walkie-talkie transmitter. The specific radio standard that this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System). You can listen to the transmitted signal using a commercial walkie-talkie device.

In order to run this example, you need a USRP® board with an appropriate transmitter daughterboard that supports the UHF 462-467 MHz band (for example, WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter System object™.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found at [ 1 ] and [ 2 ]. Operation in other countries may or may not work.

### Overview

Walkie-talkies provide a subscription-free method of communicating over short distances. Although their popularity has been diminished by the rise of cell phones, walkie-talkies are still useful when lack of reception or high per-minute charges hinders the cell phone use.

Modern walkie-talkies operate on the FRS/GMRS standards. Both standards use frequency modulation (FM) at 462 or 467 MHz, which is in the UHF (Ultra High Frequency) band. The USRP® device in this example will transmit messages at either 462 or 467 MHz, in a manner that is compatible with FRS/GMRS devices.

Please refer to the Simulink® model in the FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware example for a block diagram view of the system.

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdrdru` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system but does not run the main loop.

```
connectedRadios = findsdrdru;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    radioFound = true;
    platform = connectedRadios(1).Platform;
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
        case {'N200/N210/USRP2','X300','X310'}
            address = connectedRadios(1).IPAddress;
    end
else
    radioFound = false;
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end

Checking radio connections...
```

## Initialization

The `getParamsSdruFRSGMRSTxDemo.m` script initialize some simulation parameters and generates a structure, `frsTx`. The fields of this structure are the parameters of the FRS/GMRS transmitter system at hand.

```
% Configure the example to transmit on channel 12 with the CTCSS code 5.
channel = 12;
CTCSSCode = 5;

frsTxParams = getParamsSdruFRSGMRSTxDemo(platform, channel, CTCSSCode)

frsTxParams =
    RadioMasterClockRate: 20000000
        RadioSampleRate: 200000
            RadioGain: 15
    RadioInterpolationFactor: 100
        RadioFrameLength: 24000
        FrequencyDeviation: 2500
        InterpolationFactor: 25
    InterpolationNumerator: [1x600 double]
        SourceSampleRate: 8000
        SourceFrameLength: 960
            ToneFrequency: 880
            ChirpSweeptime: 1
            ChirpInitialFreq: 600
            ChirpTargetFreq: 1200
    AudioInterpolationFactor: 160
        AudioDecimationFactor: 441
        AudioFrameLength: 2646
            Channel: 12
            CTCSSCode: 5
            CTCSSAmplitude: 0.1500
        CTCSSToneFrequencies: [38x1 double]
            StopTime: 10
        SourceFrameTime: 0.1200
```

## FRS/GMRS Transmitter

The FRS/GMRS transmitter combines signals from a source and CTCSS tone generator. The script `FM` modulates the combined signal and sends to the USRP® board to transmit over the air.

### Source Signals

This example uses an `FRSGMRSDemoSource` System object to generate data signals for the transmitter. The source signal can be a pure tone sine wave, a chirp signal, or a multimedia file. To switch between these sources, you can change the `Signal` property of `hSource`. The source object also allows you to set the pure tone frequency or the chirp signal target/sweep time (which controls the duration of the chirp signal). This example works properly with tones as low as 500 Hz and as high as 4 kHz. When using a multimedia file, the sampling rate needs to be converted to 8 kHz; therefore, the `FRSGMRSDemo AudioSource` System object class employs a rate conversion filter to convert the 22.5 kHz signal to an 8 kHz signal.

```
% Create a data source to transmit the contents of a sound file at a
% sampling frequency of 8 kHz.
```

```
source = FRSGMRSDemoSource('Sound file', frsTxParams.SourceSampleRate, ...
    'SamplesPerFrame', frsTxParams.SourceFrameLength);
```

### Continuous Tone-Coded Squelch System (CTCSS)

Walkie-Talkies operate on a shared public channel, allowing multiple users to access the same channel simultaneously. The CTCSS [ 3 ] method filters out undesired communication or interference from these other users by generating a tone between 67 Hz and 250 Hz and transmitting it along with the source signal. The receiver contains logic to detect this tone, and acknowledges a message if the detected tone matches the code setting on the receiver. The receiver filters out the tone so that the user does not hear it.

The CTCSS tone generator generates a continuous phase sine wave with a frequency corresponding to the selected private code. The amplitude of the tone is usually 10%-15% of the maximum amplitude of the modulating signal. Note that because the maximum amplitude of all the source signals is 1, the default amplitude of 0.15 for the CTCSS tone corresponds to 15% of the modulating signal's maximum amplitude.

```
ctcss = dsp.SineWave(frsTxParams.CTCSSToneAmplitude, ...
    frsTxParams.CTCSSToneFrequencies(frsTxParams.CTCSSToneCode), ...
    'SampleRate', frsTxParams.SourceSampleRate, ...
    'SamplesPerFrame', frsTxParams.SourceFrameLength, ...
    'OutputDataType', 'single');
```

### Interpolator and FM Modulator

The interpolator converts the sampling rate of the sum of the modulating signal and the CTCSS tone to match the SDR hardware sampling rate of 200 kHz. The resampling filter is designed using the FIRInterpolator object from the DSP System Toolbox™.

```
interpolator = dsp.FIRInterpolator(frsTxParams.InterpolationFactor, ...
    frsTxParams.InterpolationNumerator);
```

This example uses the FM Modulator Baseband System object whose sample rate and maximum frequency deviations are set to 200 kHz and 2.5 kHz, respectively.

```
fmMod = comm.FMModulator('SampleRate', frsTxParams.RadioSampleRate, ...
    'FrequencyDeviation', frsTxParams.FrequencyDeviation);
```

### Configuration of Receiver Object

The script communicates with the USRP® board using the SDRu transmitter System object. B200 and B210 series USRP® radios are addressed using a serial number while USRP2, N200, N210, X300 and X310 radios are addressed using an IP address.

```
% Setup radio object to use the found radio
switch platform
    case {'B200','B210'}
        radio = comm.SDRuTransmitter(...
            'Platform', platform, ...
            'SerialNum', address, ...
            'MasterClockRate', frsTxParams.RadioMasterClockRate);
    case {'X300','X310'}
        radio = comm.SDRuTransmitter(...
            'Platform', platform, ...
            'IPAddress', address, ...
```

```

'MasterClockRate', frsTxParams.RadioMasterClockRate);
case {'N200/N210/USRP2'}
    radio = comm.SDRuTransmitter(...
        'Platform', platform, ...
        'IPAddress', address);
end

```

Set the master clock rate and interpolation factor to obtain a sample rate of 200 kHz at the output of the SDRu receiver object. For example, for a B210 radio, set MasterClockRate to 20 MHz and DecimationRate to 100. For N200, N210, and USRP2 radios master clock rate is fixed at 100 MHz. The 200 kHz sample rate enables us to use a simple interpolation filter to convert the sampling rate from 8 kHz to 200 kHz. This example configures the *radio* object to accept the center frequency as an input argument. Set the gain to 30 dB.

```

radio.CenterFrequencySource = 'Input port';
radio.Gain = frsTxParams.RadioGain;
radio.InterpolationFactor = frsTxParams.RadioInterpolationFactor;

```

```
% Display SDRu receiver object
radio
```

```

radio =
System: comm.SDRuTransmitter
Properties:
    Platform: 'B210'
    SerialNum: 'F5BA51'
    ChannelMapping: 1
    LocalOscillatorOffset: 0
        Gain: 15
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 20000000
    InterpolationFactor: 100
    TransportDataType: 'int16'
    EnableBurstMode: false

```

You can obtain the information about the daughterboard using the *info* method of the object. This method returns a structure with fields that specify the valid range of SDRu properties. You can verify that the daughterboard covers the UHF frequency range, which is 462 MHZ to 467 MHz.

```

hwInfo = info(radio)

hwInfo =
    Mboard: 'B210'
    RXSubdev: 'FE-RX2'
    TXSubdev: 'FE-TX2'
    MinimumCenterFrequency: 40000000
    MaximumCenterFrequency: 6.0100e+09
        MinimumGain: 0
        MaximumGain: 89.7500
            GainStep: 0.2500
    CenterFrequency: 0

```

```
LocalOscillatorOffset: 0
    Gain: 15
MasterClockRate: 20000000
InterpolationFactor: 100
BasebandSampleRate: 200000
```

## Running the Example

Turn on your walkie-talkie, set the channel to 12 and the private code to 5. The center frequency is a function of the selected channel number.

```
% Get the carrier frequency for the selected channel
fc = convertChan2FreqFRSGMRSDemo(frsTxParams.Channel);
```

## Stream Processing Loop

Generate a data signal for the FRS/GMRS transmitter using the hSource object. Combine the data signal with the CTCSS tone, and pass them through a rate converter to generate a 200 kHz signal. FM modulate the resampled signal and send it to the USRP® board. The loop runs for 10 seconds and you should be able to hear the voice from your commercial walkie-talkie device.

Check for the status of the USRP® radio

```
if radioFound
    % Loop until the example reaches the target stop time.
    timeCounter = 0;
    while timeCounter < frsTxParams.StopTime

        data = step(source);
        dataWTone = data + step(ctcss);
        outResamp = step(interpolator, dataWTone);

        % FM modulator
        outMod = step(fmMod, outResamp);

        step(radio, outMod, fc);

        timeCounter = timeCounter + frsTxParams.SourceFrameTime;
    end
else
    warning(message('sdru:sysobjdemos:MainLoop'))
end
```

Release the USRP® resources and FM Modulator

```
release(fmMod)
release(radio)
```

If you cannot hear the voice, slightly increase the CTCSS amplitude parameter using the Amplitude property of the hCTCSS object, and try again. You can also edit the value of the CTCSSAmplitude field of the frsTxParams structure in getParamsSdruFRSGMRSTxDemo.m.

Instead of using a commercial walkie-talkie device, you can also run this example alongside an additional USRP® device running the FRS/GMRS receiver example.

## Conclusion

In this example, you used Communications Toolbox™ System objects to build an FRS/GMRS transmitter utilizing a USRP® device. This example showed that the MATLAB script can generate signals for the USRP® device in real time.

## Further Exploration

You can set your walkie-talkie channel to one of the 14 channels (numbered 1 to 14) and the private code to either one of the 38 private codes (numbered 1 to 38) or 0, in which case the squelch system is not used and all received messages are accepted. Note that the private codes above 38 are digital codes and are not implemented in this example. Set the channel and private code in the example so that they match the walkie-talkie.

Part 95.637 (Modulation standards) of the FCC wireless standards [ 4 ] state that the maximum frequency deviation (FD) is 2.5 kHz for FRS and 5 kHz for GMRS. In practice, it is usually set to 2.5 kHz for both systems. If the maximum signal amplitude increases, the frequency sensitivity parameter (K) should decrease. Otherwise, the receiving walkie-talkie will not decode the CTCSS code correctly. You can try to use a different signal with different values for frequency deviation to see if your walkie-talkie works properly. If the frequency deviation value is too large, you may not hear anything from your receiver when using a non-zero CTCSS private code. Set the CTCSS code to 0, which disables the squelch system. If you hear the transmitted signal, the CTCSS decoding of the non-zero code is incorrect.

You can reduce the amplitude of the CTCSS tone to determine the minimum amplitude required for your receiver to work correctly.

## Appendix

The following function and System object are used in this example.

- `getParamsSdruFRSGMRSTxDemo.m`
- `convertChan2FreqFRSGMRSDemo.m`

## References

- Family Radio Service on Wikipedia
- General Mobile Radio Service on Wikipedia
- Continuous Tone-Coded Squelch System on Wikipedia
- Part 95.637 (Modulation standards) of the FCC wireless standards

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

# IEEE® 802.11™ WLAN - OFDM Beacon Receiver with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral (USRP®) device using SDRu (Software Defined Radio USRP®) System objects™ to implement a WLAN receiver. The receiver is able to recover 802.11 OFDM Non-HT beacon frames [ 1 ] transmitted over the air from commercial 802.11 hardware. OFDM beacons can be transmitted from 802.11a/g/n/ac access points and are typically found in the 5 GHz band. Packet information such as SSID and MAC addresses are printed to the command line during recovery. This example assumes an access point is within range and transmits OFDM beacons in the desired channel.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object.

This example requires the WLAN Toolbox™.

## Introduction

This example has the following objectives:

- Receive signals from commercial WLAN transmitters in MATLAB® using SDRu System objects.
- Illustrate the use of the WLAN Toolbox and Communications Toolbox™ with real-world signals and radio hardware demonstrating full packet synchronization and decoding.

In this example, the SDRuReceiver System object receives data corrupted by the transmission over the air and outputs complex baseband signals which are processed by the WLAN RF Front End object. This example shows the WLAN signal recovery functions working together in a streaming arrangement to recover packets. It also shows how to decode the recovered medium access control (MAC) layer packets using WLAN Toolbox.

## Code Architecture

The function *runWLANNonHTReceiver* implements packet capture and synchronization using two System objects, *comm.SDRuReceiver* and *RFFrontEnd*. Once captured, the packet is decoded with WLAN Toolbox functions.

### SDRu Receiver

MATLAB communicates with the USRP® board using the SDRu receiver System object. The parameter structure *Config.RadioInfo* sets the hardware parameters such as *Gain*, *DecimationFactor*, and *MasterClockRate*.

### RF Front End

*nonHTFrontEnd* provides packet synchronization and collection. This happens in four stages:

- 1 **Packet Detection:** A packet must be detected before any processing begins. This is accomplished by auto-correlating input symbols. Since the beginning of each 802.11 OFDM packet contains a repetitive structure called the L-STF, peaks will occur in the correlation when this packet is present. The L-STF is then extracted and used for coarse frequency offset estimation.
- 1 **Symbol Timing:** Once a packet has been detected, several future symbols are captured into a buffer. This buffer is cross-correlated against to locate the L-LTF. Locating the L-LTF first provides fine symbol timing, identifying OFDM symbol boundaries for all successive symbols in

the packet. After the entire L-LTF is captured it is used for channel estimation and fine frequency offset estimation.

- 1 L-SIG Decoding: The first OFDM symbol after the L-LTF is the L-SIG field. This field must be recovered and decoded to determine the modulation, code rate, and length of the subsequent payload. The information is used to capture the correct amount of data after the L-SIG for a complete payload.
- 1 Payload Decoding: All OFDM symbols after the L-SIG are buffered to a length determined by the L-SIG field. After all the symbols have been captured, they are demodulated and decoded into their source bits. The source bits are then evaluated. This evaluation includes frame check sequence (FCS) validation and extraction of the header and body. If the packet is of subtype **beacon**, summary information will be printed about the recovered packet.

Once a full packet is received or any failures occur during the processing chain, the receiver will return to packet detection to search for more packets. This process is repeated for the duration of the requested capture time. The capture time is related to the amount of data pulled from the radio, not the runtime of the physical simulation.

### Check for presence of WLAN Toolbox

```
if isempty(ver('wlan'))
    error('Please install WLAN Toolbox to run this example.');
end
```

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the **findsdr** function. Check if the radio is available. Record the radio type and set configurations based on that type.

```
Config.RadioInfo = getRadioInfoWLANBeacon();
disp(Config.RadioInfo);
```

### Set Simulation Parameters

Request user input from the command-line for simulation parameters. You will be asked for 1) capture duration in seconds, 2) plotting status (on/off), 3) the amount of data to display from the recovered packets, 4) to enable vendor lookup for mac addresses, 5) band of interest, and 6) channels you want to scan. If vendor lookup is enabled this function will also download a MAC address lookup table, made publicly available by the IEEE.

```
Config.SimInfo = getUserInputWLANBeacon();
```

To find valid channel numbers in your geographic location, please refer to the documentation.

```
% Get channel number
reply = input(['What is the band you want to scan?\n', ...
    '1 == 5 GHz band (default)\n',...
    '2 == 2.4 GHz band\n[1]: '], 's');
if isempty(reply)
    reply = '1';
end
if strcmp(reply, '1')
    bandToScan = 5;
    validChannels = [...
        ' 7-16    (5.035-5.080 GHz)\n' ...]
```

```
' 34-64  (5.170-5.320 GHz)\n' ...
' 100-144 (5.550-5.720 GHz)\n' ...
' 149-165 (5.745-5.825 GHz)\n' ...
];
defaultChannels = '[153 157]';
else
    bandToScan = 2.4;
    validChannels = [...
        ' 1-13  (2.412-2.472 GHz)\n'...
        ' 14     (2.484 GHz)\n'...
    ];
    defaultChannels = '[1 6]';
end

% Get channels to scan
reply = input(['Valid channel numbers are:\n' validChannels ...
    'Which channels do you want to scan? ' defaultChannels ':'],'s');
if isempty(reply)
    reply = defaultChannels;
end
channelsToScan = reshape(str2num(reply),[],1); %#ok<ST2NM>
```

### Codegen acceleration

To accelerate receiver function performance, generate code if you have valid MATLAB Coder™ license.

```
runCodegen = false;
if checkCodegenLicense
    reply = input('Do you want to generate MEX file for receiver? Y/N [N]: ','s');
    if isempty(reply)
        reply = 'N';
    end
    if strcmpi(reply, 'Y')
        runCodegen = true;
    end
end
compileIt = runCodegen;

% Generate MEX file for receiver
if compileIt
    fprintf('Generating MEX file for receiver\n');
    clear runWLANNonHTReceiver_mex
    codegen('runWLANNonHTReceiver', '-args', {1e9, coder.Constant(Config)});
end
```

### Capture and Decode OFDM-based Packets

Capture and try to decode packets at the channels list for the amount of time requested. The center frequency is a tunable parameter for the radio; therefore, it can be changed without having to regenerate code for the entire receiver function.

```
for channel= 1:length(channelsToScan)
    % Calculate center frequency for channel
    WiFiCenterFrequency = helperWLANCHannelFrequency(channelsToScan(channel),bandToScan);

    % Run receiver
    fprintf('Running receiver at channel %d (%1.3f GHz)\n',...
```

```

    channelsToScan(channel), WiFiCenterFrequency/1e9);

if runCodegen
    runWLANNonHTReceiver_mex(WiFiCenterFrequency, Config);
else
    runWLANNonHTReceiver(WiFiCenterFrequency, Config);
end
end

% Run complete
fprintf('Desired channel(s) scanned\n');

```

When running the simulation, the recovered packets are decoded as soon as they are captured. If they pass their FCS check and they are of the subtype **beacon**, information will be printed to the command-line. A sample output is as follows:

```

Running receiver at channel 153 (5.765 GHz)
SSID: w-guest
Packets Decoded: 1
-----
SSID: w-guest
Packets Decoded: 2
-----
### Processed 100 milliseconds of received data ...
SSID: w-guest
Packets Decoded: 3
-----
SSID: w-guest
Packets Decoded: 4
-----
### Processed 200 milliseconds of received data ...
Running receiver at channel 157 (5.785 GHz)
SSID: w-guest
Packets Decoded: 5
-----
### Processed 100 milliseconds of received data ...
SSID: w-guest
Packets Decoded: 6
-----
### Processed 200 milliseconds of received data ...
Desired channel(s) scanned

```

The gain behavior of different USRP® daughter boards varies considerably. Thus, the gain setting defined in this example may not be well-suited for your daughter boards. If packets are not properly decoded by the receiver system, you can vary the gain of the source signals in **SDRu Receiver** System objects by changing the `Config.USRPGain` value in the receiver initialization file.

The desired collection time should not exceed 5 seconds. To ensure contiguous data from the USRP®, burst mode is used, which internally stores samples in a separate buffer. The maximum size of this buffer is OS dependent and will be exceeded when the collection time is too long. When using the B-Series USRP® devices, it can be useful to reset the radio by running `call_uhd_app('b2xx_fx3_utils', '-D')` if the function `runWLANNonHTReceiver` was exited prematurely.

## Appendix

This example uses the following script and helper functions:

- `getRadioInfoWLANBeacon`
- `getUserInputWLANBeacon`
- `runWLANNonHTReceiver`

The following WLAN Toolbox functions were also used:

- `wlanLLTFChannelEstimate`
- `wlanLLTDFdemodulate`
- `wlanLSIGRecover`
- `wlanNonHTConfig`
- `wlanNonHTDataRecover`
- `wlanPacketDetect`
- `wlanSymbolTimingEstimate`
- `wlanMPDUDecode`

## References

- 1 IEEE Std 802.11-2012 (Revision of IEEE Std 802.11-2007) - IEEE Standard for Information technology--Telecommunications and information exchange between systems Local and metropolitan area networks--Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

# LTE Cell Search, MIB and SIB1 Recovery with Two Antennas

This example uses both channels of USRP® B210, X300 or X310 to receive an LTE downlink signal. The LTE System Toolbox™ is used to synchronize, demodulate and decode the signal sent by the accompanying example sdrLTE2x2SIB1Tx.m. Since the transmitted signal uses a transmit diversity scheme, orthogonal space frequency block code (OSFBC) decoding is performed by the function lteTransmitDiversityDecode. In the end, the SIB1 field, the first of the System Information Blocks, is recovered and the CRC is checked.

This example uses the SDRu Receiver System object™. The ChannelMapping property of the object is set to [1 2] to enable use of both channels. The step method outputs a two-column matrix in which the first column is the signal from 'RF A' of the radio and the second column is the signal from 'RF B' of the radio.

Before starting this example, please run sdrLTE2x2SIB1Tx.m in a separate MATLAB session. In Windows, if two B210 radios are used for these examples, each radio must be connected to a separate computer.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object.

## Connect to Radio

```

radioFound = false;
radiolist = findsdru;
for i = 1:length(radiolist)
    if strcmp(radiolist(i).Status, 'Success')
        if strcmp(radiolist(i).Platform, 'B210')
            radio = comm.SDRuReceiver('Platform', 'B210', ...
                'SerialNum', radiolist(i).SerialNum);
            radio.MasterClockRate = 1.92e6 * 4; % Need to exceed 5 MHz minimum
            radio.DecimationFactor = 4; % Sampling rate is 1.92e6
            radioFound = true;
            break;
        end
        if (strcmp(radiolist(i).Platform, 'X300') || ...
            strcmp(radiolist(i).Platform, 'X310'))
            radio = comm.SDRuReceiver('Platform', radiolist(i).Platform, ...
                'IPAddress', radiolist(i).IPAddress);
            radio.MasterClockRate = 184.32e6;
            radio.DecimationFactor = 96; % Sampling rate is 1.92e6
            radioFound = true;
        end
    end
end

if ~radioFound
    error(message('sdru:examples:NeedMIMORadio'));
end

radio.ChannelMapping = [1 2]; % Receive signals from both channels
radio.CenterFrequency = 900e6;
radio.Gain = 30;
radio.SamplesPerFrame = 19200; % Sampling rate is 1.92 MHz. LTE frames are 10 ms long
radio.OutputDataType = 'double';
radio.EnableBurstMode = true;

```

```
radio.NumFramesInBurst = 4;
radio.OverrunOutputPort = true;

radio

Checking radio connections...

radio =

    System: comm.SDRuReceiver

Properties:
    Platform: 'B210'
    SerialNum: 'ECR04ZDBT'
    ChannelMapping: [1 2]
    CenterFrequency: 900000000
    LocalOscillatorOffset: 0
        Gain: 30
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 7680000
    DecimationFactor: 4
    TransportDataType: 'int16'
        OutputDataType: 'double'
    SamplesPerFrame: 19200
    EnableBurstMode: true
    NumFramesInBurst: 4
```

## Capture Signal

```
burstCaptures = zeros(19200,4,2);

len = 0;
for frame = 1:4
    while len == 0
        [data,len,lostSamples] = step(radio);
        burstCaptures(:,frame,:) = data;
    end
    len = 0;
end
release(radio);

eNodeB0output = reshape(burstCaptures,[],2);
sr = 1.92e6 ; % LTE sampling rate

% Check for presence of LTE System Toolbox
if isempty(ver('lte'))
    error(message('sdru:examples:NeedLST'));
end

% Prior to decoding the MIB, the UE does not know the full system
% bandwidth. The primary and secondary synchronization signals (PSS and
% SSS) and the PBCH (containing the MIB) all lie in the central 72
% subcarriers (6 resource blocks) of the system bandwidth, allowing the UE
% to initially demodulate just this central region. Therefore the bandwidth
% is initially set to 6 resource blocks. The I/Q waveform needs to be
% resampled accordingly. At this stage we also display the spectrum of the
```

```
% input signal |eNodeBOutput|.

% set up some housekeeping variables:
% separator for command window logging
separator = repmat('-',1,50);
% plots
if (~exist('channelFigure','var') || ~isValid(channelFigure))
    channelFigure = figure('Visible','off');
end
[spectrumAnalyzer,synchCorrPlot,pdcchConstDiagram] = ...
    hSIB1RecoveryExamplePlots(channelFigure,sr);
% PDSCH EVM
pdschEVM = comm.EVM();
pdschEVM.MaximumEVMOutputPort = true;

% Set eNodeB basic parameters
enb = struct; % eNodeB config structure
enb.DuplexMode = 'FDD'; % assume FDD duplexing mode
enb.CyclicPrefix = 'Normal'; % assume normal cyclic prefix
enb.NDLRB = 6; % Number of resource blocks
ofdmInfo = lteOFDMInfo(enb); % Needed to get the sampling rate

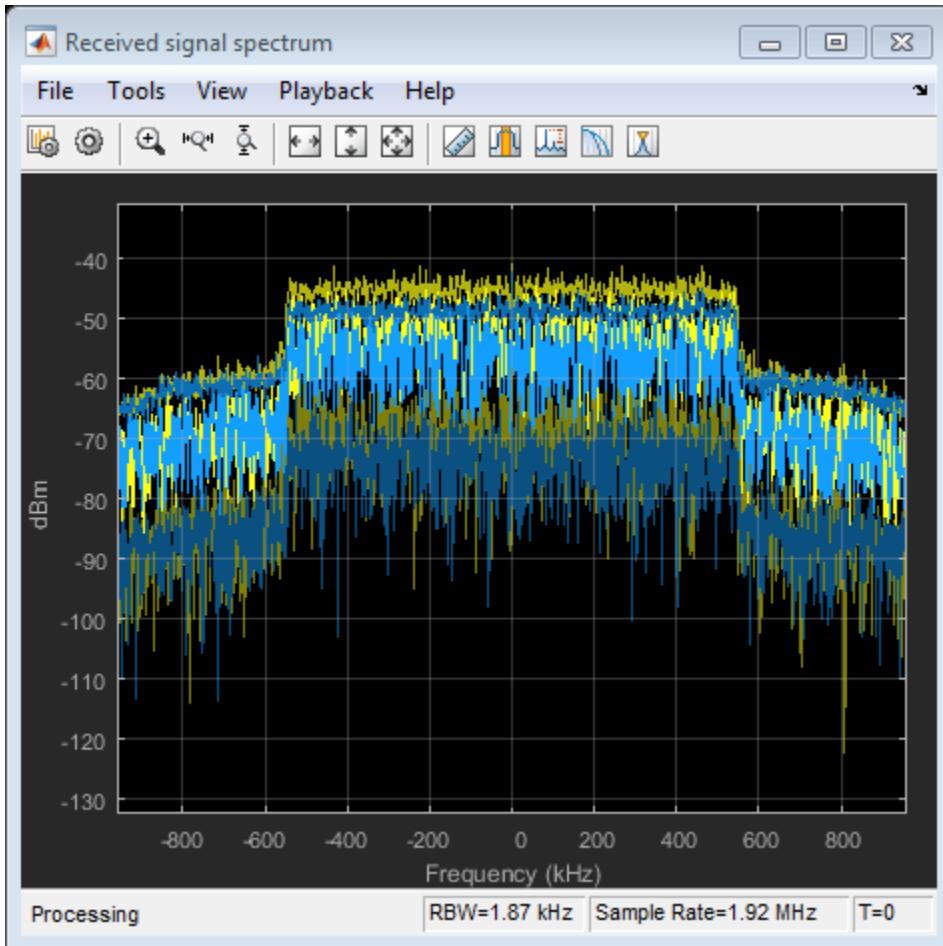
if (isempty(eNodeBOutput))
    fprintf('\nReceived signal must not be empty.\n');
    return;
end

% Display received signal spectrum
fprintf('\nPlotting received signal spectrum...\n');
step(spectrumAnalyzer, awgn(eNodeBOutput, 100.0));

if (sr~=ofdmInfo.SamplingRate)
    fprintf('\nResampling from %0.3fMs/s to %0.3fMs/s for cell search / MIB decoding...\n',sr/1e3,ofdmInfo.SamplingRate/1e3);
else
    fprintf('\nResampling not required; received signal is at desired sampling rate for cell search\n');
end
% Downsample received signal
nSamples = ceil(ofdmInfo.SamplingRate/round(sr)*size(eNodeBOutput,1));
nRxAnts = size(eNodeBOutput, 2);
downsampled = zeros(nSamples, nRxAnts);
for i=1:nRxAnts
    downsampled(:,i) = resample(eNodeBOutput(:,i), ofdmInfo.SamplingRate, round(sr));
end

Plotting received signal spectrum...

Resampling not required; received signal is at desired sampling rate for cell search / MIB decoding...
```



### Frequency offset estimation and correction

Prior to OFDM demodulation, any significant frequency offset must be removed. The frequency offset in the I/Q waveform is estimated and corrected using `lteFrequencyOffset` and `lteFrequencyCorrect`. The frequency offset is estimated by means of correlation of the cyclic prefix and therefore can estimate offsets up to +/- half the subcarrier spacing i.e. +/- 7.5kHz.

```

fprintf('\nPerforming frequency offset estimation...\n');
% Note that the duplexing mode is set to FDD here because timing synch has
% not yet been performed - for TDD we cannot use the duplexing arrangement
% to indicate which time periods to use for frequency offset estimation
% prior to doing timing synch.
delta_f = lteFrequencyOffset(setfield(enb, 'DuplexMode', 'FDD'), downsampled); %#ok<SLFD>
fprintf('Frequency offset: %0.3fHz\n', delta_f);
downsampled = lteFrequencyCorrect(enb, downsampled, delta_f);

```

```

Performing frequency offset estimation...
Frequency offset: 689.247Hz

```

## Cell Search and Synchronization

Call `lteCellSearch` to obtain the cell identity and timing offset `offset` to the first frame head. A plot of the correlation between the received signal and the PSS/SSS for the detected cell identity is produced.

```
% Cell search to find cell identity and timing offset
fprintf('\nPerforming cell search...\n');
[cellID, offset] = lteCellSearch(enb, downsampled);

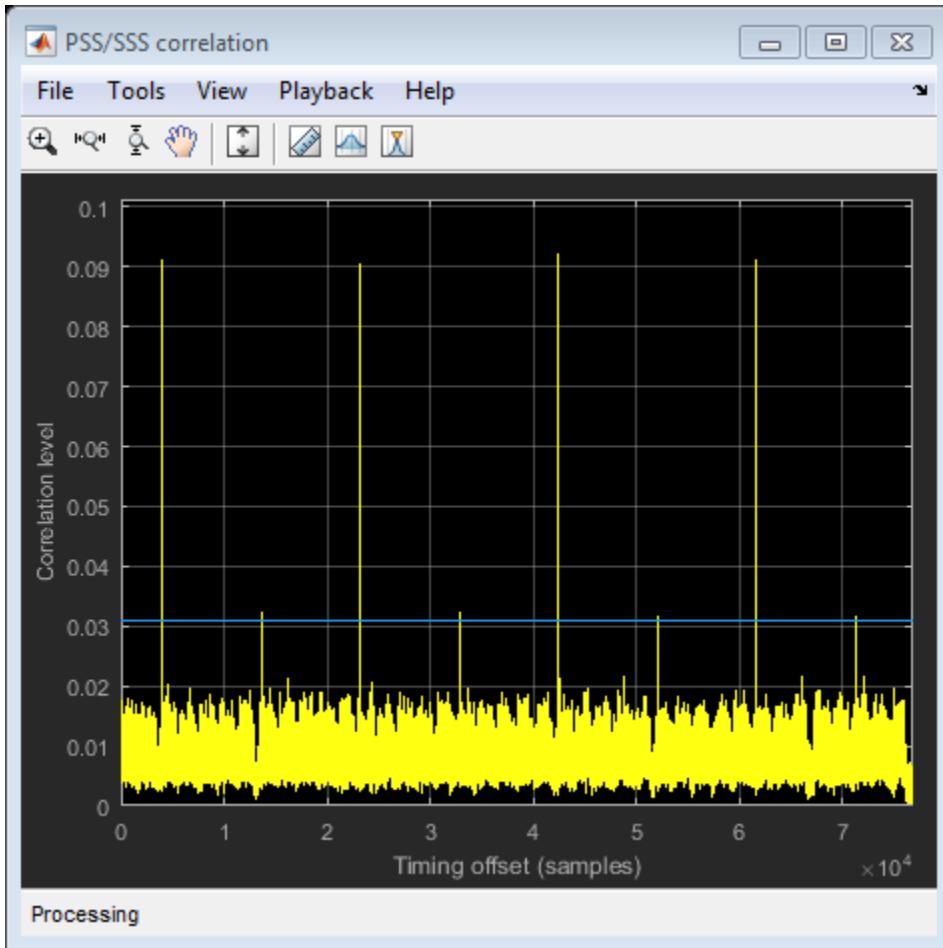
% Compute the correlation for each of the three possible primary cell
% identities; the peak of the correlation for the cell identity established
% above is compared with the peak of the correlation for the other two
% primary cell identities in order to establish the quality of the
% correlation.
corr = cell(1,3);
for i = 0:2
    enb.NCellID = mod(cellID + i,504);
    [~,corr{i+1}] = lteDLFrameOffset(enb, downsampled);
    corr{i+1} = sum(corr{i+1},2);
end
threshold = 1.3 * max([corr{2}; corr{3}]); % multiplier of 1.3 empirically obtained
if (max(corr{1})<threshold)
    warning('sdru:examples:WeakSignal','Synchronization signal correlation was weak; detected ce')
end
enb.NCellID = cellID;

% plot PSS/SSS correlation and threshold
synchCorrPlot.YLimits = [0 max([corr{1}; threshold])*1.1];
step(synchCorrPlot, [corr{1} threshold*ones(size(corr{1}))]);

% perform timing synchronisation
fprintf('Timing offset to frame start: %d samples\n',offset);
downsampled = downsampled(1+offset:end,:);
enb.NSubframe = 0;

% show cell-wide settings
fprintf('Cell-wide settings after cell search:\n');
disp(enb);

Performing cell search...
Timing offset to frame start: 4054 samples
Cell-wide settings after cell search:
    DuplexMode: 'FDD'
    CyclicPrefix: 'Normal'
        NDLRB: 6
        NCellID: 64
        NSubframe: 0
```



### OFDM Demodulation and Channel Estimation

The OFDM downsampled I/Q waveform is demodulated to produce a resource grid `rgrid`. This is used to perform channel estimation. `hest` is the channel estimate, `nest` is an estimate of the noise (for MMSE equalization) and `cec` is the channel estimator configuration.

For channel estimation the example assumes 4 cell specific reference signals. This means that channel estimates to each receiver antenna from all possible cell-specific reference signal ports are available. The true number of cell-specific reference signal ports is not yet known. The channel estimation is only performed on the first subframe, i.e. using the first L OFDM symbols in `rxgrid`.

A conservative 9-by-9 pilot averaging window is used, in time and frequency, to reduce the impact of noise on pilot estimates during channel estimation.

```
% Channel estimator configuration
cec.PilotAverage = 'UserDefined'; % Type of pilot averaging
cec.FreqWindow = 9; % Frequency window size
cec.TimeWindow = 9; % Time window size
cec.InterpType = 'cubic'; % 2D interpolation type
cec.InterpWindow = 'Centered'; % Interpolation window type
cec.InterpWinSize = 1; % Interpolation window size

% Assume 4 cell-specific reference signals for initial decoding attempt;
```

```
% ensures channel estimates are available for all cell-specific reference
% signals
enb.CellRefP = 4;

fprintf('Performing OFDM demodulation...\n\n');

griddims = lteResourceGridSize(enb); % Resource grid dimensions
L = griddims(2); % Number of OFDM symbols in a subframe
% OFDM demodulate signal
rxgrid = lteOFDDemodulate(enb, downsampled);
if (isempty(rxgrid))
    fprintf('After timing synchronization, signal is shorter than one subframe so no further demodulation is performed.\n');
    return;
end
% Perform channel estimation
if (strcmpi(enb.DuplexMode, 'TDD'))
    enb.TDDConfig = 0;
    enb.SSC = 0;
end
[hest, nest] = lteDLChannelEstimate(enb, cec, rxgrid(:,1:L,:));
Performing OFDM demodulation...
```

### PBCH Demodulation, BCH Decoding, MIB parsing

The MIB is now decoded along with the number of cell-specific reference signal ports transmitted as a mask on the BCH CRC. The function `ltePBCHDecode` establishes frame timing modulo 4 and returns this in the `nfmod4` parameter. It also returns the MIB bits in vector `mib` and the true number of cell-specific reference signal ports which is assigned into `enb.CellRefP` at the output of this function call. If the number of cell-specific reference signal ports is decoded as `enb.CellRefP=0`, this indicates a failure to decode the BCH. The function `lteMIB` is used to parse the bit vector `mib` and add the relevant fields to the configuration structure `enb`. After MIB decoding, the detected bandwidth is present in `enb.NDLRB`.

```
% Decode the MIB
% Extract resource elements (REs) corresponding to the PBCH from the first
% subframe across all receive antennas and channel estimates
fprintf('Performing MIB decoding...\n');
pbchIndices = ltePBCHIndices(enb);
[pbchRx, pbchHest] = lteExtractResources( ...
    pbchIndices, rxgrid(:,1:L,:), hest(:,1:L,:,:));

% Decode PBCH
[bchBits, pbchSymbols, nfmod4, mib, enb.CellRefP] = ltePBCHDecode( ...
    enb, pbchRx, pbchHest, nest);

% Parse MIB bits
enb = lteMIB(mib, enb);

% Incorporate the nfmod4 value output from the function ltePBCHDecode, as
% the NFrame value established from the MIB is the System Frame Number
% (SFN) modulo 4 (it is stored in the MIB as floor(SFN/4))
enb.NFrame = enb.NFrame+nfmod4;

% Display cell wide settings after MIB decoding
fprintf('Cell-wide settings after MIB decoding:\n');
disp(enb);
```

```
if (enb.CellRefP==0)
    fprintf('MIB decoding failed (enb.CellRefP=0).\n\n');
    return;
end
if (enb.NDLRB==0)
    fprintf('MIB decoding failed (enb.NDLRB=0).\n\n');
    return;
end

Performing MIB decoding...
Cell-wide settings after MIB decoding:
    DuplexMode: 'FDD'
    CyclicPrefix: 'Normal'
        NDLRB: 6
        NCellID: 64
        NSubframe: 0
        CellRefP: 2
    PHICHDuration: 'Normal'
        Ng: 'Sixth'
        NFrame: 107
```

## OFDM Demodulation on Full Bandwidth

Now that the signal bandwidth is known, the signal is resampled to the nominal sampling rate used by LTE System Toolbox for that bandwidth (see `lteOFDMMModulate` for details). Frequency offset estimation and correction is performed on the resampled signal. Timing synchronization and OFDM demodulation are then performed.

```
fprintf('Restarting reception now that bandwidth (NDLRB=%d) is known...\n',enb.NDLRB);

% Resample now we know the true bandwidth
ofdmInfo = lteOFDMDInfo(enb);
if (sr~=ofdmInfo.SamplingRate)
    fprintf('\nResampling from %0.3fMs/s to %0.3fMs/s...\n',sr/1e6,ofdmInfo.SamplingRate/1e6);
else
    fprintf('\nResampling not required; received signal is at desired sampling rate for NDLRB=%d
end
nSamples = ceil(ofdmInfo.SamplingRate/round(sr)*size(eNodeBOutput,1));
resampled = zeros(nSamples, nRxAnts);
for i = 1:nRxAnts
    resampled(:,i) = resample(eNodeBOutput(:,i), ofdmInfo.SamplingRate, round(sr));
end

% Perform frequency offset estimation and correction
fprintf('\nPerforming frequency offset estimation...\n');
% Note that the duplexing mode is set to FDD here because timing synch has
% not yet been performed - for TDD we cannot use the duplexing arrangement
% to indicate which time periods to use for frequency offset estimation
% prior to doing timing synch.
delta_f = lteFrequencyOffset(setfield(enb,'DuplexMode','FDD'), resampled); %#ok<SLFD>
fprintf('Frequency offset: %0.3fHz\n',delta_f);
resampled = lteFrequencyCorrect(enb, resampled, delta_f);

% Find beginning of frame
fprintf('\nPerforming timing offset estimation...\n');
offset = lteDLFrameOffset(enb, resampled);
```

```

fprintf('Timing offset to frame start: %d samples\n',offset);
% aligning signal with the start of the frame
resampled = resampled(1+offset:end,:);

% OFDM demodulation
fprintf('\nPerforming OFDM demodulation...\n\n');
rxgrid = lteOFMDemodulate(enb, resampled);

Restarting reception now that bandwidth (NDLRB=6) is known...

Resampling not required; received signal is at desired sampling rate for NDLRB=6 (1.920Ms/s).

Performing frequency offset estimation...
Frequency offset: 689.247Hz

Performing timing offset estimation...
Timing offset to frame start: 4054 samples

Performing OFDM demodulation...

```

## SIB1 Decoding

The following steps are performed in this section:

- Physical Control Format Indicator Channel (PCFICH) demodulation, CFI decoding
- PDCCH decoding
- Blind PDCCH search
- SIB bits recovery: PDSCH demodulation and DL-SCH decoding
- Buffering and resetting of the DL-SCH HARQ state

After recovery the SIB CRC should be 0.

These decoding steps are performed in a loop for each occurrence of a subframe carrying SIB1 in the received signal. As mentioned above, the SIB1 is transmitted in subframe 5 of every even frame, so the input signal is first checked to establish that at least one occurrence of SIB1 is present. For each SIB1 subframe, the channel estimate magnitude response is plotted, as is the constellation of the received PDCCH.

```

% Check this frame contains SIB1, if not advance by 1 frame provided we
% have enough data, terminate otherwise.
if (mod(enb.NFrame,2)~=0)
    if (size(rxgrid,2)>=(L*10))
        rxgrid(:,1:(L*10),:) = [];
        fprintf('Skipping frame %d (odd frame number does not contain SIB1).\n\n',enb.NFrame);
    else
        rxgrid = [];
    end
    enb.NFrame = enb.NFrame + 1;
end

% Advance to subframe 5, or terminate if we have less than 5 subframes
if (size(rxgrid,2)>=(L*5))
    rxgrid(:,1:(L*5),:) = []; % Remove subframes 0 to 4
else
    rxgrid = [];

```

```
end
enb.NSubframe = 5;

if (isempty(rxgrid))
    fprintf('Received signal does not contain a subframe carrying SIB1.\n\n');
end

% Reset the HARQ buffers
decState = [];

% While we have more data left, attempt to decode SIB1
while (size(rxgrid,2) > 0)

    fprintf('%s\n',separator);
    fprintf('SIB1 decoding for frame %d\n',mod(enb.NFrame,1024));
    fprintf('%s\n\n',separator);

    % Reset the HARQ buffer with each new set of 8 frames as the SIB1
    % info may be different
    if (mod(enb.NFrame,8)==0)
        fprintf('Resetting HARQ buffers.\n\n');
        decState = [];
    end

    % Extract current subframe
    rxsubframe = rxgrid(:,1:L,:);

    % Perform channel estimation
    [hest,nest] = lteDLChannelEstimate(enb, cec, rxsubframe);

    % PCFICH demodulation, CFI decoding. The CFI is now demodulated and
    % decoded using similar resource extraction and decode functions to
    % those shown already for BCH reception. lteExtractResources is used to
    % extract REs corresponding to the PCFICH from the received subframe
    % rxsubframe and channel estimate hest.
    fprintf('Decoding CFI...\n\n');
    pcfichIndices = ltePCFICHIndices(enb); % Get PCFICH indices
    [pcfichRx, pcfichHest] = lteExtractResources(pcfichIndices, rxsubframe, hest);
    % Decode PCFICH
    cfiBits = ltePCFICHDecode(enb, pcfichRx, pcfichHest, nest);
    cfi = lteCFIDecode(cfiBits); % Get CFI
    if (isfield(enb,'CFI') && cfi==enb.CFI)
        release(pdcchConstDiagram);
    end
    enb.CFI = cfi;
    fprintf('Decoded CFI value: %d\n\n', enb.CFI);

    % For TDD, the PDCCH must be decoded blindly across possible values of
    % the PHICH configuration factor m_i (0,1,2) in TS36.211 Table 6.9-1.
    % Values of m_i = 0, 1 and 2 can be achieved by configuring TDD
    % uplink-downlink configurations 1, 6 and 0 respectively.
    if (strcmpi(enb.DuplexMode,'TDD'))
        tddConfigs = [1 6 0];
    else
        tddConfigs = 0; % not used for FDD, only used to control while loop
    end
    dci = {};
    while (isempty(dci) && ~isempty(tddConfigs))
```

```
% Configure TDD uplink-downlink configuration
if (strcmpi(enb.DuplexMode,'TDD'))
    enb.TDDConfig = tddConfigs(1);
end
tddConfigs(1) = [];
% PDCCH demodulation. The PDCCH is now demodulated and decoded
% using similar resource extraction and decode functions to those
% shown already for BCH and CFI reception
pdcchIndices = ltePDCCHIndices(enb); % Get PDCCH indices
[pdcchRx, pdcchHest] = lteExtractResources(pdcchIndices, rxsubframe, hest);
% Decode PDCCH and plot constellation
[dciBits, pdcchSymbols] = ltePDCCHDecode(enb, pdcchRx, pdcchHest, nest);
step(pdcchConstDiagram, pdcchSymbols);

% PDCCH blind search for System Information (SI) and DCI decoding.
% The LTE System Toolbox provides full blind search of the PDCCH to
% find any DCI messages with a specified RNTI, in this case the
% SI-RNTI.
fprintf('PDCCH search for SI-RNTI...\n\n');
pdcch = struct('RNTI', 65535);
dci = ltePDCCHSearch(enb, pdcch, dciBits); % Search PDCCH for DCI
end

% If DCI was decoded, proceed with decoding PDSCH / DL-SCH
if ~isempty(dci)

    dci = dci{1};
    fprintf('DCI message with SI-RNTI:\n');
    disp(dci);
    % Get the PDSCH configuration from the DCI
    [pdsch, trblklen] = hPDSCHConfiguration(enb, dci, pdch.RNTI);
    pdsch.NTurboDecIts = 5;
    fprintf('PDSCH settings after DCI decoding:\n');
    disp(pdsch);

    % PDSCH demodulation and DL-SCH decoding to recover SIB bits.
    % The DCI message is now parsed to give the configuration of the
    % corresponding PDSCH carrying SIB1, the PDSCH is demodulated and
    % finally the received bits are DL-SCH decoded to yield the SIB1
    % bits.

    fprintf('Decoding SIB1...\n\n');
    % Get PDSCH indices
    [pdschIndices,pdschIndicesInfo] = ltePDSCHIndices(enb, pdsch, pdsch.PRBS);
    [pdschRx, pdschHest] = lteExtractResources(pdschIndices, rxsubframe, hest);
    % Decode PDSCH
    [dlschBits,pdschSymbols] = ltePDSCHDecode(enb, pdsch, pdschRx, pdschHest, nest);
    % Decode DL-SCH with soft buffer input/output for HARQ combining
    if ~isempty(decState)
        fprintf('Recombining with previous transmission.\n\n');
    end
    [sib1, crc, decState] = lteDLSCHDecode(enb, pdsch, trblklen, dlschBits, decState);

    % Compute PDSCH EVM
    recoded = lteDLSCH(enb, pdsch, pdschIndicesInfo.G, sib1);
    remod = ltePDSCH(enb, pdsch, recoded);
    [~,refSymbols] = ltePDSCHDecode(enb, pdsch, remod);
    release(pdschEVM);

end
```

```
[rmsevm,peakevm] = step(pdschEVM, refSymbols{1}, pdschSymbols{1});
fprintf('PDSCH RMS EVM: %.3f%%\n',rmsevm);
fprintf('PDSCH Peak EVM: %.3f%%\n\n',peakevm);

fprintf('SIB1 CRC: %d\n',crc);
if crc == 0
    fprintf('Successful SIB1 recovery.\n\n');
else
    fprintf('SIB1 decoding failed.\n\n');
end

else
    % indicate that DCI decoding failed
    fprintf('DCI decoding failed.\n\n');
end

% Update channel estimate plot
figure(channelFigure);
surf(abs(hest(:,:,1,1)));
hSIB1RecoveryExamplePlots(channelFigure);
channelFigure.CurrentAxes.XLim = [0 size(hest,2)+1];
channelFigure.CurrentAxes.YLim = [0 size(hest,1)+1];

% Skip 2 frames and try SIB1 decoding again, or terminate if we
% have less than 2 frames left.
if (size(rxgrid,2)>=(L*20))
    rxgrid(:,1:(L*20),:) = []; % Remove 2 more frames
else
    rxgrid = []; % Less than 2 frames left
end
enb.NFrame = enb.NFrame+2;

end

Skipping frame 107 (odd frame number does not contain SIB1).

-----
SIB1 decoding for frame 108
-----

Decoding CFI...
Decoded CFI value: 3
PDCCH search for SI-RNTI...
DCI message with SI-RNTI:
    DCIFormat: 'Format1A'
        CIF: 0
    AllocationType: 0
        Allocation: [1x1 struct]
        ModCoding: 5
        HARQNo: 0
        NewData: 0
        RV: 3
    TPCPUCCH: 0
    TDDIndex: 0
```

```
PDSCH settings after DCI decoding:  
    RNTI: 65535  
    PRBSet: [6x1 uint64]  
    NLayers: 2  
    Modulation: {'QPSK'}  
    RV: 3  
    TxScheme: 'TxDiversity'  
    NTurboDecIts: 5
```

Decoding SIB1...

```
PDSCH RMS EVM: 14.875%  
PDSCH Peak EVM: 47.133%
```

```
SIB1 CRC: 0  
Successful SIB1 recovery.
```

```
-----  
SIB1 decoding for frame 110  
-----
```

Decoding CFI...

Decoded CFI value: 3

PDCCH search for SI-RNTI...

```
DCI message with SI-RNTI:  
    DCIFormat: 'Format1A'  
    CIF: 0  
    AllocationType: 0  
    Allocation: [1x1 struct]  
    ModCoding: 5  
    HARQNo: 0  
    NewData: 0  
    RV: 1  
    TPCPUCCH: 0  
    TDDIndex: 0
```

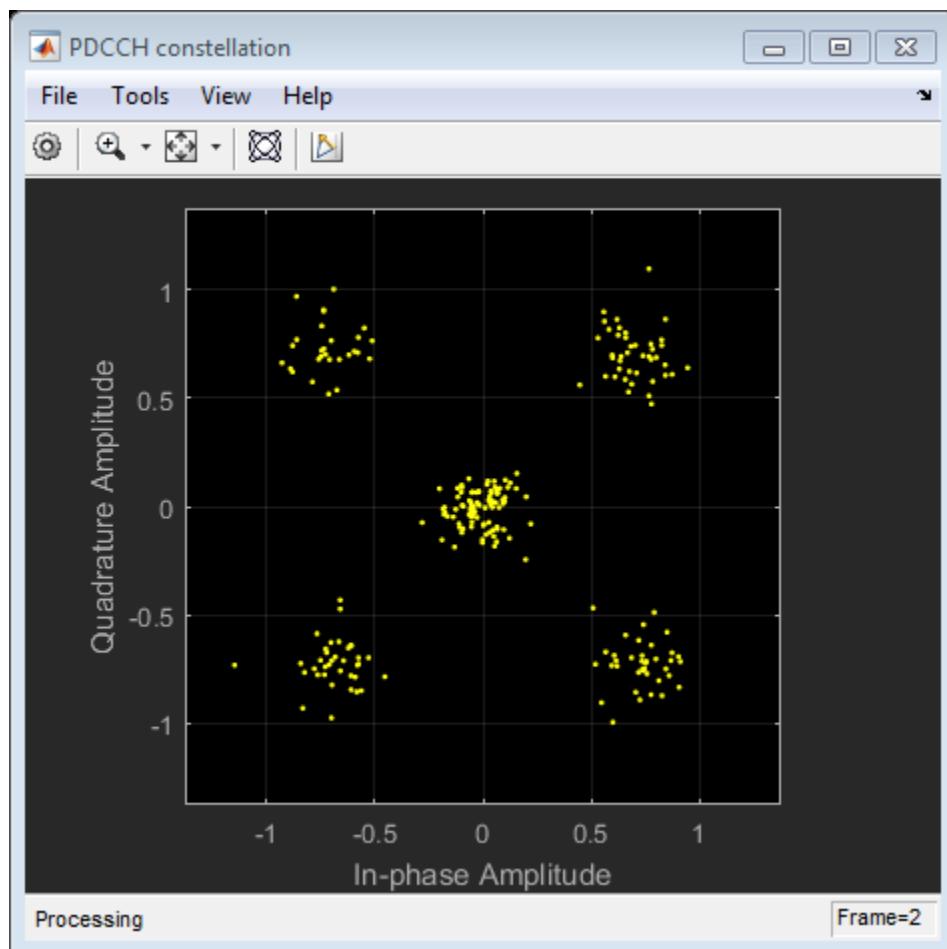
```
PDSCH settings after DCI decoding:  
    RNTI: 65535  
    PRBSet: [6x1 uint64]  
    NLayers: 2  
    Modulation: {'QPSK'}  
    RV: 1  
    TxScheme: 'TxDiversity'  
    NTurboDecIts: 5
```

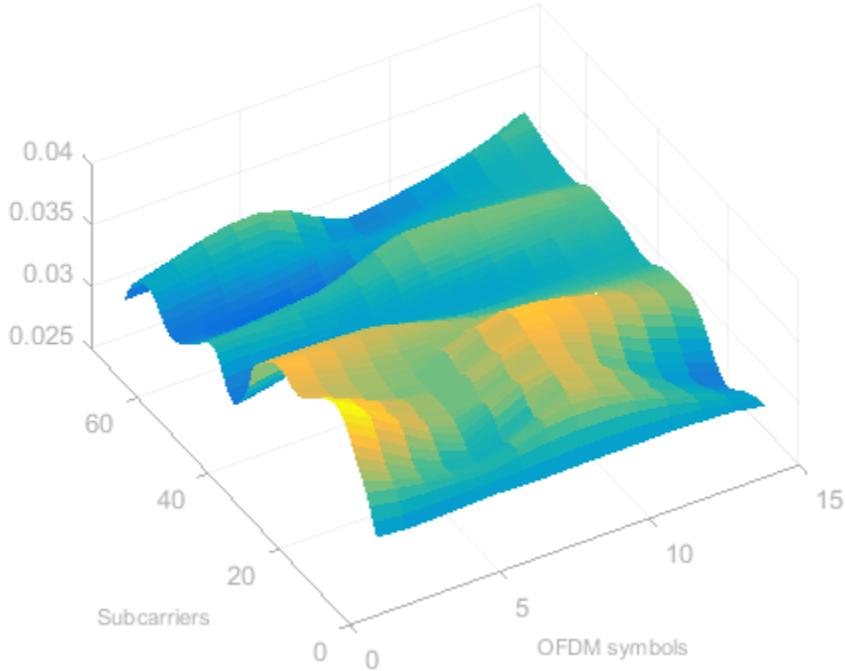
Decoding SIB1...

Recombining with previous transmission.

```
PDSCH RMS EVM: 15.589%  
PDSCH Peak EVM: 37.974%
```

```
SIB1 CRC: 0  
Successful SIB1 recovery.
```





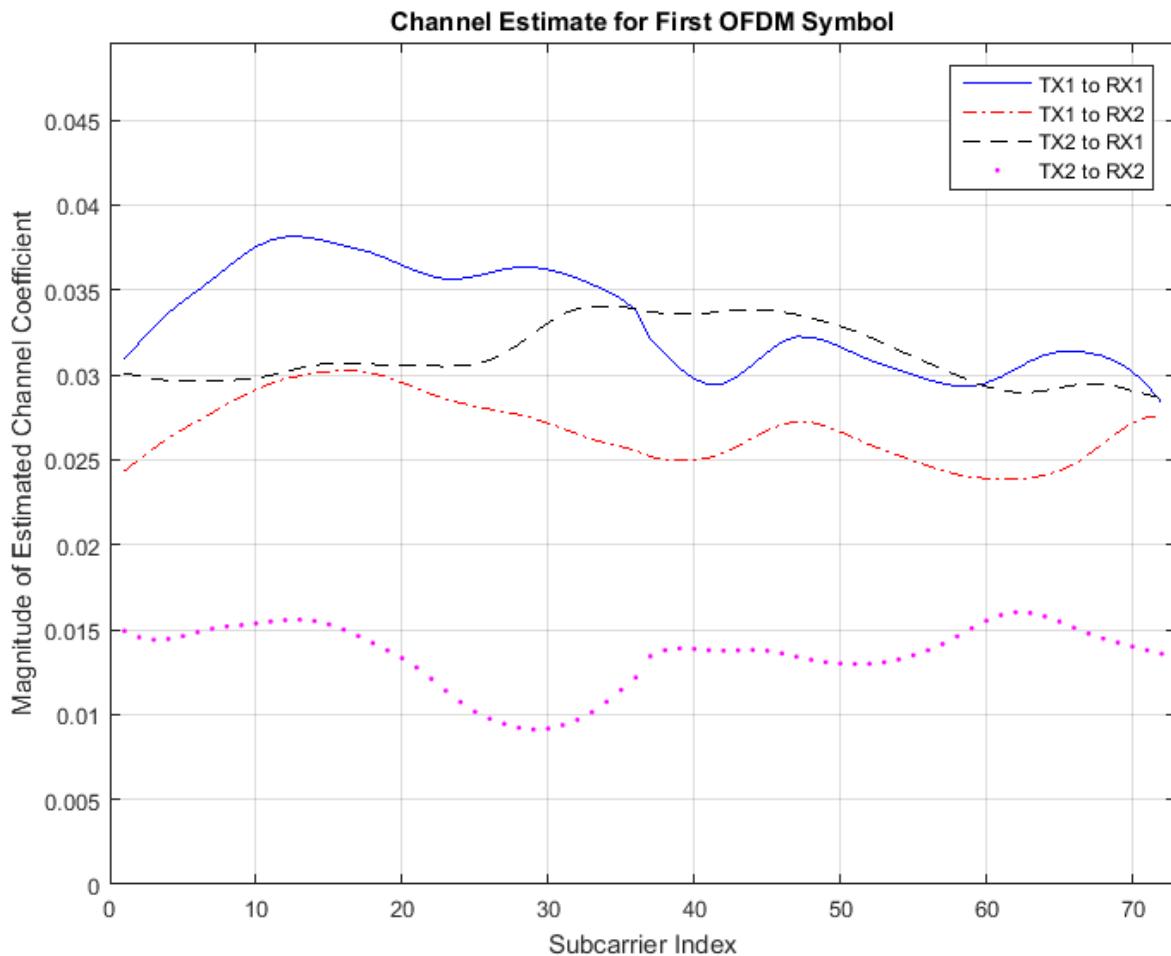
### Plot Channel Estimate for First OFDM Symbol

In this example, MIMO transmission is performed on four links between two transmit antennas and two receive antennas. The following plot provides a snapshot of the estimated channel coefficients.

```

figure('Position',[0 0 800 600])
plot(abs(hest(:,1,1,1)),'b-')
hold on
plot(abs(hest(:,1,2,1)),'r-.')
plot(abs(hest(:,1,1,2)),'k--')
plot(abs(hest(:,1,2,2)),'m.')
hold off
grid on
m = max(reshape(abs(hest(:,:,1)),[],1));
axis([0 size(hest,1)+1 0 m*1.3])
title('Channel Estimate for First OFDM Symbol')
xlabel('Subcarrier Index')
ylabel('Magnitude of Estimated Channel Coefficient')
legend('TX1 to RX1','TX1 to RX2',...
    'TX2 to RX1','TX2 to RX2',...
    'Location','northeast')

```



### Tips for Maximizing Performance

- Run this example and `sdruLTE2x2SIB1Tx.m` on two computers
- Double `DecimationFactor` in this example and `InterpolationFactor` in `sdruLTE2x2SIB1Tx.m`
- Start MATLAB in -nodesktop mode before running `sdruLTE2x2SIB1Tx.m`
- If you are using one computer with two B210 radios, do not connect them to adjacent USB ports

### Appendix

This example uses the following helper functions:

- `hPDSCHConfiguration.m`
- `hSIB1RecoveryExamplePlots.m`

### Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## LTE SIB1 Transmission over Two Antennas

This example uses both channels of USRP® B210, X300 or X310 to transmit an LTE downlink signal that requires two antennas. The signal is generated by the LTE System Toolbox™ and random bits are inserted into the SIB1 field, the first of the System Information Blocks. The accompanying example sdruLTE2x2SIB1Rx.m receives this signal with two antennas, recovers the SIB1 data, and checks the CRC.

This example uses the SDRu Transmitter System object™. The ChannelMapping property of the object is set to [1 2] to enable use of both channels. The step method takes a two-column matrix in which the first column is the signal for 'RF A' of the radio and the second column is the signal for 'RF B' of the radio.

After starting this example, please run sdruLTE2x2SIB1Rx.m in a new MATLAB session. In Windows, if two B210 radios are used for these examples, each radio must be connected to a separate computer.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter System object.

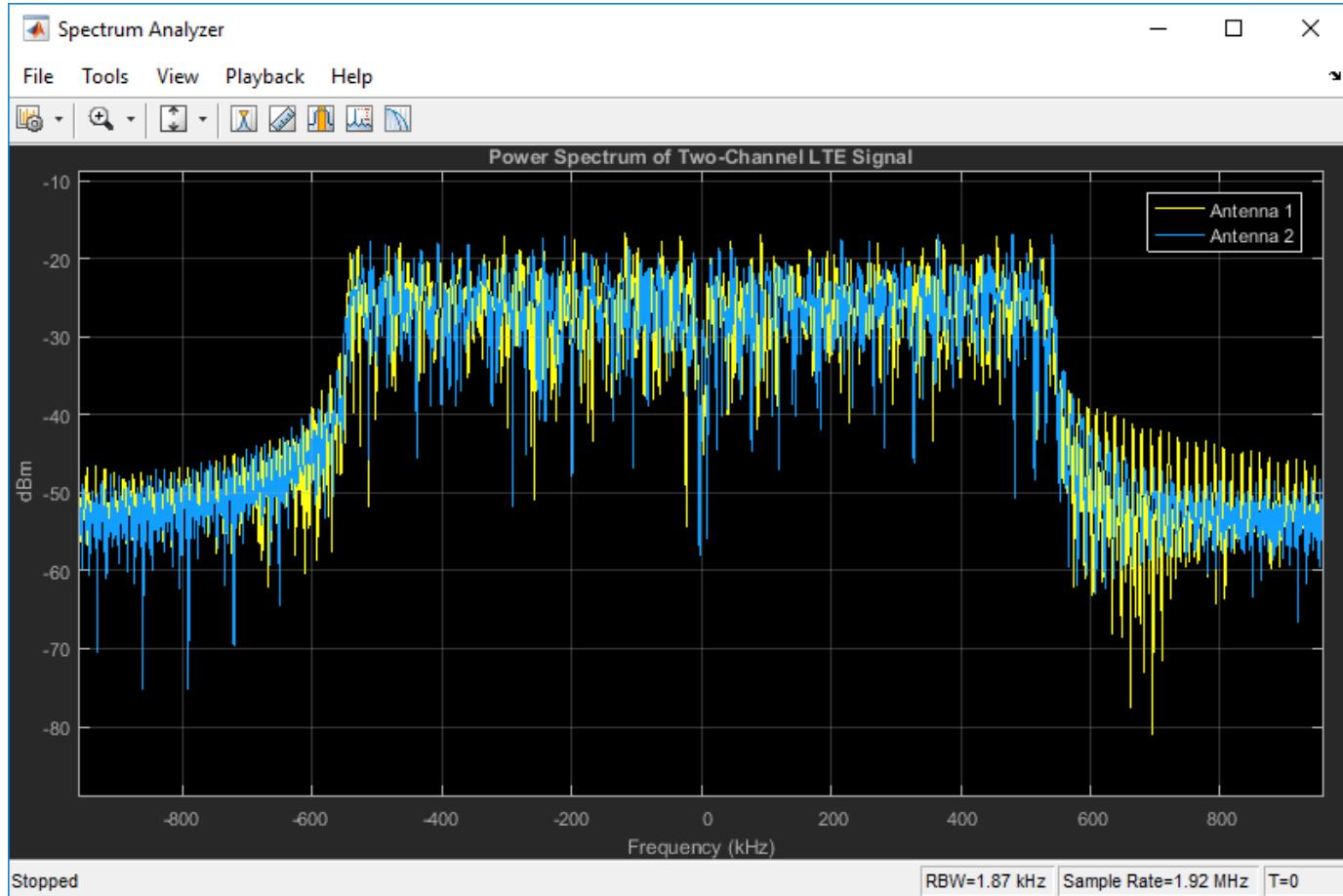
### Generate LTE Signal

```
% Check for presence of LTE System Toolbox
if isempty(ver('lte'))
    error(message('sdru:examples:NeedLST'));
end

% Generate LTE signal
rmc = lteRMCDL('R.12'); % Base RMC configuration
rmc.CellRefP = 2; % 2 transmit antennas
rmc.PDSCH.NLayers = 2; % 2 layers
rmc.NCellID = 64; % Cell identity
rmc.NFrame = 100; % Initial frame number
rmc.TotSubframes = 8*10; % Generate 8 frames. 10 subframes per frame
rmc.OCNGPDSCHEnable = 'On'; % Add noise to unallocated PDSCH resource elements
rmc.PDSCH.RNTI = 61;
rmc.SIB.Enable = 'On';
rmc.SIB.DCIFormat = 'Format1A';
rmc.SIB.AllocationType = 0;
rmc.SIB.VRBStart = 0;
rmc.SIB.VRBLength = 6;
rmc.SIB.Gap = 0;
rmc.SIB.Data = randi([0 1],144,1); % Use random bits in SIB data field. This is not a valid SIB
trData = [1;0;0;1];
[eNodeBOutput,txGrid,rmc] = lteRMCDLTool(rmc,trData);
```

### Plot Power Spectrum of Two-Channel LTE Signal

```
spectrumAnalyzer = dsp.SpectrumAnalyzer;
spectrumAnalyzer.SampleRate = rmc.SamplingRate; % 1.92e6 MHz for 'R.12'
spectrumAnalyzer.Title = 'Power Spectrum of Two-Channel LTE Signal';
spectrumAnalyzer.ShowLegend = true;
spectrumAnalyzer.ChannelNames = {'Antenna 1', 'Antenna 2'};
step(spectrumAnalyzer, eNodeBOutput);
release(spectrumAnalyzer);
```



### Connect to Radio

```

radioFound = false;
radiolist = findsdru;
for i = 1:length(radiolist)
    if strcmp(radiolist(i).Status, 'Success')
        if strcmp(radiolist(i).Platform, 'B210')
            radio = comm.SDRuTransmitter('Platform','B210', ...
                'SerialNum', radiolist(i).SerialNum);
            radio.MasterClockRate = 1.92e6 * 4; % Need to exceed 5 MHz minimum
            radio.InterpolationFactor = 4;      % Sampling rate is 1.92 MHz
            radioFound = true;
            break;
        end
        if (strcmp(radiolist(i).Platform, 'X300') || ...
            strcmp(radiolist(i).Platform, 'X310'))
            radio = comm.SDRuTransmitter('Platform',radiolist(i).Platform, ...
                'IPAddress', radiolist(i).IPAddress);
            radio.MasterClockRate = 184.32e6;
            radio.InterpolationFactor = 96;      % Sampling rate is 1.92 MHz
            radioFound = true;
        end
    end
end

```

```

if ~radioFound
    error(message('sdru:examples:NeedMIMORadio'));
end

radio.ChannelMapping = [1 2];      % Use both TX channels
radio.CenterFrequency = 900e6;
radio.Gain = 25;
radio.UnderrunOutputPort = true;

radio

Checking radio connections...

radio =

System: comm.SDRuTransmitter

Properties:
    Platform: 'B210'
    SerialNum: 'ECR04ZDBT'
    ChannelMapping: [1 2]
    CenterFrequency: 900000000
    LocalOscillatorOffset: 0
        Gain: 25
    PPSSource: 'Internal'
    ClockSource: 'Internal'
    MasterClockRate: 7680000
    InterpolationFactor: 4
    TransportDataType: 'int16'
    EnableBurstMode: false

```

## Send Signal over Two Antennas

```

% Scale signal to make maximum magnitude equal to 1
eNodeB0output = eNodeB0output/max(abs(eNodeB0output(:)));

% Reshape signal as a 3D array to simplify the for loop below
% Each call to step method of the object will use a two-column matrix
samplesPerFrame = 10e-3*rmc.SamplingRate;      % LTE frames are 10 ms long
numFrames = length(eNodeB0output)/samplesPerFrame;
txFrame = permute(reshape(permute(eNodeB0output,[1 3 2]), ...
    samplesPerFrame,numFrames,rmc.CellRefP),[1 3 2]);

disp('Starting transmission');
disp('Please run sdruLTE2x2SIB1Rx.m in a new MATLAB session');

currentTime = 0;
while currentTime < 300                         % Run for 5 minutes
    for n = 1:numFrames
        % Call step method to send a two-column matrix
        % First column for TX channel 1. Second column for TX channel 2
        bufferUnderflow = step(radio,txFrame(:,:,n));
        if bufferUnderflow~=0
            warning('sdru:examples:DroppedSamples','Dropped samples')
        end
    end
    currentTime = currentTime+numFrames*10e-3; % One frame is 10 ms

```

```
end
release(radio);
disp('Transmission finished')

Starting transmission
Please run sdruLTE2x2SIB1Rx.m in a new MATLAB session
Transmission finished
```

### Tips for Maximizing Performance

- Run this example and sdruLTE2x2SIB1Rx.m on two computers
- Double InterpolationFactor in this example and DecimationFactor in sdruLTE2x2SIB1Rx.m
- Start MATLAB in -nodesktop mode before running this example
- If you are using one computer with two B210 radios, do not connect them to adjacent USB ports

### Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

# Multi-User Transmit Beamforming with USRP® Hardware

This example uses beamforming techniques to send two different payloads to two receivers simultaneously in the same frequency band. Channel estimates from the receivers are used continuously to update the transmitted beams.

The transmitter uses multiple USRP® radios as a composite radio with 4 channels. Two X300/X310 radios or four N-series radios are required. All radios for the transmitter must be connected to the same PPS and 10 MHz clock generator via cables of equal lengths.

Besides the transmitter radios, the host computer must be connected to two more USRP® radios, one for each receiver. The receiver radios must also be connected to the same 10 MHz clock generator.

To run this example, run `MultiUserBeamformingExample`, and then run `helperMUBeamformRx1` and `helperMUBeamformRx2` in two separate MATLAB sessions on the same host computer.

## Configure radios

Find attached radios and allocate radios for transmitter and receivers.

```
radioConfig = helperMUBeamformAllocateRadios;

% Save radio configuration to a file for helperMUBeamformRx1 and helperMUBeamformRx2
save(fullfile(tempdir,'helperMUBeamformRadioConfig.mat'), 'radioConfig');

transmitter = comm.SDRuTransmitter('Platform',radioConfig.txPlatform, ...
    'IPAddress',radioConfig.txIPAddrs);
transmitter.MasterClockRate = radioConfig.txMasterClockRate;
transmitter.InterpolationFactor = radioConfig.txInterpolationfactor;
transmitter.ChannelMapping = [1 2 3 4];
transmitter.CenterFrequency = 900e6;
transmitter.Gain = 8;
transmitter.ClockSource = 'External'; % Synchronize all 4 channels in frequency
transmitter.PPSSource = 'External'; % Synchronize all 4 channels in time

% Radio settings
transmitter

Checking radio connections...

transmitter =
    comm.SDRuTransmitter with properties:

        Platform: 'X310'
        IPAddress: '192.168.20.3,192.168.20.2'
        ChannelMapping: [1 2 3 4]
        CenterFrequency: 900000000
        LocalOscillatorOffset: 0
            Gain: 8
        PPSSource: 'External'
        ClockSource: 'External'
        MasterClockRate: 200000000
        InterpolationFactor: 500
        TransportDataType: 'int16'
        EnableBurstMode: false
```

## Construct training signals and payloads

```
trainingSig = helperMUBeamformInitGoldSeq; % Based on Gold Sequences

% Construct payload 1:
% 64 symbols with IFFT length of 256
% Each symbol uses 8 subcarriers
% Subcarrier 5 uses 64-QAM. Each point of 64-QAM is used once.
% Other subcarriers use QPSK
modOut1 = [zeros(4,64);
            qammod(randperm(64)-1,64,'UnitAveragePower',true); % 64-QAM
            qammod(randi([0 3],7,64),4,'UnitAveragePower',true); % QPSK
            zeros(256-4-8,64)];
payload1 = reshape(ifft(modOut1),[],1);
% Scale time-domain signal appropriately
payload1 = payload1/max(real(payload1))*0.5;

% Construct payload 2:
% 64 symbols with IFFT length of 256
% Each symbol uses 8 subcarriers
% Subcarrier 5 uses 16-QAM. Each point of 16-QAM is used 4 times.
% Other subcarriers use QPSK
modOut2 = [zeros(4,64);
            qammod([(randperm(16)-1) (randperm(16)-1) ...
                      (randperm(16)-1) (randperm(16)-1)], ...
                      16,'UnitAveragePower',true); % 16-QAM
            qammod(randi([0 3],7,64),4,'UnitAveragePower',true); % QPSK
            zeros(256-4-8,64)];
payload2 = reshape(ifft(modOut2),[],1);
% Scale time-domain signal appropriately
payload2 = payload2/max(real(payload2))*0.5;
```

## Initialize variables and temporary files for channel feedback

```
% Have no knowledge of the channel yet
% Generate channel estimate randomly
lastFeedback1 = rand(1,4) + 1j*rand(1,4);
fid1 = fopen(fullfile(tempdir,'helperMUBeamformfeedback1.bin'),'wb');
fwrite(fid1,[real(lastFeedback1) imag(lastFeedback1)],'double');
fclose(fid1);

% Have no knowledge of the channel yet
% Generate channel estimate randomly
lastFeedback2 = rand(1,4) + 1j*rand(1,4);
fid2 = fopen(fullfile(tempdir,'helperMUBeamformfeedback2.bin'),'wb');
fwrite(fid2,[real(lastFeedback2) imag(lastFeedback2)],'double');
fclose(fid2);

% Open files for reading only
fid1 = fopen(fullfile(tempdir,'helperMUBeamformfeedback1.bin'),'rb');
fid2 = fopen(fullfile(tempdir,'helperMUBeamformfeedback2.bin'),'rb');
```

## Main loop

```
disp('Sending two different payloads to two receivers simultaneously in the same frequency band')
disp('Channel estimates from the receivers are used continuously to update the transmitted beams')
disp('Please run helperMUBeamformRx1 and helperMUBeamformRx2 in two separate MATLAB sessions on ...')
```

```

for i = 1:30000
    fseek(fid1,0,'bof'); % Read from the beginning of the file
    % The channel between 4 TX antennas and 1 RX antenna is modeled
    % by 4 complex gains. This approximation works because the
    % signal has very narrow bandwidth (400k samples per second).
    channelEst1 = fread(fid1,8,'double');
    if length(channelEst1) == 8
        % File content has expected length
        channelEst1 = (channelEst1(1:4) + 1j*channelEst1(5:8)).';
        lastFeedback1 = channelEst1;
    else
        % Use last feedback
        channelEst1 = lastFeedback1;
    end

    fseek(fid2,0,'bof'); % Read from the beginning of the file
    % The channel between 4 TX antennas and 1 RX antenna is modeled
    % by 4 complex gains. This approximation works because the
    % signal has very narrow bandwidth (400k samples per second).
    channelEst2 = fread(fid2,8,'double');
    if length(channelEst2) == 8
        % File content has expected length
        channelEst2 = (channelEst2(1:4) + 1j*channelEst2(5:8)).';
        lastFeedback2 = channelEst2;
    else
        % Use last feedback
        channelEst2 = lastFeedback2;
    end

    % For payload 1, create a spectral null at receiver 2 by
    % inverting the channel response and applying phase offsets of
    % 0, pi/2, pi, and 3*pi/2 to achieve destructive interference.
    % Hence, payload 1 will be suppressed for receiver 2.
    beamWeight1 = 1./channelEst2;
    beamWeight1 = beamWeight1/max(abs(beamWeight1)); % Normalize the gain
    beamWeight1 = beamWeight1 .* [1 1j -1 -1j];

    % Rotate payload 1 so that receiver 1 does not need to
    % correct the phase of payload 1
    phaseCorrection1 = beamWeight1 * channelEst1.';
    phaseCorrection1 = phaseCorrection1/abs(phaseCorrection1);
    beamWeight1 = beamWeight1 ./ phaseCorrection1;

    % For payload 2, create a spectral null at receiver 1 by
    % inverting the channel response and applying phase offsets of
    % 0, pi/2, pi, and 3*pi/2 to achieve destructive interference
    % Hence, payload 2 will be suppressed for receiver 1.
    beamWeight2 = 1./channelEst1;
    beamWeight2 = beamWeight2/max(abs(beamWeight2));
    beamWeight2 = beamWeight2 .* [1 1j -1 -1j];

    % Rotate payload 2 so that receiver 2 does not need to
    % correct the phase of payload 2
    phaseCorrection2 = beamWeight2 * channelEst2.';
    phaseCorrection2 = phaseCorrection2/abs(phaseCorrection2);
    beamWeight2 = beamWeight2 ./ phaseCorrection2;

    % Beamforming for payload

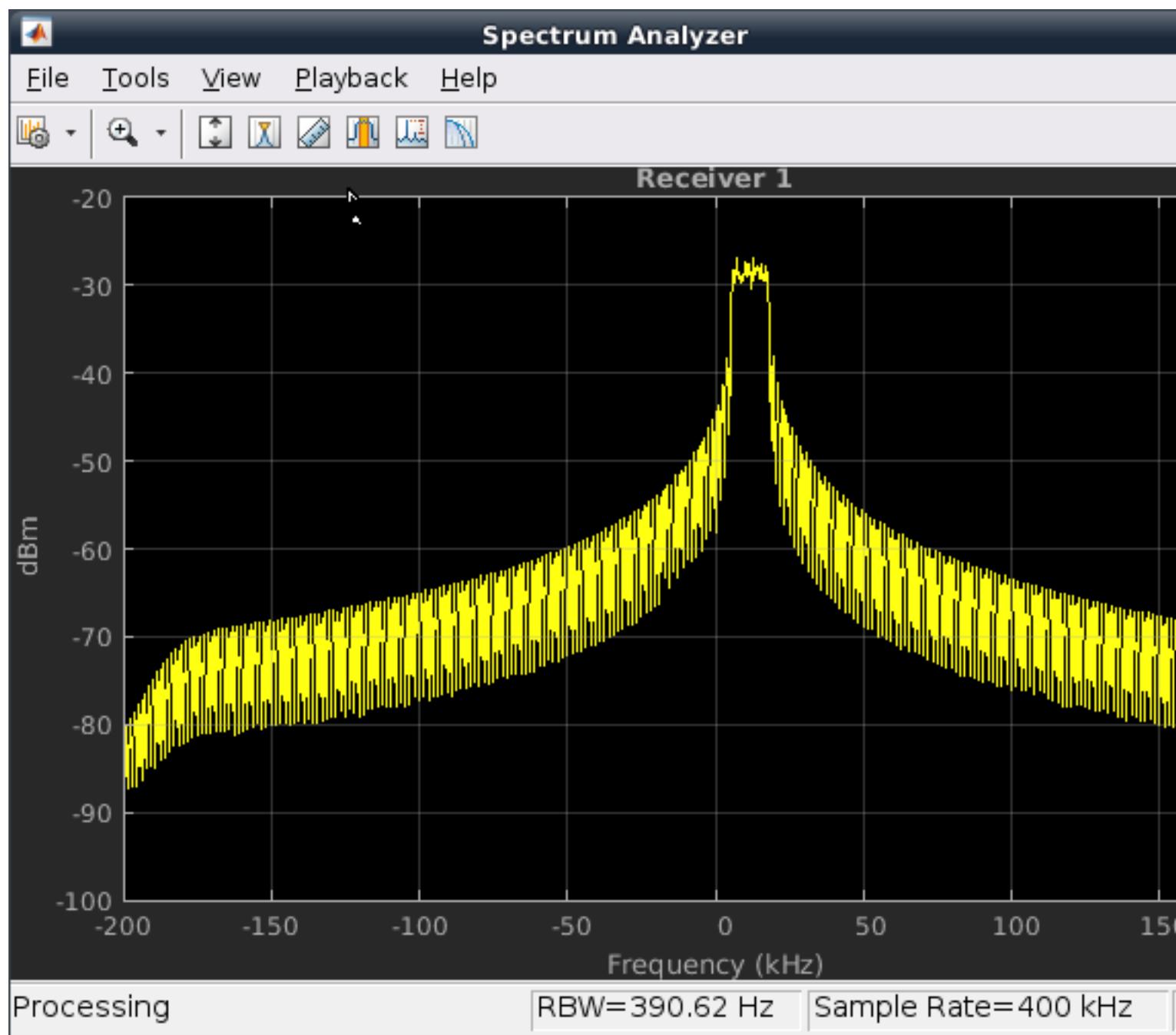
```

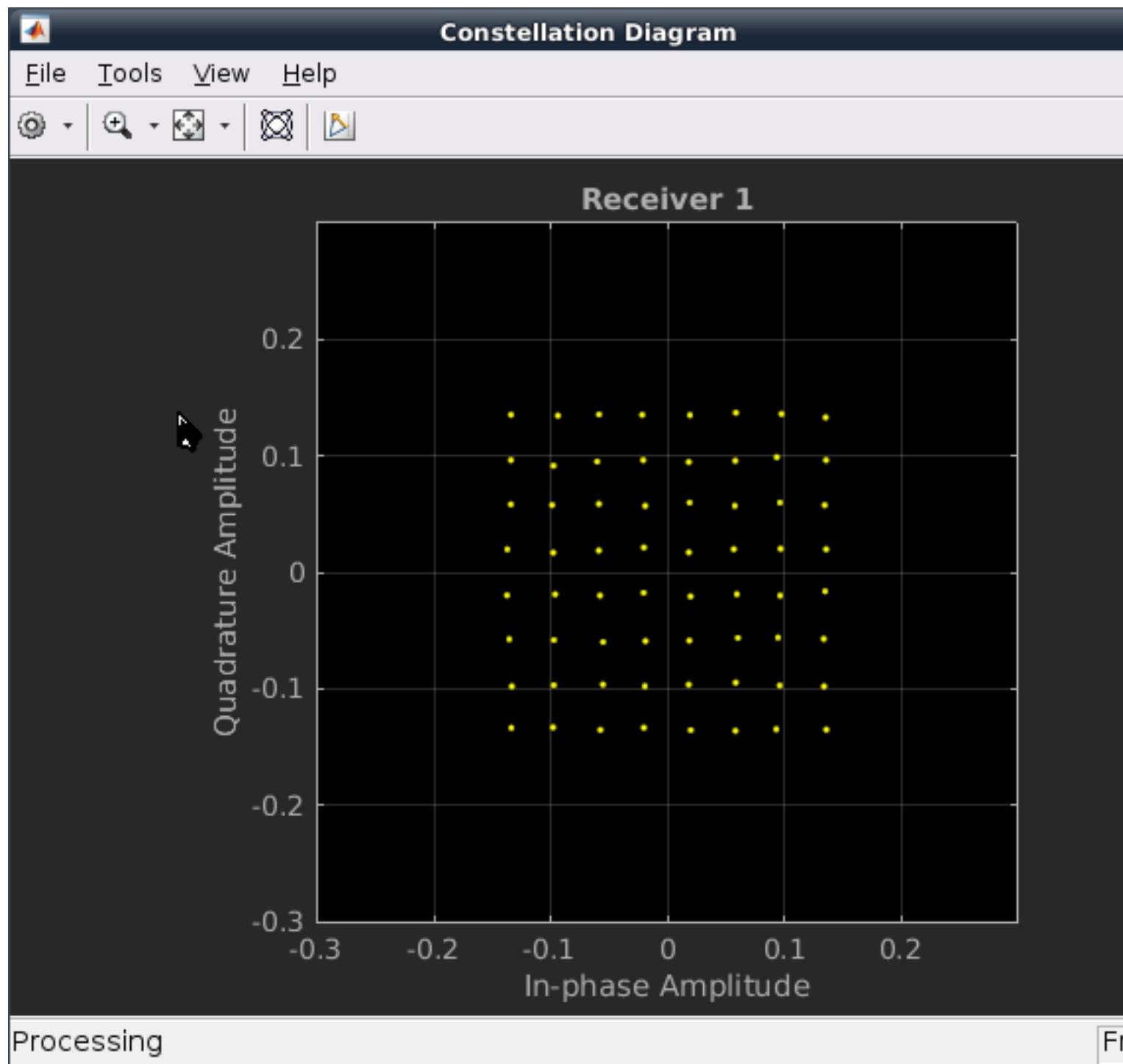
```
payload = payload1*beamWeight1 + payload2*beamWeight2;

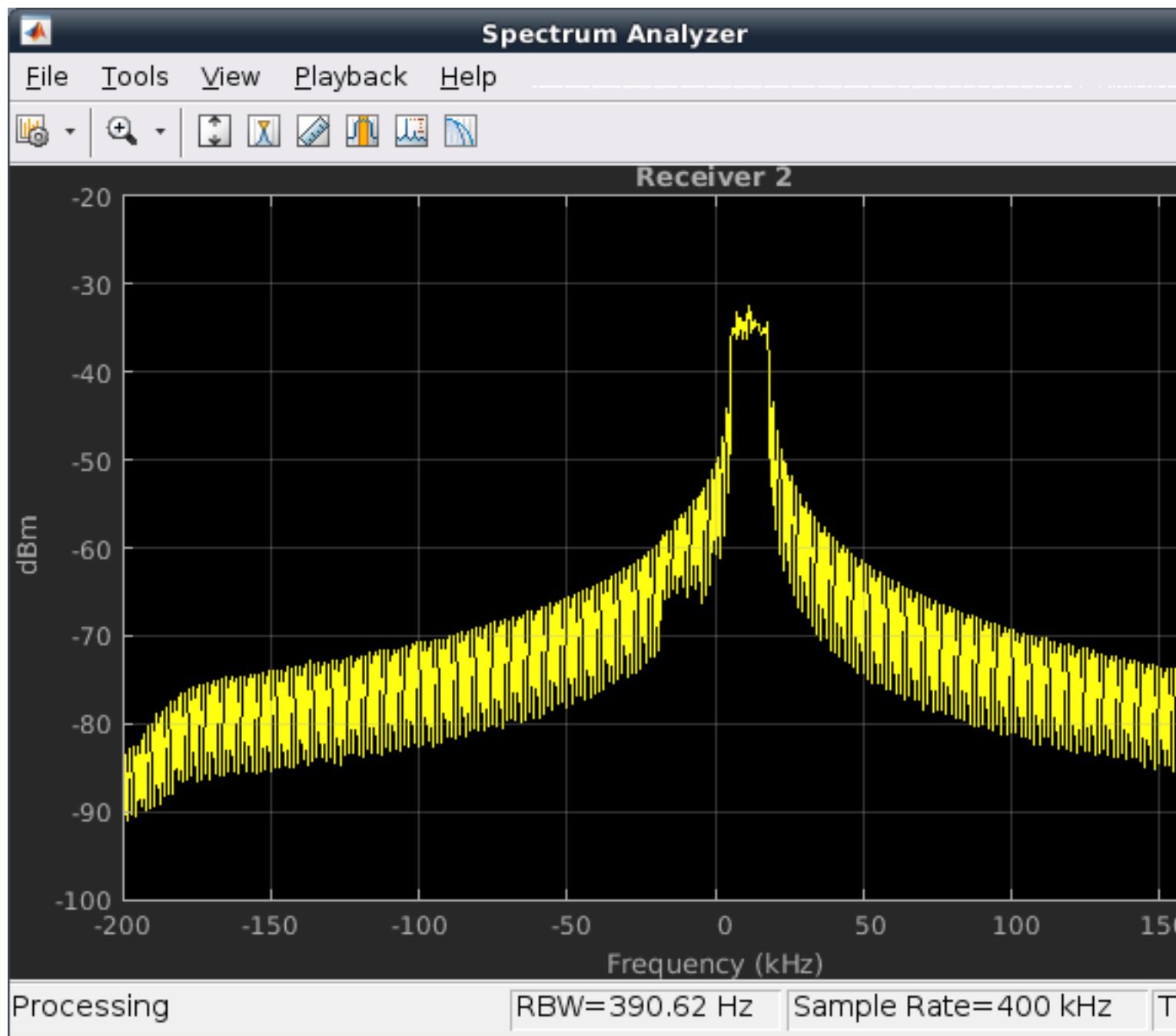
% Send signals to the radios
txSig = [trainingSig; zeros(400,4); payload; zeros(100,4)] * 0.2;
transmitter(txSig);
end

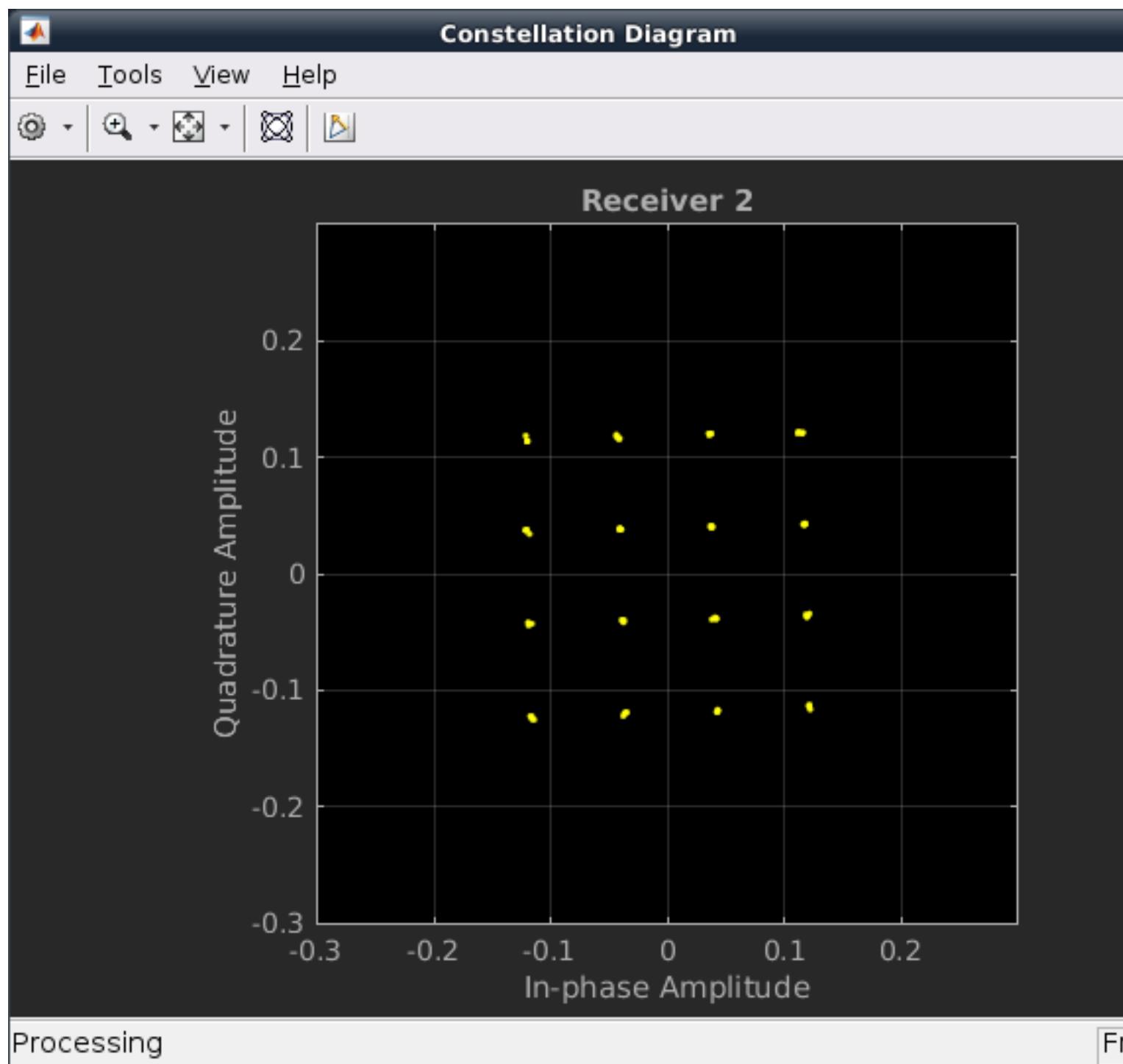
release(transmitter);
```

Sending two different payloads to two receivers simultaneously in the same frequency band ...  
Channel estimates from the receivers are used continuously to update the transmitted beams.  
Please run helperMUBeamformRx1 and helperMUBeamformRx2 in two separate MATLAB sessions on this computer.









### Appendix

This example uses the following helper functions:

- helperMUBeamformAllocateRadios.m

- helperMUBeamformInitGoldSeq.m
- helperMUBeamformRx1.m
- helperMUBeamformRx2.m
- helperMUBeamformEstimateChannel.m

### **Copyright Notice**

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

## QPSK Receiver with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® (USRP®) device using SDRu (Software Defined Radio USRP®) System objects to implement a QPSK receiver. The receiver addresses practical issues in wireless communications, such as carrier frequency and phase offset, timing offset and frame synchronization. This system receives the signal sent by the QPSK Transmitter with USRP® Hardware example. The receiver demodulates the received symbols and prints a simple message to the MATLAB® command line.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver System object.

### Implementations

This example describes the MATLAB implementation of a QPSK receiver with USRP® Hardware. There is another implementation of this example that uses Simulink®.

MATLAB script using System objects: `sdruQPSKReceiver.m`.

Simulink implementation using blocks: `sdruqpskrx.mdl`.

You can also explore a no-USRP QPSK Transmitter and Receiver example that models a general wireless communication system using an AWGN channel and simulated channel impairments at `commQPSKTransmitterReceiver.m`.

### Introduction

This example has the following motivation:

- To implement a real QPSK-based transmission-reception environment in MATLAB using SDRu System objects.
- To illustrate the use of key Communications Toolbox™ System objects for QPSK system design, including coarse and fine carrier frequency compensation, closed-loop timing recovery with bit stuffing and stripping, frame synchronization, carrier phase ambiguity resolution, and message decoding.

In this example, the SDRuReceiver System object receives data corrupted by the transmission over the air and outputs complex baseband signals which are processed by the QPSK Receiver System object. This example provides a reference design of a practical digital receiver that can cope with wireless channel impairments. The receiver includes FFT-based coarse frequency compensation, PLL-based fine frequency compensation, timing recovery with fixed-rate resampling and bit stuffing/ skipping, frame synchronization, and phase ambiguity resolution.

### Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdr` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system.

```
connectedRadios = findsdr;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
```

```

platform = connectedRadios(1).Platform;
case {'N200/N210/USRP2'}
    address = connectedRadios(1).IPAddress;
    platform = 'N200/N210/USRP2';
case {'X300','X310'}
    address = connectedRadios(1).IPAddress;
    platform = connectedRadios(1).Platform;
end
else
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end

```

## Initialization

The `sdrqpskreceiver_init.m` script initializes the simulation parameters and generates the structure `prmQPSKReceiver`.

```

% Receiver parameter structure
prmQPSKReceiver = sdrqpskreceiver_init(platform)
prmQPSKReceiver.Platform = platform;
prmQPSKReceiver.Address = address;
compileIt = false; % true if code is to be compiled for accelerated execution
useCodegen = false; % true to run the latest generated code (mex file) instead of MATLAB code

prmQPSKReceiver =

```

MasterClockRate:	20000000
M:	4
Upsampling:	4
Downsampling:	2
Fs:	200000
Ts:	5.0000e-06
FrameSize:	100
BarkerLength:	13
DataLength:	174
MessageLength:	105
FrameCount:	100
ScramblerBase:	2
ScramblerPolynomial:	[1 1 1 0 1]
ScramblerInitialConditions:	[0 0 0 0]
RxBuffedFrames:	10
RCFiltSpan:	10
SquareRootRaisedCosineFilterOrder:	40
RollOff:	0.5000
ReceiverFilterCoefficients:	[1x41 double]
PhaseErrorDetectorGain:	2.0000
PhaseRecoveryGain:	1
TimingErrorDetectorGain:	5.4000
TimingRecoveryGain:	-1
CoarseCompFrequencyResolution:	50
PhaseRecoveryLoopBandwidth:	0.0100
PhaseRecoveryDampingFactor:	1
TimingRecoveryLoopBandwidth:	0.0100
TimingRecoveryDampingFactor:	1
USRPCenterFrequency:	1.8500e+09

```
USRPGain: 31
USRPDecimationFactor: 100
USRPFrontEndSampleRate: 5.0000e-06
USRPFrmLength: 4000
FrameTime: 0.0200
StopTime: 5
```

To transmit successfully, ensure that the specified center frequency of the SDRu Receiver is within the acceptable range of your USRP® daughterboard.

Also, by using the `compileIt` and `useCodegen` flags, you can interact with the code to explore different execution options. Set the MATLAB variable `compileIt` to true in order to generate C code; this can be accomplished by using the **codegen** command provided by the MATLAB Coder™ product. The **codegen** command compiles MATLAB® functions to a C-based static or dynamic library, executable, or MEX file, producing code for accelerated execution. The generated executable runs several times faster than the original MATLAB code. Set `useCodegen` to true to run the executable generated by **codegen** instead of the MATLAB code.

### Code Architecture

The function `runSDRuQPSKReceiver` implements the QPSK receiver using two System objects, `QPSKReceiver` and `comm.SDRuReceiver`.

### SDRu Receiver

This example communicates with the USRP® board using the SDRu receiver System object. The parameter structure `prmQPSKReceiver` sets the `CenterFrequency`, `Gain`, and `InterpolationFactor` arguments.

### QPSK Receiver

This component regenerates the original transmitted message. It is divided into five subcomponents, modeled using System objects. Each subcomponent is modeled by other subcomponents using System objects.

- 1) Automatic Gain Control: Sets its output amplitude to  $1/\sqrt{(\text{Upsampling Factor})}$  (0.5), so that the equivalent gains of the phase and timing error detectors keep constant over time. The AGC is placed before the **Raised Cosine Receive Filter** so that the signal amplitude can be measured with an oversampling factor of four. This process improves the accuracy of the estimate.
- 2) Coarse frequency compensation: Uses nonlinearity and a Fast Fourier Transform (FFT) to roughly estimate the frequency offset and then compensate for it. The object raises the input signal to the power of four to obtain a signal that is not a function of the QPSK modulation. Then it performs an FFT on the modulation-independent signal to estimate the tone at four times the frequency offset. After dividing the estimate by four, the **Phase/Frequency Offset** System object corrects the frequency offset.
- 3) Fine frequency compensation: Performs closed-loop scalar processing and compensates for the frequency offset accurately. The Fine Frequency Compensation object implements a phase-locked loop (PLL) to track the residual frequency offset and the phase offset in the input signal. For more information, see Chapter 7 of [ 1 ]. The PLL uses a **Direct Digital Synthesizer (DDS)** to generate the compensating phase that offsets the residual frequency and phase offsets. The phase offset estimate from **DDS** is the integral of the phase error output of the **Loop Filter**. To obtain details of PLL design, refer to Appendix C.2 of [ 1 ].

4) Timing recovery: Performs timing recovery with closed-loop scalar processing to overcome the effects of delay introduced by the channel. The **Timing Recovery** object implements a PLL, described in Chapter 8 of [ 1 ], to correct the timing error in the received signal. The **NCO Control** object implements a decrementing modulo-1 counter described in Chapter 8.4.3 of [ 1 ] to generate the control signal for the **Modified Buffer** to select the interpolants of the **Interpolation Filter**. This control signal also enables the **Timing Error Detector (TED)**, so that it calculates the timing errors at the correct timing instants. The **NCO Control** object updates the timing difference for the **Interpolation Filter**, generating interpolants at optimum sampling instants. The **Interpolation Filter** is a Farrow parabolic filter with alpha set to 0.5 as described in Chapter 8.4.2 of [ 1 ]. Based on the interpolants, timing errors are generated by a zero-crossing **Timing Error Detector** as described in Chapter 8.4.1 of [ 1 ], filtered by a tunable proportional-plus-integral **Loop Filter** as described in Appendix C.2 of [ 1 ], and fed into the **NCO Control** for a timing difference update. The *Loop Bandwidth* (normalized by the sample rate) and *Loop Damping Factor* are tunable for the **Loop Filter**. The default normalized loop bandwidth is set to 0.01 and the default damping factor is set to 1 for critical damping. These settings ensure that the PLL quickly locks to the correct timing while introducing little phase noise.

5) Data decoder: Uses a Barker code to perform frame synchronization, phase ambiguity resolution, and demodulation. Also, the data decoder compares the regenerated message with the transmitted message and calculates the BER.

For more information about the system components, refer to the QPSK Receiver with USRP® Hardware example using Simulink.

## Execution and Results

Before running the script, first turn on the USRP® and connect it to the computer. To ensure data reception, first start the QPSK Transmitter with USRP® Hardware example.

```

if compileIt
    codegen('runSDRuQPSKReceiver', '-args', {coder.Constant(prmQPSKReceiver)});
end
if useCodegen
    clear runSDRuQPSKReceiver_mex %#ok<UNRCH>
    BER = runSDRuQPSKReceiver_mex(prmQPSKReceiver);
else
    BER = runSDRuQPSKReceiver(prmQPSKReceiver);
end

fprintf('Error rate is = %f.\n',BER(1));
fprintf('Number of detected errors = %d.\n',BER(2));
fprintf('Total number of compared samples = %d.\n',BER(3));

```

When you run the simulations, the received messages are decoded and printed out in the MATLAB command window while the simulation is running. BER information is also shown at the end of the script execution. The calculation of the BER value includes the first received frames, when some of the adaptive components in the QPSK receiver still have not converged. During this period, the BER is quite high. Once the transient period is over, the receiver is able to estimate the transmitted frame and the BER dramatically improves. In this example, to guarantee a reasonable execution time of the system in simulation mode, the simulation duration is fairly short. As such, the overall BER results are significantly affected by the high BER values at the beginning of the simulation. To increase the simulation duration and obtain lower BER values, you can change the SimParams.StopTime variable in the receiver initialization file.

Also, the gain behavior of different USRP® daughter boards varies considerably. Thus, the gain setting in the transmitter and receiver defined in this example may not be well-suited for your daughter boards. If the message is not properly decoded by the receiver system, you can vary the gain of the source signals in the **SDRu Transmitter** and **SDRu Receiver** System objects by changing the SimParams.USRPGain value in the transmitter initialization file and in the receiver initialization file.

Finally, a large relative frequency offset between the transmit and receive USRP® radios can prevent the receiver functions from properly decoding the message. If that happens, you can determine the offset by sending a tone at a known frequency from the transmitter to the receiver, then measuring the offset between the transmitted and received frequency, then applying that offset to the center frequency of the SDRu Receiver System object.

## Appendix

This example uses the following script and helper functions:

- runSDRuQPSKReceiver.m
- sdruqpskreceiver\_init.m
- QPSKReceiver.m

## References

1. Rice, Michael. *Digital Communications - A Discrete-Time Approach*. 1st ed. New York, NY: Prentice Hall, 2008.

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

# QPSK Transmitter with USRP® Hardware

This example shows how to use the Universal Software Radio Peripheral® (USRP®) device using SDRu (Software Defined Radio USRP®) System objects to implement a QPSK transmitter. The USRP® device in this system will keep transmitting indexed "Hello world" messages at its specified center frequency. You can demodulate the transmitted message using the QPSK Receiver with USRP® Hardware example with an additional USRP® device.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter System object.

## Implementations

This example describes the MATLAB® implementation of a QPSK transmitter with USRP® hardware. There is another implementation of this example that uses Simulink®.

MATLAB script using System objects: `sdruQPSKTransmitter.m`.

Simulink implementation using blocks: `sdruqpsktx.mdl`.

You can also explore a no-USRP QPSK Transmitter and Receiver example that models a general wireless communication system using an AWGN channel and simulated channel impairments at `commQPSKTransmitterReceiver.m`.

## Introduction

This example has the following motivation:

- To implement a real QPSK-based transmission-reception environment in MATLAB using SDRu System objects.
- To illustrate the use of key Communications Toolbox™ System objects for QPSK system design.

In this example, the transmitter generates a message using ASCII characters, converts the characters to bits, and prepends a Barker code for receiver frame synchronization. This data is then modulated using QPSK and filtered with a square root raised cosine filter. The filtered QPSK symbols can be transmitted over the air using the SDRu transmitter System object and the USRP® hardware.

## Discover Radio

Discover radio(s) connected to your computer. This example uses the first USRP® radio found using the `findsdr` function. Check if the radio is available and record the radio type. If no available radios are found, the example uses a default configuration for the system.

```
connectedRadios = findsdr;
if strncmp(connectedRadios(1).Status, 'Success', 7)
    switch connectedRadios(1).Platform
        case {'B200','B210'}
            address = connectedRadios(1).SerialNum;
            platform = connectedRadios(1).Platform;
        case {'N200/N210/USRP2'}
            address = connectedRadios(1).IPAddress;
            platform = 'N200/N210/USRP2';
        case {'X300','X310'}
            address = connectedRadios(1).IPAddress;
            platform = connectedRadios(1).Platform;
```

```
    end
else
    address = '192.168.10.2';
    platform = 'N200/N210/USRP2';
end
```

## Initialization

The `sdrqpsktransmitter_init.m` script initializes the simulation parameters and generates the structure `prmQPSKTransmitter`.

```
% Transmitter parameter structure
prmQPSKTransmitter = sdrqpsktransmitter_init(platform)
prmQPSKTransmitter.Platform = platform;
prmQPSKTransmitter.Address = address;
compileIt = false; % true if code is to be compiled for accelerated execution
useCodegen = false; % true to run the latest generated mex file

prmQPSKTransmitter =
    MasterClockRate: 20000000
        Upsampling: 4
            Fs: 200000
            Ts: 5.0000e-06
        FrameSize: 100
        BarkerLength: 13
        DataLength: 174
        MessageLength: 105
        FrameCount: 100
        RxBufferedFrames: 10
        RCFiltSpan: 10
        ScramblerBase: 2
        ScramblerPolynomial: [1 1 1 0 1]
        ScramblerInitialConditions: [0 0 0 0]
    SquareRootRaisedCosineFilterOrder: 40
        RollOff: 0.5000
    TransmitterFilterCoefficients: [1x41 double]
        USRPCenterFrequency: 1.8500e+09
        USRPGain: 25
    USRPInterpolationFactor: 100
        USRPFrameLength: 4000
        FrameTime: 0.0200
        StopTime: 1000
```

To achieve a successful transmission, ensure that the specified center frequency of the SDRu Transmitter is within the acceptable range of your USRP® daughterboard.

Also, by using the `compileIt` and `useCodegen` flags, you can interact with the code to explore different execution options. Set the MATLAB variable `compileIt` to true in order to generate C code; this can be accomplished by using the **codegen** command provided by the MATLAB Coder™ product. The **codegen** command compiles MATLAB® functions to a C-based static or dynamic library, executable, or MEX file, producing code for accelerated execution. The generated executable runs several times faster than the original MATLAB code. Set `useCodegen` to true to run the executable generated by **codegen** instead of the MATLAB code.

## Code Architecture

The function `runSDRuQPSKTransmitter` implements the QPSK transmitter using two System objects, `QPSKTransmitter` and `comm.SDRuTransmitter`.

### QPSK Transmitter

The transmitter includes the **Bit Generation**, **QPSK Modulator** and **Raised Cosine Transmit Filter** objects. The **Bit Generation** object generates the data frames. Each frame contains 200 bits. The first 26 bits are header bits, a 13-bit Barker code that has been oversampled by two. The Barker code is sent on both in-phase and quadrature components of the QPSK modulated symbols. This is achieved by repeating the Barker code bits twice before modulating them with the QPSK modulator.

The remaining bits are the payload. The first 105 bits of the payload correspond to the ASCII representation of 'Hello world ####', where '####' is an incrementing sequence of '001', '002', '003',..., '100'. The remaining payload bits are random bits. The payload is scrambled to guarantee a balanced distribution of zeros and ones for the timing recovery operation in the receiver object. The scrambled bits are modulated by the **QPSK Modulator** (with Gray mapping). The **Raised Cosine Transmit Filter** upsamples the modulated symbols by four, and has a roll-off factor of 0.5. The output rate of the **Raised Cosine Filter** is set to be 200e3 samples per second.

### SDRu Transmitter

The host computer communicates with the USRP® radio using the SDRu transmitter System object. The `CenterFrequency`, `Gain`, and `InterpolationFactor` arguments are set by the parameter variable `prmQPSKTransmitter`.

### Execution

Before running the script, first turn on the USRP® radio and connect it to the computer. As already mentioned, you can check the correct data transmission by running the QPSK Receiver with USRP® Hardware example while running the transmitter script.

```
if compileIt
    codegen('runSDRuQPSKTransmitter', '-args', {coder.Constant(prmQPSKTransmitter)}); %#ok<UNRCH>
end
if useCodegen
    clear runSDRuQPSKTransmitter_mex %#ok<UNRCH>
    runSDRuQPSKTransmitter_mex(prmQPSKTransmitter);
else
    runSDRuQPSKTransmitter(prmQPSKTransmitter);
end
```

The gain behavior of different USRP® daughter boards varies considerably. Thus, the gain setting in the transmitter and receiver defined in this example may not be well suited for your daughter boards. If the message is not properly decoded by the receiver object, you can vary the gain of the source signals in the **SDRu Transmitter** and **SDRu Receiver** System objects by changing the `SimParams.USRPGain` value in the transmitter initialization file and in the receiver initialization file.

Also, a large relative frequency offset between the transmit and receive USRP® radios can prevent the receiver functions from properly decoding the message. If that happens, you can determine the offset by sending a tone at a known frequency from the transmitter to the receiver, then measuring the offset between the transmitted and received frequency, then applying that offset to the center frequency of the SDRu Receiver System object.

## Appendix

This example uses the following script and helper functions:

- runSDRuQPSKTransmitter.m
- sdruqpsktransmitter\_init.m
- QPSKTransmitter.m
- QPSKBitsGenerator.m

## Copyright Notice

Universal Software Radio Peripheral® and USRP® are trademarks of National Instruments Corp.

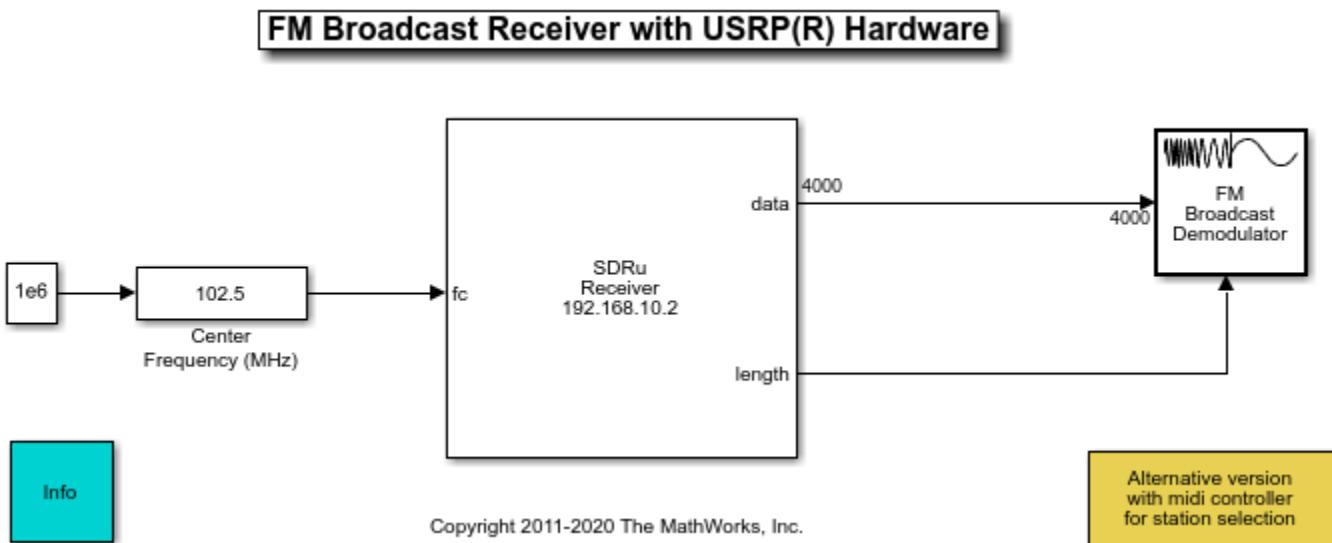
# FM Receiver with USRP® Hardware

This model shows how to use the Universal Software Radio Peripheral® (USRP®) device with Simulink® to build an FM receiver.

In order to run this model, you need a USRP® board with an appropriate receiver daughterboard that supports the FM band (e.g., TVRX or WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver block.

## Structure of the Example

This is the top-level block diagram of the model.



The SDRu Receiver block takes in the baseband discrete-time complex samples from the USRP® hardware. The master clock rate and decimation factor are set to obtain sample rate of 200 kHz at the output of the SDRu Receiver block. For example, for a B210 radio, set MasterClockRate to 20 MHz and DecimationRate to 100. For N200, N210, and USRP2 radios master clock rate is fixed at 100 MHz.

The FM Broadcast Demodulator Baseband block converts the sampling rate of 240 kHz to 48 kHz, a native sampling rate for your host computer's audio device. According to the FM broadcast standard in the United States, the deemphasis lowpass filter time constant is set to 75 microseconds.

To perform stereo decoding, the FM Broadcast Demodulator Baseband block uses a peaking filter which picks out the 19 kHz pilot tone from which the 38 kHz carrier is created. Using the obtained carrier signal, the block downconverts the L-R signal, centered at 38 kHz, to baseband. Afterwards, the L-R and L+R signals pass through a 75 microsecond deemphasis filter. The FM Broadcast Demodulator Baseband block separates the L and R signals and converts them to the 48 kHz audio signal. Note that, if you uncheck the Stereo audio parameter of the block, the demodulator process the signals in a mono fashion.

Set the Center Frequency to a local FM radio station, click the run button, and listen to the sound from the audio device. Change the Center Frequency to listen to a different station.

If you hear some dropouts or delay in the sound, run the model under Accelerator mode. From the model menu, select Simulation->Accelerator, then click the run button.

### **Exploring the Example**

If you have your own FM transmitter that can transmit .wma files, you can duplicate the test that shows the channel separation result above. Load the `sdruFMStereoTestSignal.wma` file into your transmitter. The channel separation can be easily observed from the Spectrum Scope block and heard from the audio device. You can also adjust the Gain Compensation to see its effect on stereo separation.

### **MIDI Controller for Station Selection**

The FM Receiver with USRP® Hardware and MIDI Controller example includes a **MIDI Controls** block. If you have a MIDI controller connected to your computer, you can change the center frequency, i.e. FM radio station, using the controller.

### **References**

- FM broadcasting on Wikipedia

### **Copyright Notice**

USRP® is a trademark of National Instruments Corp.

# Frequency Offset Calibration with USRP® Hardware

These two models show how to determine the relative frequency offset between two Universal Software Radio Peripheral® (USRP®) devices using Simulink®.

The transmitter sends a 100 Hz sine wave with the Frequency Offset Calibration (Tx) with USRP® Hardware model. The receiver receives the signal, calculates the frequency offset and displays the offset in the Frequency Offset Calibration (Rx) with USRP® Hardware model.

In order to run these two models, you need to ensure that the specified center frequency of the SDRu Transmitter and Receiver blocks is within the acceptable range of your USRP® daughter board.

Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter and Receiver blocks.

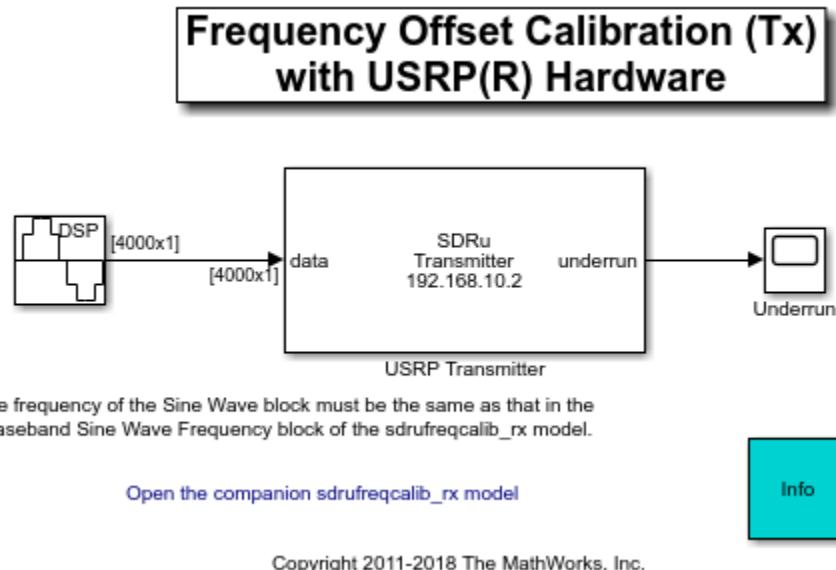
## Overview

These two models perform an FFT-based frequency offset calculation at complex baseband. The receiver model provides the following information:

- The quantitative value of the frequency offset
- A graphical view of the spur-free dynamic range of the receiver
- A graphical view of the qualitative SNR level of the received signal

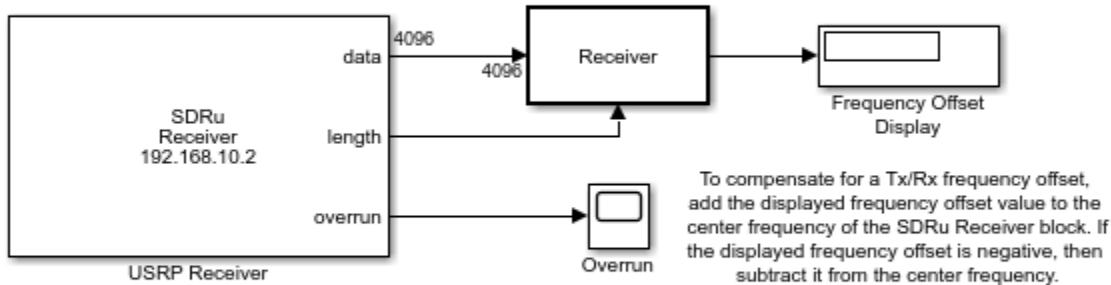
## Structure of the Example

The following figure shows the transmitter model:



The following figure shows the receiver model:

## Frequency Offset Calibration (Rx) with USRP(R) Hardware

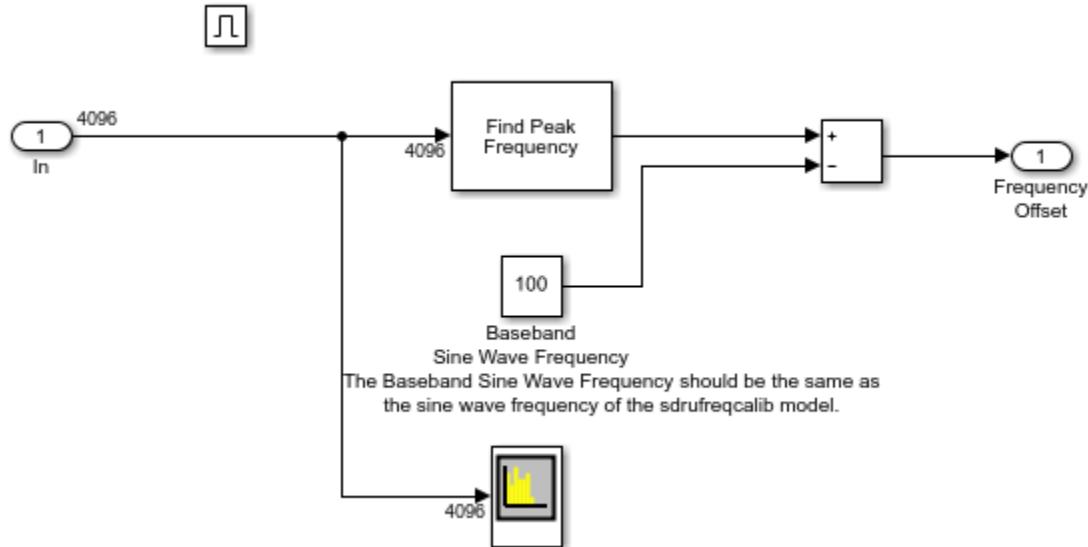


[Open the companion sdrufreqcalib model](#)

Copyright 2011-2018 The MathWorks, Inc.



The following figure shows the detailed structure of the **Receiver** subsystem:



- The **Find Peak Frequency** block - uses an FFT to find the frequency with the maximum power in the received signal.
- The **Spectrum Analyzer** block - computes and displays the power spectral density of the received signal.

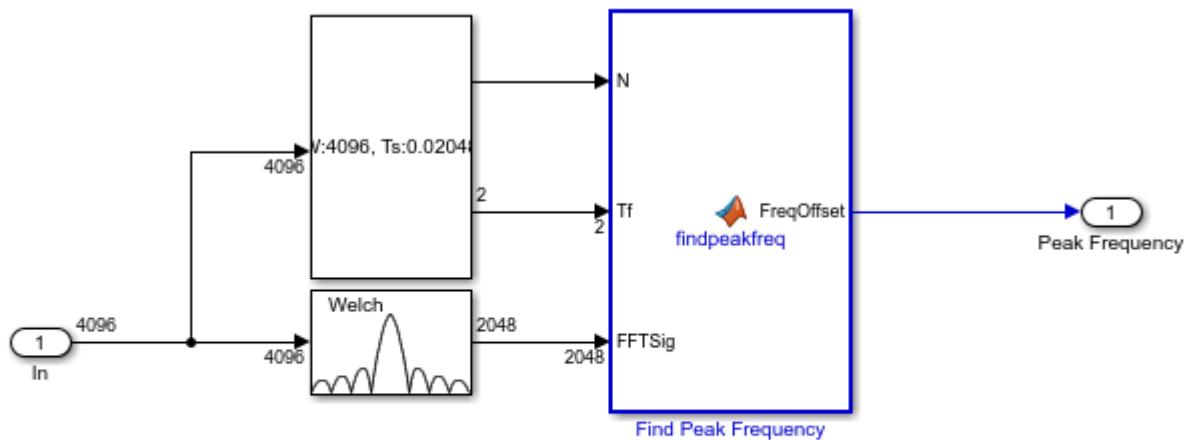
## Receiver

### Find Peak Frequency

The **Find Peak Frequency** subsystem finds the frequency with the maximum power in the received signal, which equals the frequency offset plus 100 Hz. The following diagram shows the subsystem. In this subsystem, the Periodogram block returns the PSD estimate of the received signal. The Probe block finds the frame size and the frame sample time. With this information, this subsystem finds the index of the maximum amplitude across the frequency band and converts the index to the frequency value according to

$$F_{\text{offset}} = \text{IndexofMaxAmplitude} * \text{FrameSize} / (\text{FFTLength} * \text{FrameSampleTime})$$

The MATLAB function `findpeakfreq.m` performs this conversion.



This subsystem does the following:

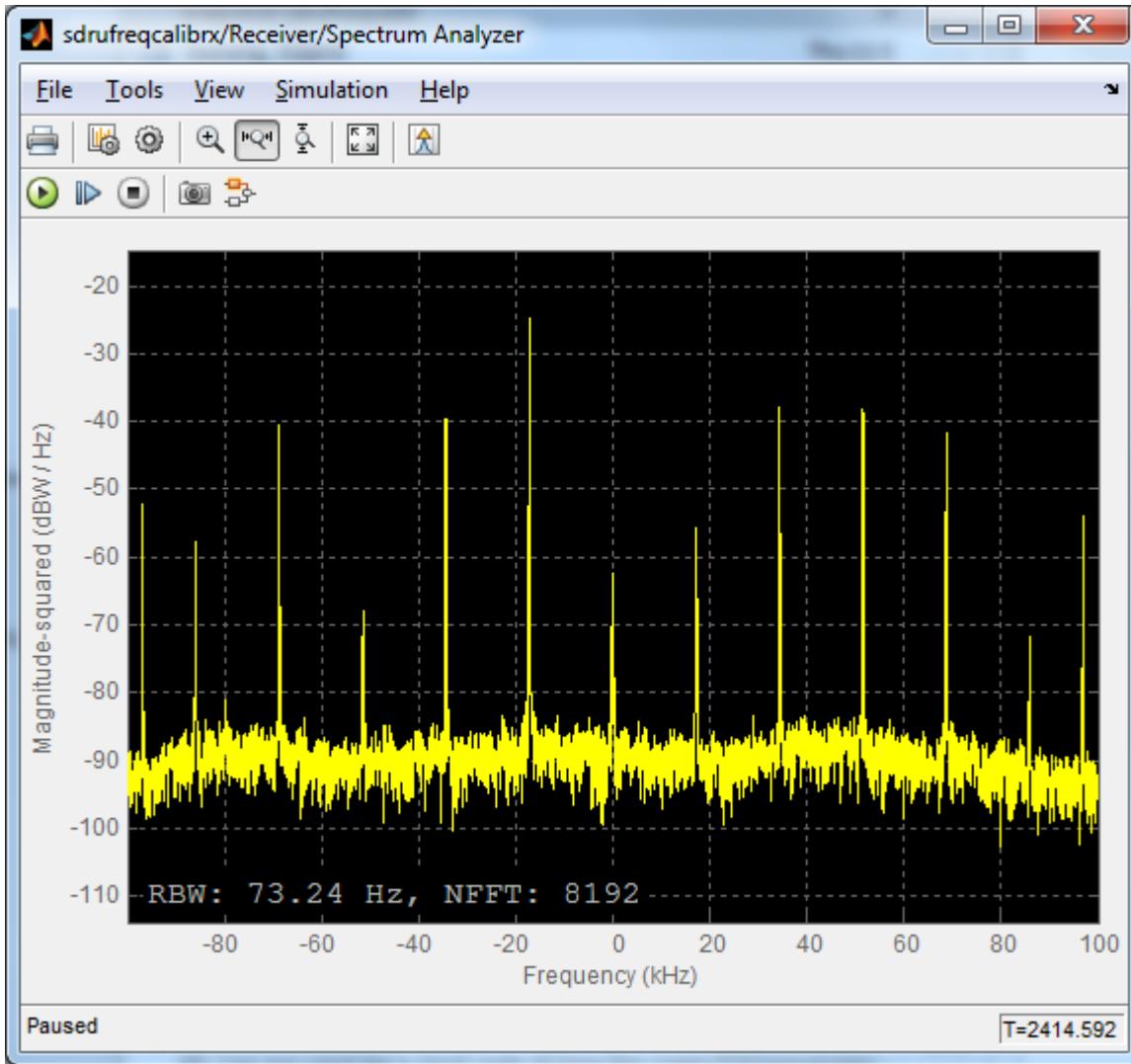
1. Finds the index of the maximum amplitude across the frequency band

2. Converts the index to the frequency offset according to

$$F_{\text{offset}} = \text{IndexofMaxAmplitude} * \text{FrameSize} / (\text{FFTLength} * \text{FrameSampleTime})$$

### Spectrum Analyzer

The following figure shows the output of the Spectrum Analyzer on a frequency range of -100 kHz to 100 kHz. In the case shown below, the frequency with the maximum power of the received signal is about -17 kHz, and the spur-free dynamic range of the receiver is about 14 dB.



### Running the Example

In order to calibrate the frequency offset between two USRP® devices, first start the Frequency Offset Calibration (Tx) with USRP® Hardware model on one USRP® radio, and then start the Frequency Offset Calibration (Rx) with USRP® Hardware model on another USRP® radio.

To run the receiver model, set the *Center frequency* parameter of the **SDRu Receiver** block to the same value as the center frequency setting of the Frequency Offset Calibration (Tx) with USRP® Hardware model. Then run the model. The frequency offset is calculated and displayed while the simulation is running.

To compensate for a transmitter/receiver frequency offset, add the displayed frequency offset value to the center frequency of the SDRu Receiver block. If the displayed frequency offset is negative, then subtract it from the center frequency. The spectrum displayed by the Spectrum Analyzer block should then have its maximum at 0 Hz.

### **Copyright Notice**

USRP® is a trademark of National Instruments Corp.

## FRS/GMRS Walkie-Talkie Receiver with USRP® Hardware

This model shows how to use the Universal Software Radio Peripheral® (USRP®) device with Simulink® to build a walkie-talkie that can receive messages from a physical walkie-talkie. The specific radio standard that this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System).

In order to run this model, you need a USRP® board with an appropriate receiver daughterboard that supports the UHF 462-467 MHz band (for example, WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Receiver block.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found at [1] and [2]. Operation in other countries may or may not work.

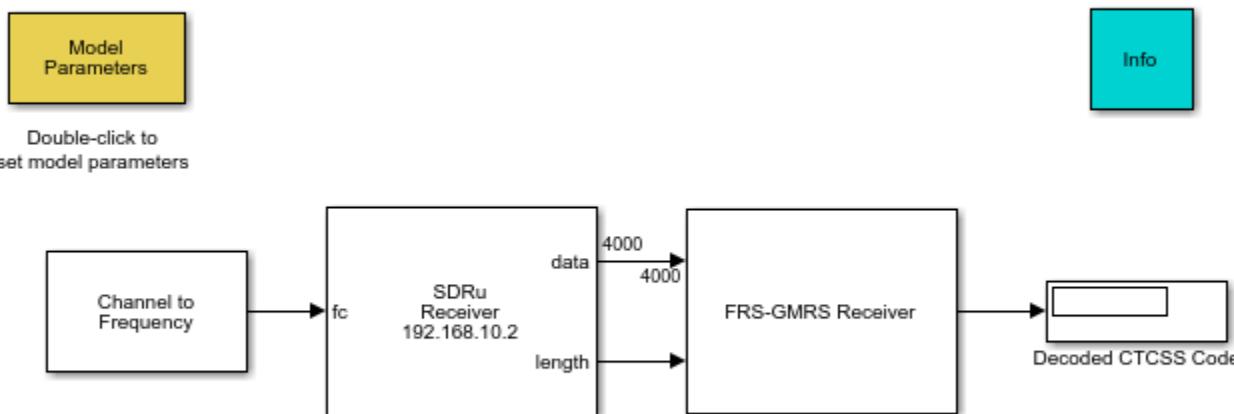
### Overview

Please refer to the “FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware” on page 10-100 example for general information and overview details. Note that all the information in that section applies to this example, except that this example is designed to receive signals instead of transmit them.

### Structure of the Example

This is the top-level block diagram of the model:

#### FRS/GMRS Receiver Using USRP(R) Hardware



Copyright 2011-2020 The MathWorks, Inc.

### Running the Example

Turn on your walkie-talkie, set the channel to be one of the 14 channels (numbered 1 to 14) and the private code to be either one of the 38 private codes (numbered 1 to 38) or 0, in which case no

squelch system is used and all received messages are accepted. Note that the private codes above 38 are digital codes and are not implemented in this example.

Set the channel and private code in the **Model Parameters** GUI in the model so that they match the settings on the walkie-talkie. Run the model, speak into the walkie-talkie, and see if you can hear your voice come out of the computer speakers. If not, try adjusting the "Average signal threshold for squelch" parameter downward slightly. You can change the channel and private code without stopping and restarting the model.

If you hear some dropouts or delay in the sound, run the model in Accelerator mode. From the model menu, select Simulation->Accelerator, then click the run button. If you still experience dropouts or delay in Accelerator mode, try running the model in Rapid Accelerator mode.

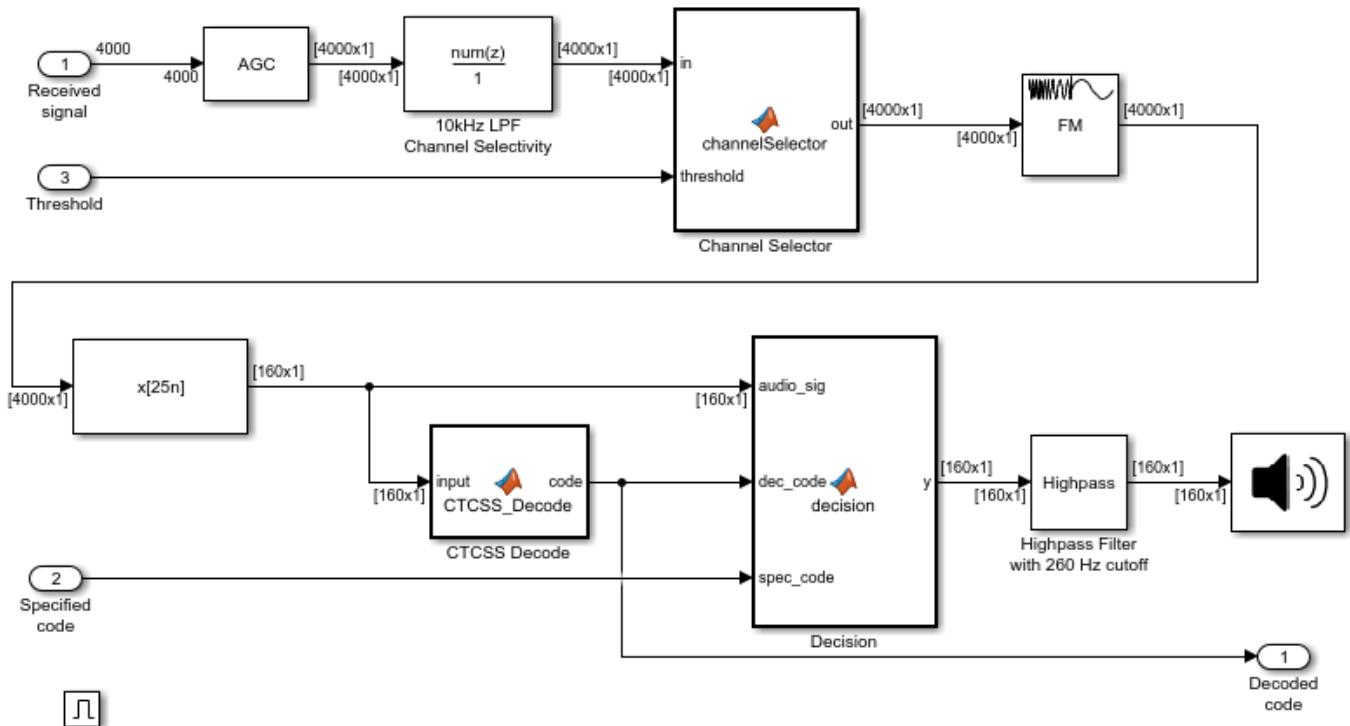
You can also run this model alongside an additional USRP® device running the FRS/GMRS transmitter example, instead of with a physical walkie-talkie.

### FRS/GMRS Receiver Subsystem

The **SDR<sub>u</sub> Receiver** block converts the RF waveform to complex baseband samples, which become the input to this subsystem.

This subsystem is an enabled subsystem, which means that it is only active when the driving 'length' output is greater than 0.

Below is the block diagram of the FRS - GMRS Receiver subsystem:



### Automatic Gain Control

The **Automatic Gain Control** block is the first block that processes the received signal. It processes the signal to ensure that the average magnitude of the samples is about 1. In this case, the

walkie-talkie transmitter is likely nearby the USRP® radio, which implies that the received signal should not suffer from fading, and the received noise should be low. However, in general this may not be the case.

### Channel Selectivity and FM Demodulation

The channel selectivity filter is the next block. If the incoming signal is from an adjacent channel, a low pass channel separation filter will reduce its power significantly. The gap between adjacent channels is 25 kHz, which means the baseband bandwidth is at most 12.5 kHz. Thus, we choose the cutoff frequency to be 10 kHz.

Next, a channel selector computes the average power of the filtered signal, and if it is greater than a threshold (set to a default of 10%), the channel selector determines that the received signal is from the correct channel and it allows the signal to pass through. In the case of an out-of-band signal, although the channel separation filter reduces its magnitude, it is still FM modulated and the modulating signal will be present after FM demodulation. To reject such a signal completely, the channel selector outputs zero.

The **FM Demodulator Baseband** block, with a 2.5 kHz frequency deviation, processes the channel selector output.

After FM demodulation, the **FIR Decimation** block converts the sampling rate to  $200 \text{ kHz} / 25 = 8 \text{ kHz}$ . This is one of the native sample rates of the audio device on your host computer.

### Continuous Tone-Coded Squelch System (CTCSS) and Decision Block

The CTCSS [3] decoder computes the power at each CTCSS tone frequency using the Goertzel algorithm [4] and outputs the code with the largest power into the **Decision** block. The Goertzel algorithm is used because it provides an efficient method to compute the frequency components at predetermined frequencies (namely, the tone code frequencies used by FRS/GMRS).

The **Decision** block compares the decoded code with a preselected code and sends the signal to the audio device if the two codes match. When the preselected code is zero, it indicates no squelch system is used and the decision block passes the signal at the channel to the audio device no matter which code is used.

### Audio Output

Before the audio device, a high pass filter with a cutoff frequency of 260 Hz is used to filter out the CTCSS tones (which have a maximum frequency of 250 Hz) so that they are not heard.

The **Audio Device Writer** block is set up by default to output to the current audio device in your system preferences.

### Exploring the Example

The performance of the model may vary depending on the board that you use. If your model does not produce any sound, you can vary the **Gain (dB)** parameter in the mask of the SDRu Receiver block.

The CTCSS decoding computes the DTFT (Discrete-Time Fourier Transform) of the incoming signal using the Goertzel algorithm and computes the power at the tone frequencies. Since the tone frequencies are very close to each other (only 3-4 Hz apart) the block length of the DTFT should be large enough to provide enough resolution for the frequency analysis. However, long block lengths cause decoding delay; for example, a block length of 16000 will cause 2 seconds of delay, since the CTCSS decoder operates at an 8 kHz sampling rate. This creates a tradeoff between detection

performance and processing latency. The optimal block length may depend on the quality of the transmitter and receiver, the distance between the transmitter and receiver, and other factors. You are encouraged to change the block length in the initialization function by navigating to the `getParamsSdruFRSGMRSRxDemo` function and changing the value of the `FRSRxParams.DecodeBlockSize` field. This will enable you to observe the tradeoff and find the optimal value for your transmitter/receiver pair.

## References

- Family Radio Service on Wikipedia
- General Mobile Radio Service on Wikipedia
- Continuous Tone-Coded Squelch System on Wikipedia
- Goertzel Algorithm on Wikipedia

## Copyright Notice

USRP® is a trademark of National Instruments Corp.

## FRS/GMRS Walkie-Talkie Transmitter with USRP® Hardware

This model shows how to use the Universal Software Radio Peripheral® (USRP®) device with Simulink® to implement a walkie-talkie transmitter. The specific radio standard that this example follows is FRS/GMRS (Family Radio Service / General Mobile Radio Service) with CTCSS (Continuous Tone-Coded Squelch System). You can listen to the transmitted signal using a commercial walkie-talkie device.

In order to run this model, you need a USRP® board with an appropriate transmitter daughterboard that supports the UHF 462-467 MHz band (for example, WBX). Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter block.

This example is designed to work with USA standards for FRS/GMRS operation. The technical specifications for these standards can be found at [1] and [2]. Operation in other countries may or may not work.

### Overview

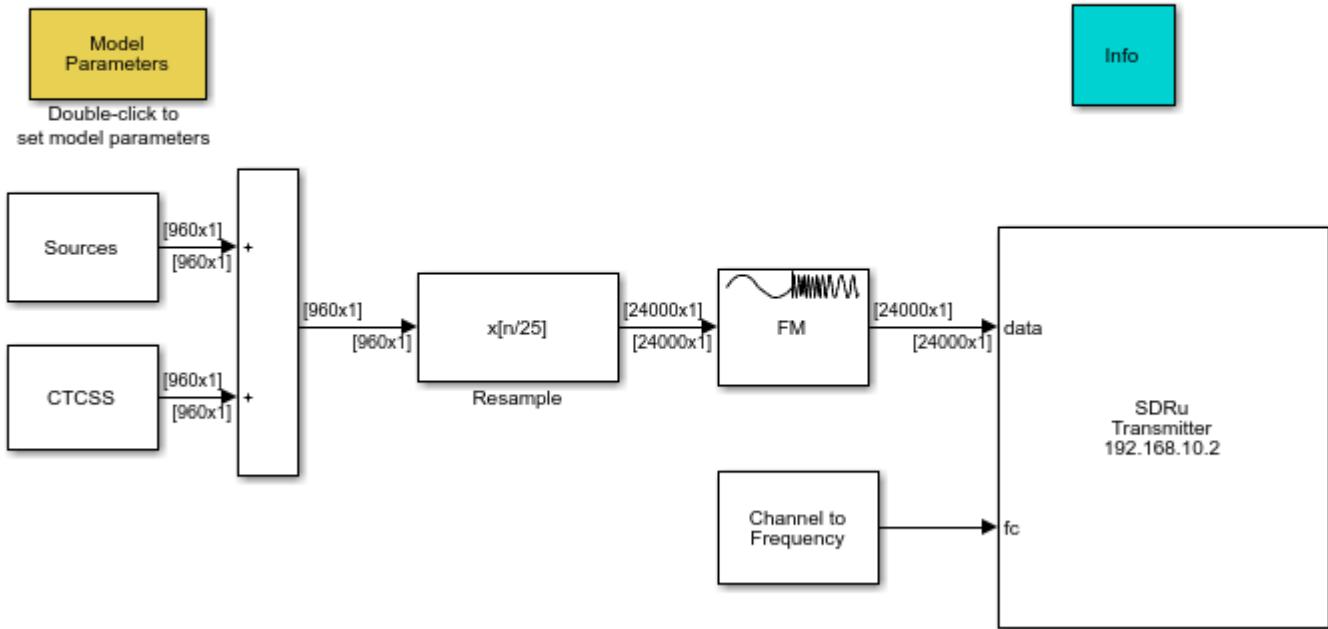
Walkie-talkies provide a subscription-free method of communicating over short distances. Although their popularity has been diminished by the rise of cell phones, they are still useful when lack of reception or high per-minute charges hinders the use of cell phones.

Modern walkie-talkies operate on the FRS/GMRS standards. Both standards use frequency modulation (FM) at 462 or 467 MHz, which is in the UHF (Ultra High Frequency) band. The USRP® device in this example will transmit messages at either 462 or 467 MHz, in a manner that is compatible with FRS/GMRS devices.

### Structure of the Example

This is the top-level block diagram of the model:

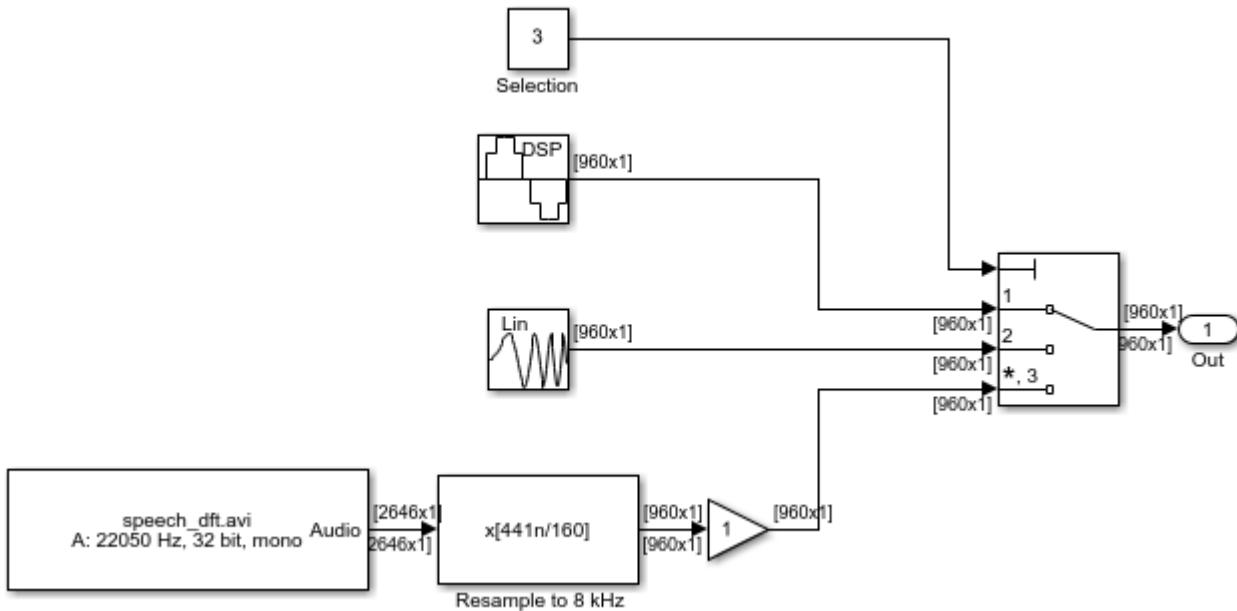
## FRS/GMRS Transmitter Using USRP(R) Hardware



Copyright 2011-2020 The MathWorks, Inc.

### Source Signals

Below is the subsystem of sources for the model:



The source signal can be a pure tone sine wave, a chirp signal, or a multimedia file. To switch between these sources, double-click the **Model Parameters** block to bring up a GUI. This GUI also allows you to set the pure tone frequency or the chirp signal target/sweep time (which controls the duration of the chirp signal). This example works properly with tones as low as 500 Hz and as high as 4 kHz.

When using a multimedia file, a resampler converts the sample rate from 22050 Hz to 8 kHz. This provides a convenient intermediate sample rate from which to resample to the final 200 kHz rate, consistent with the sample rate of the USRP® hardware in this example.

### **Continuous Tone-Coded Squelch System (CTCSS)**

Walkie-talkies operate on a shared public channel, allowing multiple users to access the same channel simultaneously. The CTCSS [3] method filters out undesired communication or interference from these other users by generating a tone between 67 Hz and 250 Hz and transmitting it along with the source signal. The receiver contains logic to detect this tone, and acknowledges a message if the detected tone matches the code setting on the receiver. The receiver filters the tone out, so that the user will not hear the tone.

The CTCSS tone generator generates a continuous phase sine wave with frequency corresponding to the selected private code. The amplitude of the tone is usually 10%-15% of the maximum amplitude of the modulating signal. Note that since the maximum amplitude of all the source signals is 1, the default amplitude of 0.1 for the CTCSS tone corresponds to 10% of the modulating signal's maximum amplitude.

### **Resampler and FM Modulator**

An FIR Interpolator block converts the sampling rate of the sum of the modulating signal and CTCSS tone to match the USRP® device's sampling rate of  $8 \text{ kHz} * 25 = 200 \text{ kHz}$ . The resampling filter is designed using the **FIRInterpolator** System object (TM) from the DSP System Toolbox™.

Below is the block diagram of the **FM Baseband Modulator** subsystem:

This example uses the **FM Modulator** Baseband System object whose maximum frequency deviations and sample rate are set to 2.5 kHz and 200 kHz, respectively.

### **Running the Example**

Turn on your walkie-talkie, set the channel to be one of the 14 channels (numbered 1 to 14) and the private code to be either one of the 38 private codes (numbered 1 to 38) or 0, in which case no squelch system is used and all received messages are accepted. Note that the private codes above 38 are digital codes and are not implemented in this example.

Set the channel and private code in the **Model Parameters** GUI so that they match the walkie-talkie. Run the model and see if you can hear the voice or tone output from the walkie-talkie. If you cannot, try adjusting the amplitude parameter slightly upward. You can change the channel and private code, as well as toggle between the pure tone, the chirp signal, and the multimedia file without stopping and restarting the model.

If you hear some dropouts or delay in the sound, run the model in Accelerator mode. From the model menu, select **Simulation->Accelerator**, then click the run button. If you still experience dropouts or delay in Accelerator mode, try running the model in Rapid Accelerator mode.

You can also run this model alongside an additional USRP® device running the FRS/GMRS receiver example, instead of with a commercial walkie-talkie.

## Exploring the Example

Due to hardware variations among FRS radios and the USRP® boards, you may not hear continuous transmission on your FRS radio. In that case, you can vary the gain of the source signals in the Sources block and examine the resulting behavior.

Part 95.637 (Modulation standards) of the FCC wireless standards [4] states that the maximum frequency deviation is 2.5 kHz for FRS and 5 kHz for GMRS. In practice, it is usually set to 2.5 kHz for both systems. If the maximum signal amplitude increases, the frequency deviation parameter should decrease. Otherwise, the receiving walkie-talkie will not decode the CTCSS code correctly. You can try to use a different signal with different values for frequency deviation to see if your walkie-talkie works properly. If the frequency deviation value is too large, you may not hear anything from your receiver when using a non-zero CTCSS private code. Set the CTCSS code to 0, which disables the squelch system. If you hear the transmitted signal, the CTCSS decoding of the non-zero code is incorrect.

You can reduce the amplitude of the CTCSS tone to determine the minimum amplitude required for your receiver to work correctly.

## References

- Family Radio Service on Wikipedia
- General Mobile Radio Service on Wikipedia
- Continuous Tone-Coded Squelch System on Wikipedia
- Part 95.637 (Modulation standards) of the FCC wireless standards

## Copyright Notice

USRP® is a trademark of National Instruments Corp.

## QPSK Receiver with USRP® Hardware

This model shows how to use the Universal Software Radio Peripheral® (USRP®) device with Simulink® to implement a QPSK receiver. The receiver addresses practical issues in wireless communications, e.g. carrier frequency and phase offset, timing offset and frame synchronization. This model receives the signal sent by the QPSK Transmitter with USRP® Hardware model. The receiver demodulates the received symbols and outputs a simple message to the MATLAB® command line.

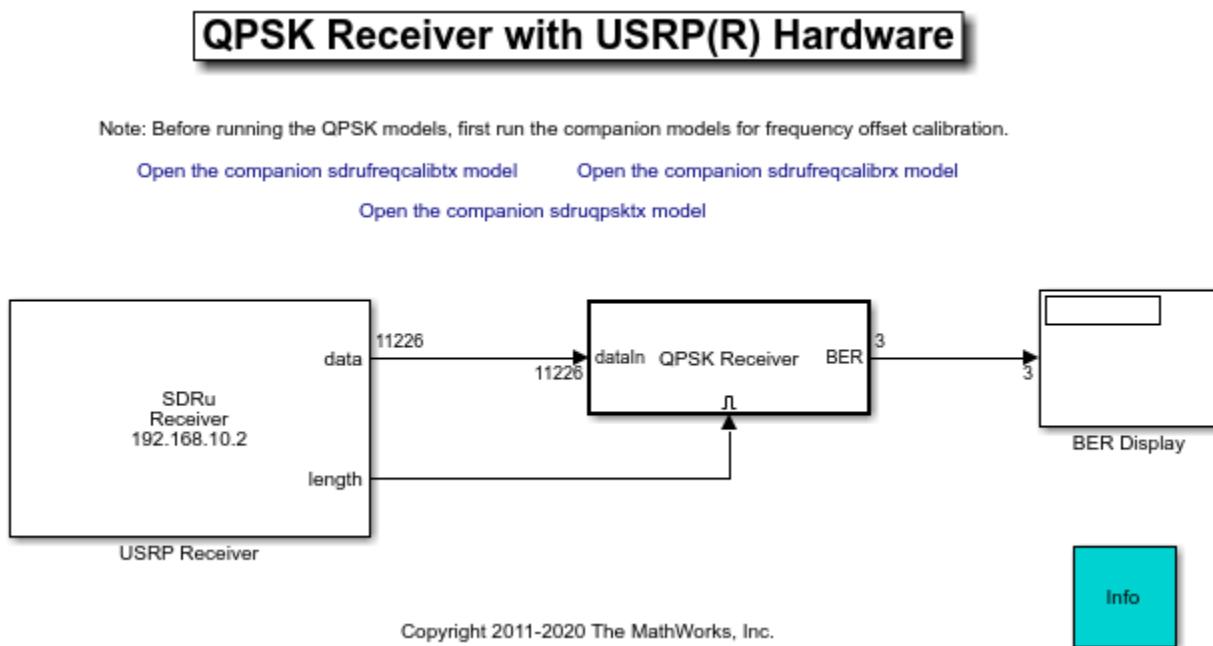
In order to run this model, you need to ensure that the specified center frequency of the USRP Receiver is within the acceptable range of your USRP® daughter board. Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the USRP Receiver block.

### Overview

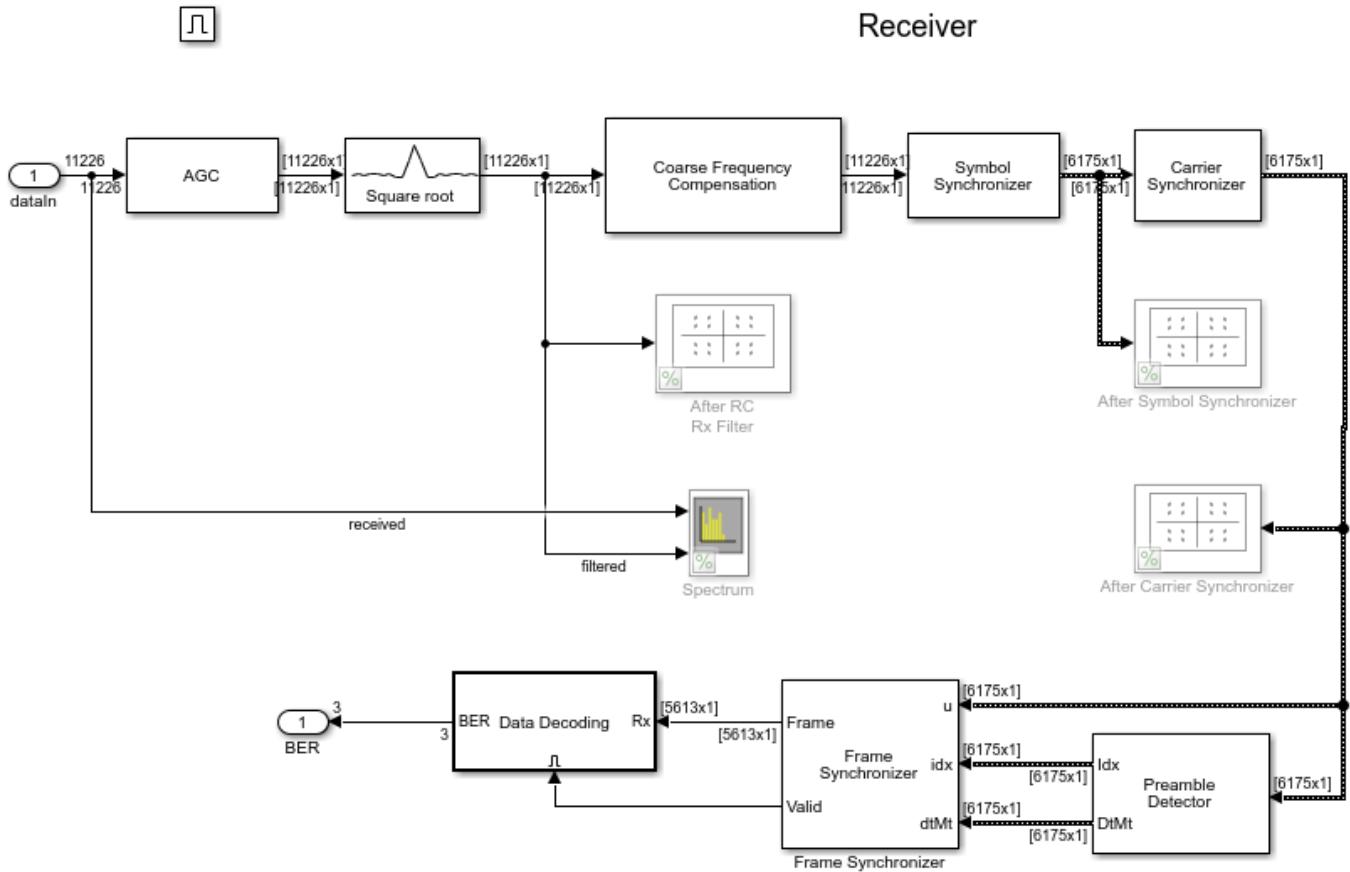
This model performs all processing at complex baseband to handle a time-varying frequency offset, a time-varying symbol delay, and Gaussian noise. To cope with the above-mentioned impairments, this example provides a reference design of a practical digital receiver, which includes correlation-based coarse frequency compensation, symbol timing recovery with fixed-rate resampling and bit stuffing/ skipping, fine frequency compensation, frame synchronization and phase ambiguity resolution. The example uses some key algorithms in MATLAB, emphasizing textual algorithm expression over graphical algorithm expression.

### Structure of the Example

The top-level structure of the model is shown in the following figure. Which includes a USRP receiver block, a QPSK Receiver subsystem and a BER Display blocks.



The detailed structures of the **QPSK Receiver** subsystem are illustrated in the following figure.



The components are further described in the following sections.

- **AGC** - Automatic gain control
- **Raised Cosine Receive Filter** - Uses a rolloff factor of 0.5
- **Coarse Frequency Compensation** - Estimates an approximate frequency offset of the received signal and corrects it
- **Symbol Synchronizer** - Resamples the input signal according to a recovered timing strobe so that symbol decisions are made at the optimum sampling instants
- **Carrier Synchronizer** - Compensates for the residual frequency offset and the phase offset
- **Preamble Detector** - Detect location of the frame header
- **Frame Synchronizer** - Aligns the frame boundaries at the known frame header
- **Data Decoding** - Resolves the phase ambiguity caused by the **Carrier Synchronizer**, demodulates the signal, and decodes the text message

## Receiver

### AGC

The received signal amplitude affects the accuracy of the carrier and symbol synchronizer. Therefore the signal amplitude should be stabilized to ensure an optimum loop design. The AGC output power is set to a value ensuring that the equivalent gains of the phase and timing error detectors keep

constant over time. The AGC is placed before the **Raised Cosine Receive Filter** so that the signal amplitude can be measured with an oversampling factor of two, thus improving the accuracy of the estimate. You can refer to Chapter 7.2.2 and Chapter 8.4.1 of [ 1 ] for details on how to design the phase detector gain.

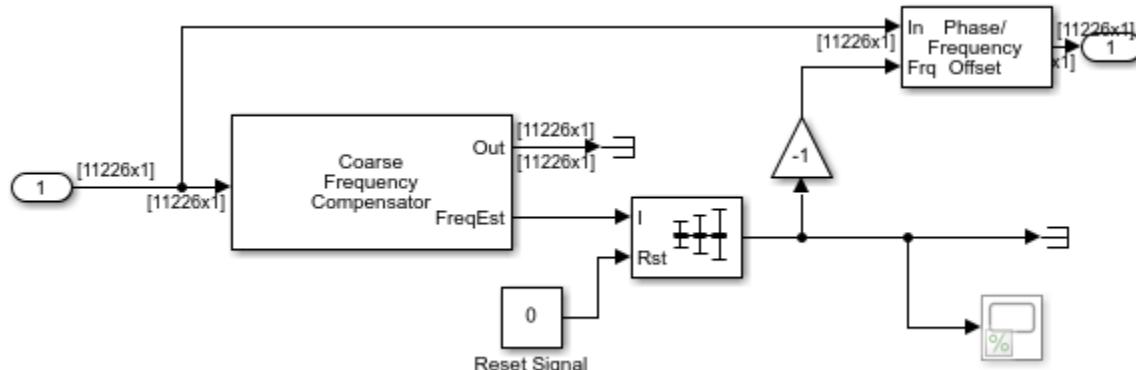
### Raised Cosine Receive Filter

The **Raised Cosine Receive Filter** provides matched filtering for the transmitted waveform with a rolloff factor of 0.5.

### Coarse Frequency Compensation

The **Coarse Frequency Compensation** subsystem corrects the input signal with a rough estimate of the frequency offset. The following diagram shows the subsystem, in which the frequency offset is estimated by averaging the output of the correlation-based algorithm of the **Coarse Frequency Compensator** block. The compensation is performed by the Phase/Frequency Offset block. There is usually a residual frequency offset even after the coarse frequency compensation, which would cause a slow rotation of the constellation. The **Carrier Synchronizer** block compensates for this residual frequency.

The accuracy of the **Coarse Frequency Compensator** decreases with its maximum frequency offset value. Ideally, this value should be set just above the expected frequency offset range.



### Symbol Synchronizer

The timing recovery is performed by a **Symbol Synchronizer** library block, which implements a PLL, described in Chapter 8 of [ 1 ], to correct the timing error in the received signal. The timing error detector is estimated using the Gardner algorithm, which is rotationally invariant. In other words, this algorithm can be used before or after frequency offset compensation. The input to the block is oversampled by two. On average, the block generates one output symbol for every two input samples. However, when the channel timing error (delay) reaches symbol boundaries, there will be one extra or missing symbol in the output frame. In that case, the block implements bit stuffing/skipping and generates one more or less samples comparing to the desired frame size. So the output of this block is a variable-size signal.

The *Damping factor*, *Normalized loop bandwidth*, and *Detector gain* parameters of the block are tunable. Their default values are set to 1 (critical damping), 0.01 and 5.4 respectively, so that the PLL quickly locks to the correct timing while introducing little timing jitter.

## Carrier Synchronizer

The fine frequency compensation is performed by a **Carrier Synchronizer** library block, which implements a phase-locked loop (PLL), described in Chapter 7 of [ 1 ], to track the residual frequency offset and the phase offset in the input signal. The PLL uses a Direct Digital Synthesizer (DDS) to generate the compensating phase that offsets the residual frequency and phase offsets. The phase offset estimate from DDS is the integral of the phase error output of a Loop Filter.

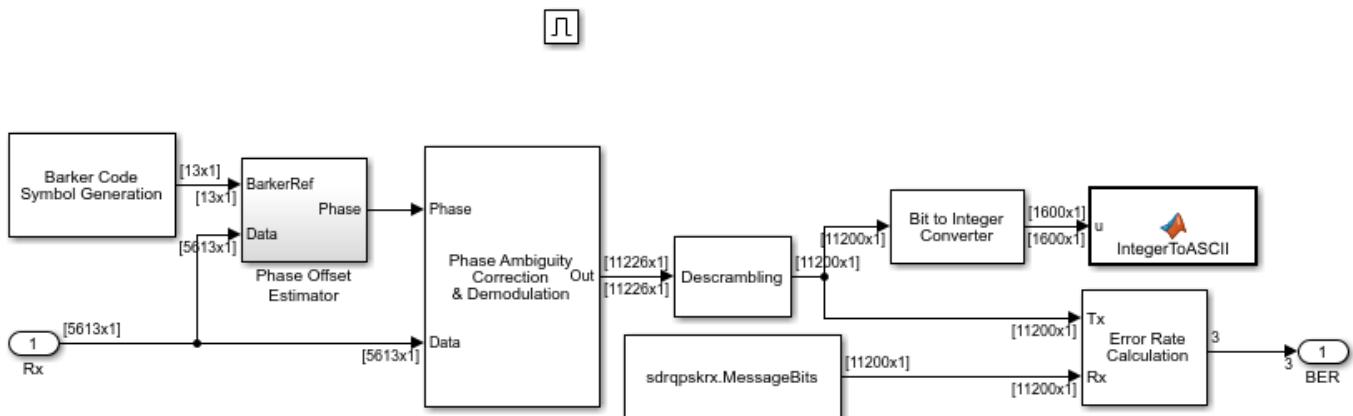
The *Damping factor* and *Normalized loop bandwidth* parameters of the block are tunable. Their default values are set to 1 (critical damping) and 0.01 respectively, so that the PLL quickly locks to the intended phase while introducing little phase noise.

## Preamble Detector and Frame Synchronizer

The location of the known frame header is detected by a **Preamble Detector** library block and the frame synchronization is performed by a MATLAB System block using a **FrameSynchronizer** System object. The Preamble Detector block uses the known frame header (QPSK-modulated Barker code) to correlate against the received QPSK symbols in order to find the location of the frame header. The Frame Synchronizer block uses this location information to align the frame boundaries. It also transforms the variable-size output of the **Symbol Synchronizer** block into a fixed-size frame, which is necessary for the downstream processing. The second output of the block is a boolean scalar indicating if the first output is a valid frame with the desired header and if so, enables the **Data Decoding** subsystem to run.

## Data Decoding

The **Data Decoding** enabled subsystem performs phase ambiguity resolution, demodulation and text message decoding. The **Carrier Synchronizer** block may lock to the unmodulated carrier with a phase shift of 0, 90, 180, or 270 degrees, which can cause a phase ambiguity. For details of phase ambiguity and its resolution, please refer to Chapter 7.2.2 and 7.7 in [ 1 ]. The **Phase Offset Estimator** subsystem determines this phase shift. The **Phase Ambiguity Correction & Demodulation** subsystem rotates the input signal by the estimated phase offset and demodulates the corrected data. The payload bits are descrambled, and then decoded. All of the stored bits are converted to characters are printed in the Simulink Diagnostic Viewer.



## Running the Example

Before running this model, first start the QPSK Transmitter with USRP® Hardware model.

This receiver model is capable of handling a frequency offset of 12.5kHz between the transmitter and receiver boards. However, when the frequency offset exceeds this range, the **Coarse Frequency Compensation** subsystem cannot accurately determine the offset of the received signal, which is critical for correct timing recovery and data decoding. We encourage you to run the companion frequency calibration transmitter and receiver models with your USRP® transmitter and receiver hardware to roughly determine the frequency offset between your two USRP® boards. With that frequency offset value, you can manually adjust the *Center frequency* of the **USRP Receiver** subsystem in the receiver model to ensure a residual frequency offset that the model can track.

If the received signal is too weak or too strong, you might notice some garbled message output. In that case, you can change the gain of either the **USRP Transmitter** subsystem in the QPSK Transmitter with USRP® Hardware model or the **USRP Receiver** subsystem in the current model for better reception. Please also change the preamble detector threshold, in case you may see some recurrent garbled message output. This is because when the threshold of the preamble detector is set too low, the following steps will try to decode the header. When the threshold is set too high, you may not get any outputs.

To run this model, first turn on the USRP® hardware and connect it to the computer. Set the *Center frequency* parameter of the **USRP Receiver** block according to the center frequency setting of the QPSK Transmitter with USRP® Hardware model and the frequency calibration result. Then run the model. To ensure real-time processing, the model is by default set to run in Accelerator mode, and to remove all signal visualization. The received messages are decoded and printed out in the **View diagnostics** window while the simulation is running.

### Exploring the Example

The example allows you to experiment with multiple system capabilities to examine their effect on bit error rate performance.

You can tune the *Normalized loop bandwidth* and *Damping factor* parameters of the **Symbol Synchronizer** and **Carrier Synchronizer** blocks, to assess their convergence time and estimation accuracy. In addition, you can assess the pull-in range of the **Carrier Synchronizer** block. With a large *Normalized loop bandwidth* and *Damping factor*, the PLL can acquire over a greater frequency offset range. However a large *Normalized loop bandwidth* allows more noise, which leads to a large mean squared error in the phase estimation. "Underdamped systems (with Damping Factor less than one) have a fast settling time, but exhibit overshoot and oscillation; overdamped systems (with Damping Factor greater than one) have a slow settling time but no oscillations." [ 1 ]. For more detail on the design of these PLL parameters, you can refer to Appendix C in [ 1 ].

You can also tune the Preamble detector threshold and see its effects on the output message. If your model does not run in real-time and if you have a supported compiler installed on your computer, set the Simulink Simulation Mode to "accelerator" or "Rapid Accelerator" to improve model execution speed and enable real-time operation. If your model is already running in real-time, you can also achieve higher symbol rates. If you change the symbol rate of the receiver, also set the symbol rate of the transmitter model.

### References

1. Michael Rice, "Digital Communications - A Discrete-Time Approach", Prentice Hall, April 2008.

### Copyright Notice

USRP® is a trademark of National Instruments Corp.

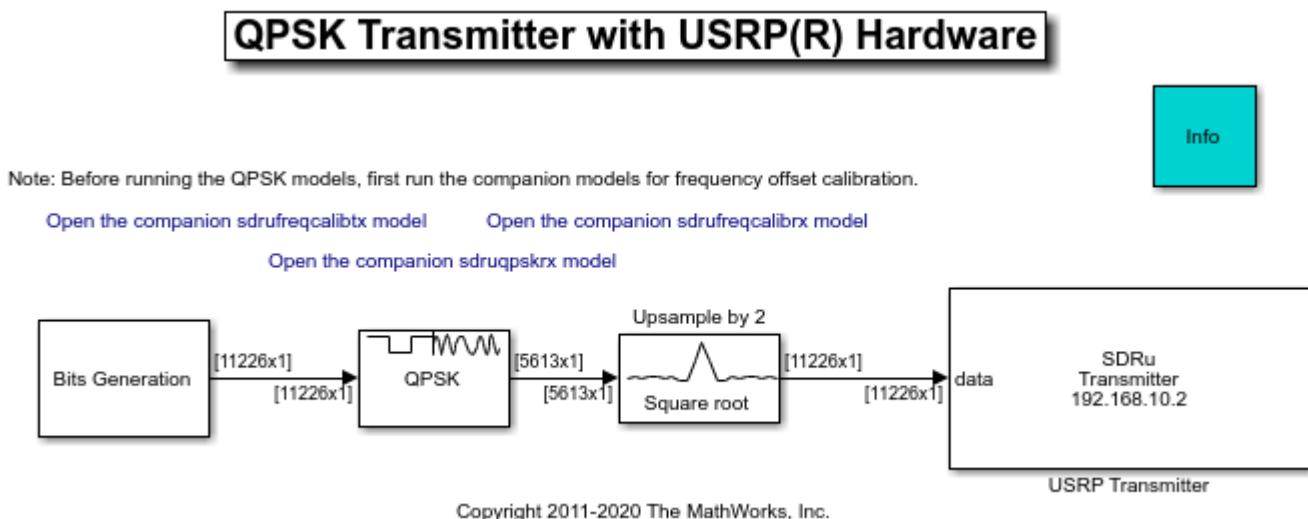
## QPSK Transmitter with USRP® Hardware

This model shows how to use the Universal Software Radio Peripheral® (USRP®) device with Simulink® to implement a QPSK transmitter. The USRP® device in this model will keep transmitting indexed 'Hello world' messages at its specified center frequency. You can demodulate the transmitted message using the QPSK Receiver with USRP® Hardware model with an additional USRP® device.

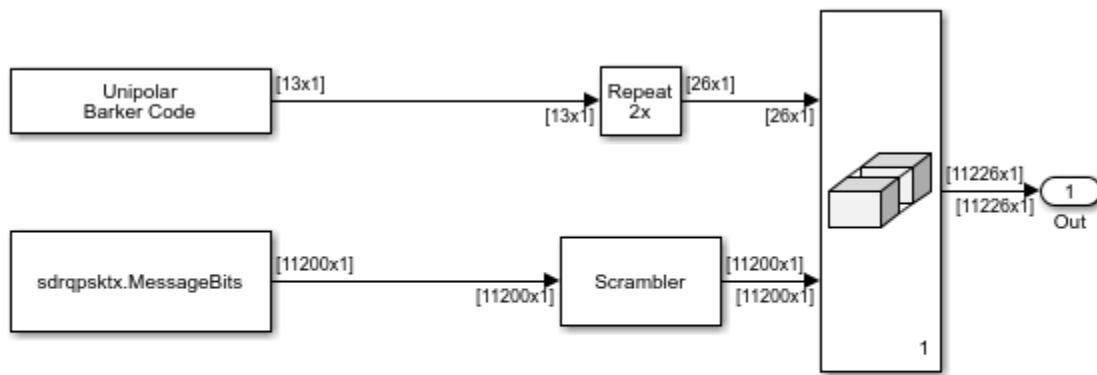
In order to run this model, you need to ensure that the specified center frequency of the SDRu Transmitter is within the acceptable range of your USRP® daughterboard. Please refer to the Setup and Configuration section of Documentation for USRP® Radio for details on configuring your host computer to work with the SDRu Transmitter block.

### Structure of the Example

The top-level structure of the model is shown in the following figure:



The transmitter includes the **Bit Generation** subsystem, the **QPSK Modulator** block, and the **Raised Cosine Transmit Filter** block. The **Bit Generation** subsystem uses a MATLAB workspace variable as the payload of a frame, as shown in the figure below. Each frame contains 100 'Hello world ####' messages and a header. The first 26 bits are header bits, a 13-bit Barker code that has been oversampled by two. The Barker code is oversampled by two in order to generate precisely 13 QPSK symbols for later use in the **Data Decoding** subsystem of the receiver model. The remaining bits are the payload. The payload correspond to the ASCII representation of 'Hello world ####', where '####' is a repeating sequence of '000', '001', '002', ..., '099'. The payload is scrambled to guarantee a balanced distribution of zeros and ones for the timing recovery operation in the receiver model. The scrambled bits are modulated by the **QPSK Modulator** (with Gray mapping). The modulated symbols are upsampled by two by the **Raised Cosine Transmit Filter** with a roll-off factor 0.5. The output rate of the **Raised Cosine Filter** is set to be 400k samples/second with a symbol rate of 200k symbols per second. Please match the symbol rate of the transmitter model and the receiver model correspondingly.



Each data frame contains 26 bits header (For Sync Purpose) and 100 "Hello world ####" message. Scrambler is there to improve data transition density and frequency offset estimation.

## Running the Example

Before running the model, first turn on the USRP® and connect it to the computer. Set the *Center frequency* parameter of the **SDRu Transmitter** block and run the model. You can run the QPSK Receiver with USRP® Hardware model with an additional USRP® device to receive the transmitted signal.

## Exploring the Example

Due to hardware variations among the USRP® boards, a frequency offset will likely exist between the USRP® transmitter hardware and the USRP® receiver hardware. In that case, perform a manual frequency calibration using the companion frequency offset calibration transmitter and receiver models and examine the resulting behavior.

Since the gain behavior of different USRP® daughter boards also varies considerably, the default gain setting in the transmitter and receiver models may not be well-suited for your daughter boards. If the message is not properly decoded by the receiver model, you can vary the gain of the source signals in the **SDRu Transmitter** block of this model, and that of the **SDRu Receiver** block in the receiver model.

## Copyright Notice

USRP® is a trademark of National Instruments Corp.