Measuring Congestion of Wifi Channels with the USRP B206mini-i

by P. Trujillo

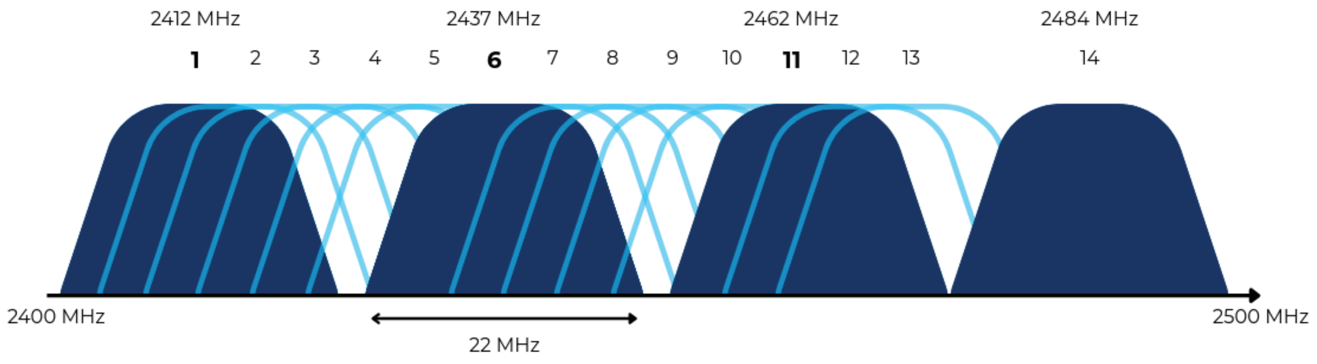
DIGILENT®
Part of the NI Product Family

Legal & Ethical Notice: Capturing 802.11 frames may be restricted by law. Only sniff traffic you are authorized to monitor, respect privacy, and comply with local regulations and license requirements. This guide focuses on **test environments** and **lab setups.**

Introduction

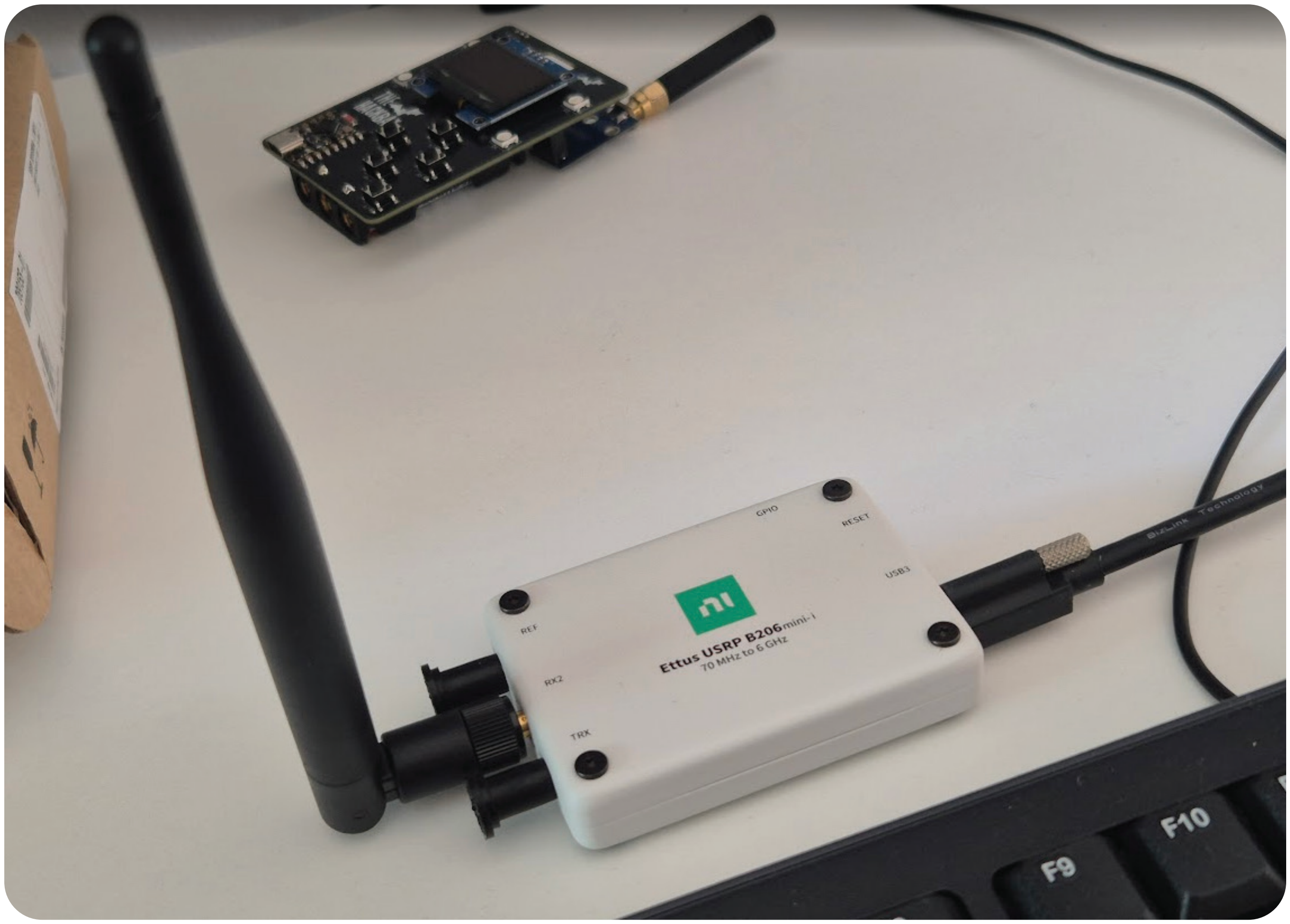
I am pretty sure you are reading this over Wi-Fi. If you are not a telecom or RF specialist, it is easy to imagine that Wi-Fi—more precisely, the IEEE 802.11 standard—lets every device talk over a single 2.4 GHz "pipe." That's not quite right.

When we browse the web on 2.4 GHz, we are actually using one channel within that band—and so is your neighbor. Think of channels as lanes on a highway: if everyone piles into the same lane, traffic slows down. That is a congested channel. To make things trickier, 2.4 GHz channels are spaced 5 MHz apart while typical transmissions occupy ~20 MHz, so the lanes overlap. Even if you pick a different lane, you can still feel your neighbor's traffic bleeding into yours.



Understanding which channels are busiest—and by how much—helps you choose better settings and improve your network's performance. In this article I will show a simple, practical way to measure channel occupancy: sweep the 2.4 GHz band, dwell briefly on each channel, and compute the percentage of time the signal sits above the noise floor. It doesn't decode packets or identify specific transmitters, but it gives a clear, actionable picture of channel load you can use for planning and troubleshooting.

To do this I am going to use the USRP B206mini-i, a portable SDR device able of working with frequencies from 70 MHz to 6 GHz, enough for 2.4 GHz and algo 5 GHz wifi interfaces.



Installing UHD and enabling Python API

In order to work with the USRP B206mini-i, we need to use the latest verion of the UHD drivers. To install this version, we need to navigate to the Ettus Research GitHUb repository, and clone the `/usd` repository.

```
pablo@friday:~$ git clone git@github.com:EttusResearch/uhd.git
```

Then we will need to point to the version 4.9.0.1 or a newer one.

```
pablo@friday:~$ cd uhd
pablo@friday:~/uhd$ git v4.9.0.1
```

Then, we can install the drivers using `cmake`. To add the Python drivers for the devices, we have to add the `-DENABLE_PYTHON_API=ON`.

```
pablo@friday:~/uhd$ cd host
pablo@friday:~/uhd/host$ mkdir build
pablo@friday:~/uhd/host$ cd build
pablo@friday:~/uhd/host/build$ cmake -DENABLE_PYTHON_API=ON ../
pablo@friday:~/uhd/host/build$ make
pablo@friday:~/uhd/host/build$ sudo make install
```

Now we need to update and copy the rules file to **`/etc/udev/rules.d`**.

```
pablo@friday:~/usr/local/lib/uhd$ sudo cp uhd-usrp.rules /etc/udev/rules.d/
pablo@friday:~/usr/local/lib/uhd$ sudo udevadm control –reload-rules
pablo@friday:~/usr/local/lib/uhd$ sudo udevadm trigger
```

At this point, we will be able to communicate with the B206mini-i. To check the connection, we can execute **`uhd_find_devices`**.

```
pablo@friday:~$ uhd_find_devices
[INFO] [UHD] linux; GNU C++ version 13.3.0; Boost_108300; UHD_4.9.0.HEAD-0-
g006d7f76
--------------------------------------------------
-- UHD Device 0
--------------------------------------------------
Device Address:
    serial: 34CD2D4
    name: B206i
    product: B206mini
    type: b200
```

Now, we are ready to use the SDR device from Python.

Capturing 2.4 GHz band

In Python, we'll add the **`uhd`** package, and we'll also use **`numpy`** and **`matplotlib`**.

First, we need to connect to the USRP device with **`uhd.usrp.MultiUSRP()`**. This call initializes the radio (including loading the FPGA image) and makes it ready to use.

To capture data, we'll use the convenience function **`recv_num_samps()`**. It takes the radio configuration as arguments and returns the requested number of samples. Its arguments are:

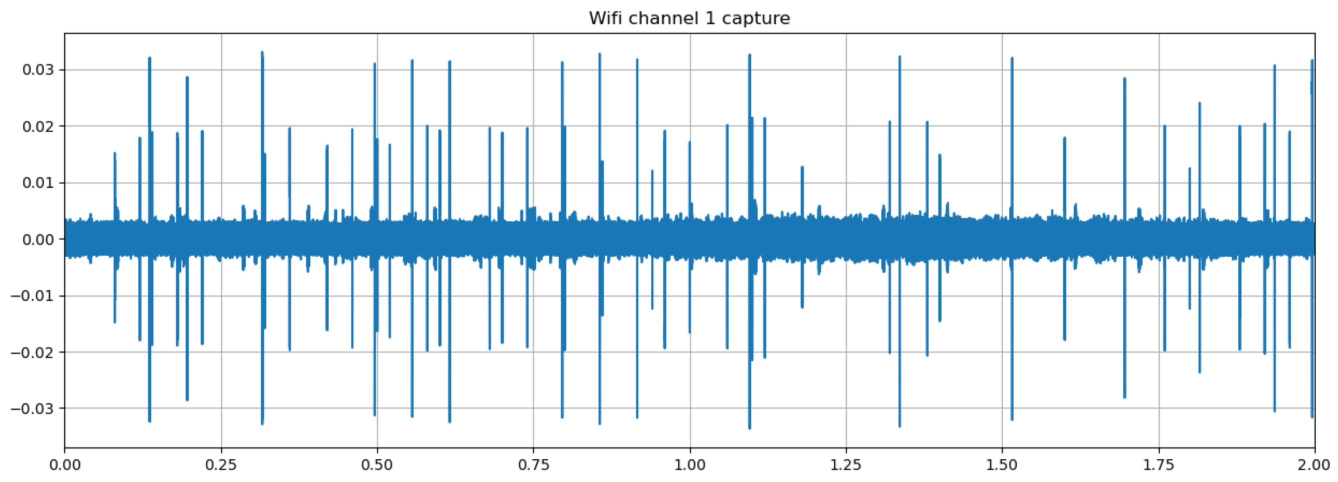
- **`nsamps`**: number of samples to receive
- **`center_frequency`**: RF tuning frequency
- **`sample_rate`**: acquisition/sample rate
- **`radio`**: channel index (e.g., **`0`**)
- **`gain`**: receiver gain in dB

The output is a NumPy array of complex floats (**`complex64`**). If you want to plot components, use **`np.real`** for the real part or **`np.imag`** for the imaginary part.

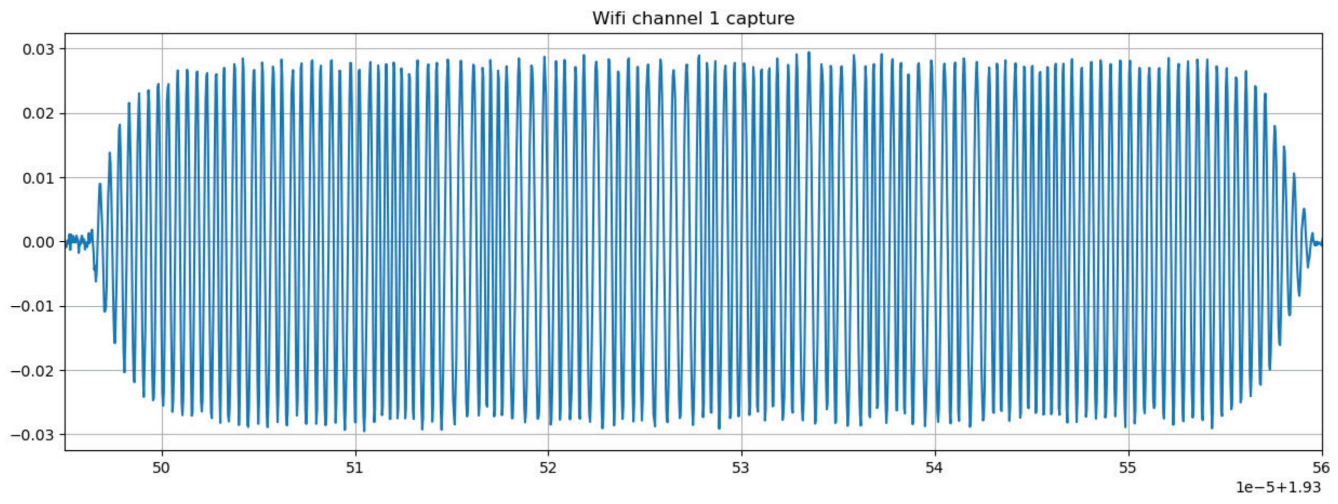
The following script puts this all together.

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

# detect device
device = uhd.usrp.MultiUSRP()

# configure capture
t = 2.0
wifi_channel = 1
fadq = 25e6
fcenter = 2412e6 + ((wifi_channel-1) * 5)
gain = 20
nsamples = int(t*fadq)

tVector = np.linspace(0,t,nsamples)

# Capture data
samples = device.recv_num_samps(nsamples, fcenter, fadq, [0], gain)

plt.figure(figsize=(15, 5))
plt.plot(tVector, np.real(samples[0]), label="I")
plt.grid()
```

The entire capture is shown in the next figure. We can see that several frames have been captured.



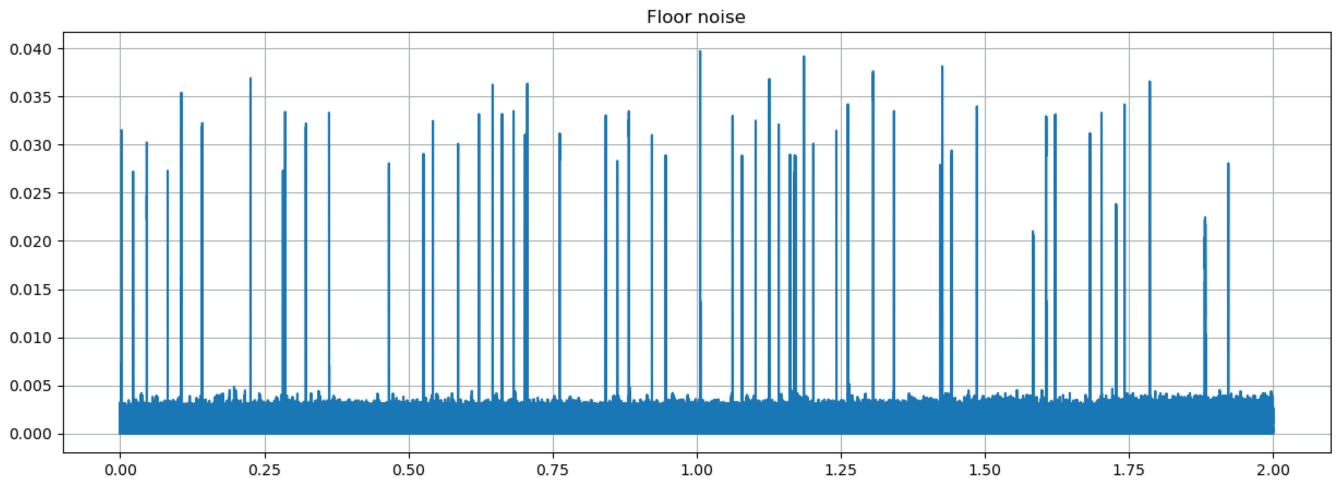
We can zoom into one of the frames to check its aspect.



Computing the noise floor

Now, in order to obtain the utilization or the occupancy of each channel, we need to determine if a frame is received or not. To do this, we need to check the noise floor of the signal.

To obtain this value we are going to plot the absolute value of the data, and verify the amplitude of the noise.



With this data, we can determine that the bus is occupied when the value of the signal received is above 0.05.

Measuring channel occupancy

Now, we can calculate the amount of samples in a configured time in which the signal is above the threshold. This value will be related to the channel utilization, but notice that channels are overlapped, so it also will be related to the occupancy of the adjacent channels.

The next Python function will perform this calculation for a given wifi channel.

```
# Obtain % of time that bus is occupied
def measure_occupancy (data, threshold):
    occupancy = 0 # init value

  for point in data:
    # Verify if samples are above the threshold
    if point > threshold:
      occupancy += 1

  return occupancy/len(data)
```

Then, we can make this for the wifi channels from 1 to 13 by executing the function recursively.

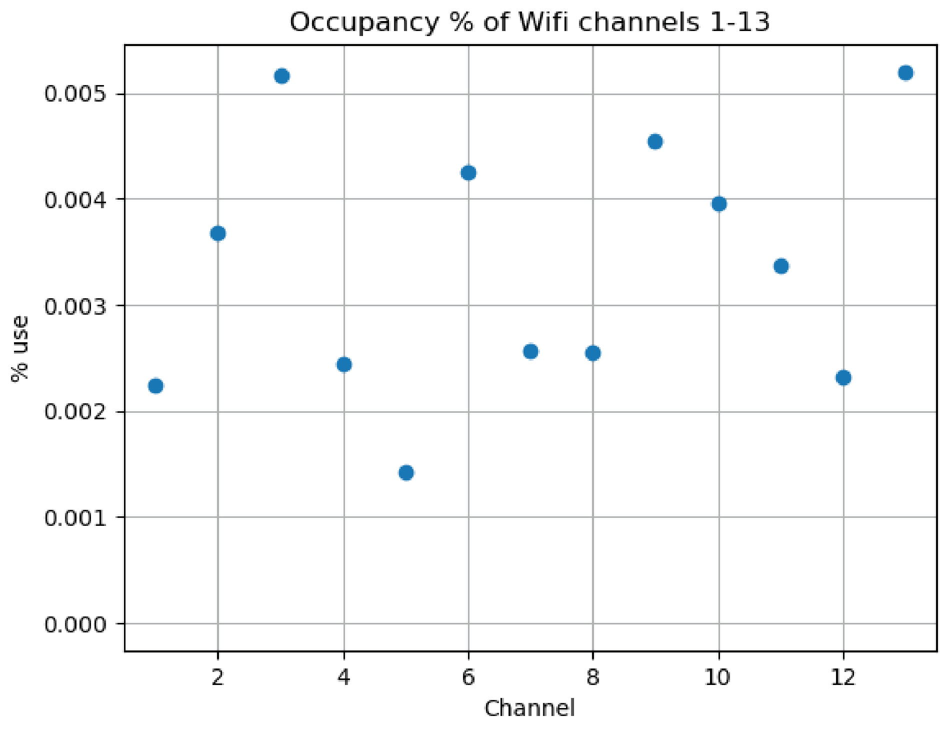
```
occupancy = 14*[0]

# Measure all channels
for channel in range(1,14):
    print(f'Measuring channel {channel}...')
    fcenter = 2412e6 + ((wifi_channel-1) * 5)
    samples = device.recv_num_samps(nsamples, fcenter, fadq, [0], gain)
    print(f'Computing occupancy of channel {channel}...')
    occupancy[channel] = measure_occupancy(np.abs(samples[0]), 0.005)
    print(f'Channel {channel} occupancy: {occupancy[channel]}')
```

The output of this script is as follows.

```
Measuring channel 1...
Computing occupancy of channel 1...
Channel 1 occupancy: 0.00224288
Measuring channel 2...
Computing occupancy of channel 2...
Channel 2 occupancy: 0.0036764
Measuring channel 3...
Computing occupancy of channel 3...
Channel 3 occupancy: 0.00517016
Measuring channel 4...
Computing occupancy of channel 4...
Channel 4 occupancy: 0.00243592
Measuring channel 5...
Computing occupancy of channel 5...
Channel 5 occupancy: 0.0014204
Measuring channel 6...
Computing occupancy of channel 6...
Channel 6 occupancy: 0.00425696
Measuring channel 7...
Computing occupancy of channel 7...
Channel 7 occupancy: 0.00256688
Measuring channel 8...
Computing occupancy of channel 8...
Channel 8 occupancy: 0.00255416
Measuring channel 9...
Computing occupancy of channel 9...
Channel 9 occupancy: 0.00454312
Measuring channel 10...
Computing occupancy of channel 10...
Channel 10 occupancy: 0.00395856
Measuring channel 11...
Computing occupancy of channel 11...
Channel 11 occupancy: 0.00336856
Measuring channel 12...
Computing occupancy of channel 12...
Channel 12 occupancy: 0.00231984
Measuring channel 13...
Computing occupancy of channel 13...
Channel 13 occupancy: 0.00519664
```

Finally, we can represent the occupancy of all the channels by plotting them with the following code.

```
channels = np.linspace(0,13,14)
plt.plot(channels, occupancy, 'o')
plt.grid()
plt.title("Occupancy % of Wifi channels 1-13")
plt.xlim(0.5,13.5)
plt.ylabel("% use")
plt.xlabel("Channel")
```



Conclusions

In this project I swept the 2.4 GHz band with a B206mini-i and measured, per channel, the fraction of time the received power stayed above the noise floor. By converting that dwell-time metric into a percentage, I obtained a compact view of “how busy” each channel really is, independent of protocol decoding.

Because 2.4 GHz channels overlap (5 MHz spacing for ~20 MHz-wide signals), an energy-only metric can’t reliably attribute which specific transmission belongs to each channel - power “spills” into adjacent channels. Even so, it provides a useful estimate of relative channel occupancy and helps identify which channels are consistently more congested. For precise attribution, you’d need preamble/packet detection and more selective filtering.