HackRF

Mar 17, 2022

User Documentation

CHAPTER 1

HackRF One

HackRF One is the current hardware platform for the HackRF project. It is a Software Defined Radio peripheral capable of transmission or reception of radio signals from 1 MHz to 6 GHz. Designed to enable test and development of modern and next generation radio technologies, HackRF One is an open source hardware platform that can be used as a USB peripheral or programmed for stand-alone operation.

1.1 Features

- half-duplex transceiver
- operating freq: 1 MHz to 6 GHz
- supported sample rates: 2 Msps to 20 Msps (quadrature)
- resolution: 8 bits
- interface: High Speed USB (with USB Micro-B connector)
- power supply: USB bus power
- software-controlled antenna port power (max 50 mA at 3.3 V)
- SMA female antenna connector (50 ohms)
- SMA female clock input and output for synchronization
- convenient buttons for programming
- pin headers for expansion
- portable
- open source

CHAPTER 2

Opera Cake

Opera Cake is an antenna switching add-on board for HackRF One. Consisting of two 1x4 switches, Opera Cake also has a cross-over switch that permits operation as a 1x8 switch. Up to eight Opera Cakes may be stacked onto a single HackRF One provided that each Opera Cake is configured with a different board address.

Opera Cake is configured with the `hackrf_operacake` command-line tool.

2.1 Using Opera Cake

2.1.1 Banks

Opera Cake's ports are grouped in two banks (or "sides"), one on each end of the board. Bank A consists of ports A0 through A4 while bank B consists of ports B0 through B4.

2.1.2 Ports

Opera Cake has two primary ports, A0 and B0, each of which can be switched to any of eight secondary ports, A1-A4 and B1-B4. Each primary port is always connected to one secondary port. By default, A0 is connected to A1, and B0 is connected to B1. It is not possible to connect both primary ports to secondary ports in the same bank at the same time.

Port connections may be configured manually. For example, to connect A0 to A2 and B0 to B3:

```
hackrf_operacake -a A2 -b B3
```

To connect A0 to B2 and B0 to A4:

```
hackrf_operacake -a B2 -b A4
```

If only one primary port is configured, the other primary port will be connected to the first secondary port on the opposite side. For example, after the next two commands B0 will be connected to A1:

```
hackrf_operacake -a A2 -b B3
hackrf_operacake -a B2
```

2.1.3 LEDs

Port selections are indicated by LEDs next to each port's connector. Port A0 and the secondary port connected to A0 are indicated with a green LED. Port B0 and the secondary port connected to B0 are indicated with a yellow LED.

2.1.4 Board Address

Each Opera Cake has a numeric address set by optional jumpers installed on header P1. The default address (without jumpers) is 0. The `--list` or `-l` option can be used to list the address(es) of one or more Opera Cakes installed on a HackRF One:

```
hackrf_operacake -l
```

The address may be set to any number from 0 to 7 by installing jumpers across the A0, A1, and/or A2 pins of header P1.

Address	A2 Jumper	A1 Jumper	A0 Jumper
0	No	No	No
1	No	No	Yes
2	No	Yes	No
3	No	Yes	Yes
4	Yes	No	No
5	Yes	No	Yes
6	Yes	Yes	No
7	Yes	Yes	Yes

When configuring an Opera Cake, the address may be specified with the `--address` or `-o` option:

```
hackrf_operacake -o 1 -a A1 -b B2
```

If the address is unspecified, 0 is assumed. It is only necessary to specify the address if the address has been changed with the addition of jumpers, typically required only if multiple Opera Cakes are stacked onto a single HackRF One.

2.2 Modes of Operation

Opera Cake supports three modes of operation: `manual`, `frequency`, and `time`. The operating mode can be set with the `--mode` or `-m` option, and the active operating mode is displayed with the `--list` or `-l` option.

2.2.1 Manual Mode

The default mode of operation is `manual`. In manual mode, fixed port connections are configured with the `-a` and `-b` options as in the port configuration examples above. If the operating mode has been changed, it can be changed back to manual mode with:

```
hackrf_operacake -m manual
```

2.2.2 Frequency Mode

In frequency mode, the A0 port connection switches automatically whenever the HackRF is tuned to a different frequency. This is useful when antennas for different frequency bands are connected to various ports.

The bands are specified in priority order. The final band specified will be used for frequencies not covered by the other bands specified.

To assign frequency bands to ports you must use the `-f <port:min:max>` option for each band, with the minimum and maximum frequencies specified in MHz. For example, to use port A1 for 100 MHz to 600 MHz, A3 for 600 MHz to 1200 MHz, and B2 for 0 MHz to 4 GHz:

```
hackrf_operacake -m frequency -f A1:100:600 -f A3:600:1200 -f B2:0:4000
```

If tuning to precisely 600 MHz, A1 will be used as it is listed first. Tuning to any frequency over 4 GHz will use B2 as it is the last listed and therefore the default port.

Only the A0 port connection is specified in frequency mode. Whenever the A0 connection is switched, the B0 connection is also switched to the secondary port mirroring A0's secondary port. For example, when A0 switches to B2, B0 is switched to A2.

Once configured, an Opera Cake will remain in frequency mode until the mode is reconfigured or until the HackRF One is reset. You can pre-configure the Opera Cake in frequency mode, and the automatic switching will continue to work while using other software.

Although multiple Opera Cakes on a single HackRF One may be set to frequency mode at the same time, they share a single switching plan. This can be useful, for example, for a filter bank consisting of eight filters.

2.2.3 Time Mode

In time mode, the A0 port connection switches automatically over time, counted in units of the sample period. This is useful for experimentation with pseudo-doppler direction finding.

To cycle through four ports, one port every 1000 samples:

```
hackrf_operacake -m time -t A1:1000 -t A2:1000 -t A3:1000 -t A4:1000
```

When the duration on multiple ports is the same, the `-w` option can be used to set the default dwell time:

```
hackrf_operacake --mode time -w 1000 -t A1 -t A2 -t A3 -t A4
```

Only the A0 port connection is specified in time mode. Whenever the A0 connection is switched, the B0 connection is switched to the secondary port mirroring A0's secondary port. For example, when A0 switches to B2, B0 is switched to A2.

Once configured, an Opera Cake will remain in time mode until the mode is reconfigured or until the HackRF One is reset. You can pre-configure the Opera Cake in time mode, and the automatic switching will continue to work while using other software.

Although multiple Opera Cakes on a single HackRF One may be set to time mode at the same time, they share a single switching plan.

CHAPTER 3

FAQ

3.1 What is the Transmit Power of HackRF?

HackRF One's absolute maximum TX power varies by operating frequency:

- 1 MHz to 10 MHz: 5 dBm to 15 dBm, generally increasing as frequency increases (see this blog post)
- 10 MHz to 2150 MHz: 5 dBm to 15 dBm, generally decreasing as frequency increases
- 2150 MHz to 2750 MHz: 13 dBm to 15 dBm
- 2750 MHz to 4000 MHz: 0 dBm to 5 dBm, decreasing as frequency increases
- 4000 MHz to 6000 MHz: -10 dBm to 0 dBm, generally decreasing as frequency increases

Through most of the frequency range up to 4 GHz, the maximum TX power is between 0 and 10 dBm. The frequency range with best performance is 2150 MHz to 2750 MHz.

Overall, the output power is enough to perform over-the-air experiments at close range or to drive an external amplifier. If you connect an external amplifier, you should also use an external bandpass filter for your operating frequency.

Before you transmit, know your laws. HackRF One has not been tested for compliance with regulations governing transmission of radio signals. You are responsible for using your HackRF One legally.

3.2 What is the Receive Power of HackRF?

The maximum RX power of HackRF One is -5 dBm. Exceeding -5 dBm can result in permanent damage!

In theory, HackRF One can safely accept up to 10 dBm with the front-end RX amplifier disabled. However, a simple software or user error could enable the amplifier, resulting in permanent damage. It is better to use an external attenuator than to risk damage.

3.3 What is the minimum signal power level that can be detected by HackRF?

This isn't a question that can be answered for a general purpose SDR platform such as HackRF. Any answer would be very specific to a particular application. For example, an answerable question might be: What is the minimum power level in dBm of modulation M at frequency F that can be detected by HackRF One with software S under configuration C at a bit error rate of no more than E%? Changing any of those variables (M, F, S, C, or E) would change the answer to the question. Even a seemingly minor software update might result in a significantly different answer. To learn the exact answer for a specific application, you would have to measure it yourself.

HackRF's concrete specifications include operating frequency range, maximum sample rate, and dynamic range in bits. These specifications can be used to roughly determine the suitability of HackRF for a given application. Testing is required to finely measure performance in an application. Performance can typically be enhanced significantly by selecting an appropriate antenna, external amplifier, and/or external filter for the application.

3.4 Is HackRF full-duplex?

HackRF One is a half-duplex transceiver. This means that it can transmit or receive but not both at the same time.

3.5 Why isn't HackRF One full-duplex?

HackRF One is designed to support the widest possible range of SDR applications in a single, low cost, portable device. Many applications do not require full-duplex operation. Full-duplex support would have made HackRF larger and more expensive, and it would have required an external power supply. Since full-duplex needs can be met by simply using a second HackRF One, it made sense to keep the device small, portable, and low cost for everyone who does not require full-duplex operation.

3.6 How could the HackRF One design be changed to make it full-duplex?

The HackRF One hardware design is actually full-duplex (at lower sample rates) from the USB connection through the ADC/DAC. The RF section is the only part of the design that cannot support full-duplex operation. The easiest way to make HackRF One full-duplex would be to create an add-on board that duplicates the RF section and also provides an external power input (from a wall wart, for example) for the additional power required. This would also require software effort; the firmware, CPLD, libhackrf, and other host software would all need work to support full-duplex operation.

If you were to try to redesign the RF section on HackRF One to support full-duplex, the main thing to focus on would be the MAX2837 (intermediate frequency transceiver). This part is half-duplex, so you would either need two of them or you would have to redesign the RF section to use something other than the MAX2837, likely resulting in a radically different design. If you used two MAX2837s you might be able to use one RFFC5071 instead of two RFFC5072s.

3.7 Are those connectors SMA or RP-SMA?

Some connectors that appear to be SMA are actually RP-SMA. If you connect an RP-SMA antenna to HackRF One, it will seem to connect snugly but won't function at all because neither the male nor female side has a center pin. RP-SMA connectors are most common on 2.4 GHz antennas and are popular on Wi-Fi equipment. Adapters are available.

3.8 What is the big spike in the center of my received spectrum?

If you see a large spike in the center of your FFT display regardless of the frequenecy you are tuned to, you are seeing a DC offset (or component or bias). The term "DC" comes from "Direct Current" in electronics. It is the unchanging aspect of a signal as opposed to the "alternating" part of the signal (AC) that changes over time. Take, for example, the signal represented by the digital sequence:

```
-2, -1, 1, 6, 8, 9, 8, 6, 1, -1, -2, -1, 1, 6, 8, 9, 8, 6, 1, -1, -2, -1, 1, 6, 8, 9,␣
↪8, 6, 1, -1
```

This periodic signal contains a strong sinusoidal component spanning from -2 to 9. If you were to plot the spectrum of this signal, you would see one spike at the frequency of this sinusoid and a second spike at 0 Hz (DC). If the signal spanned from values -2 to 2 (centered around zero), there would be no DC offset. Since it is centered around 3.5 (the number midway between -2 and 9), there is a DC component.

Samples produced by HackRF are measurements of radio waveforms, but the measurement method is prone to a DC bias introduced by HackRF. It's an artifact of the measurement system, not an indication of a received radio signal. DC offset is not unique to HackRF; it is common to all quadrature sampling systems.

There was a bug in the HackRF firmware (through release 2013.06.1) that made the DC offset worse than it should have been. In the worst cases, certain Jawbreakers experienced a DC offset that drifted to a great extreme over several seconds of operation. This bug has been fixed. The fix reduces DC offset but does not do away with it entirely. It is something you have to live with when using any quadrature sampling system like HackRF.

A high DC offset is also one of a few symptoms that can be caused by a software version mismatch. A common problem is that people run an old version of gr-osmosdr with newer firmware.

3.9 What gain controls are provided by HackRF?

HackRF (both Jawbreaker and One) provides three different analog gain controls on RX and two on TX. The three RX gain controls are at the RF ("amp", 0 or 14 dB), IF ("lna", 0 to 40 dB in 8 dB steps), and baseband ("vga", 0 to 62 dB in 2 dB steps) stages. The two TX gain controls are at the RF (0 or 14 dB) and IF (0 to 47 dB in 1 dB steps) stages.

3.10 Why is the RF gain setting restricted to two values?

HackRF has two RF amplifiers close to the antenna port, one for TX and one for RX. These amplifiers have two settings: on or off. In the off state, the amps are completely bypassed. They nominally provide 14 dB of gain when on, but the actual amount of gain varies by frequency. In general, expect less gain at higher frequencies. For fine control of gain, use the IF and/or baseband gain options.

3.11 Why are the LEDs on HackRF different colours?

Each LED is a single color. There are no multi-colored LEDs on HackRF One. Adjacent LEDs are different colors in order to make them easier to distinguish from one another. The colors do not mean anything.

3.12 Where can I purchase HackRF?

HackRF is designed and manufactured by Great Scott Gadgets. We do not sell low volumes of HackRFs to people individually; instead we have agreements with specific resellers. Please see our reseller list on the Great Scott Gadgets website for availability: http://greatscottgadgets.com/hackrf/.

HackRF is open source hardware, so you can also build your own.

CHAPTER 4

Troubleshooting

4.1 How do I deal with the big spike in the middle of my spectrum?

Start by reading *our FAQ Response on the DC Spike*. After that, there are a few options:

1. Ignore it. For many applications it isn't a problem. You'll learn to ignore it.
2. Avoid it. The best way to handle DC offset for most applications is to use offset tuning; instead of tuning to your exact frequency of interest, tune to a nearby frequency so that the entire signal you are interested in is shifted away from 0 Hz but still within the received bandwidth. If your algorithm works best with your signal centered at 0 Hz (many do), you can shift the frequency in the digital domain, moving your signal of interest to 0 Hz and your DC offset away from 0 Hz. HackRF's high maximum sampling rate can be a big help as it allows you to use offset tuning even for relatively wideband signals.
3. Correct it. There are various ways of removing the DC offset in software. However, these techniques may degrade parts of the signal that are close to 0 Hz. It may look better, but that doesn't necessarily mean that it is better from the standpoint of a demodulator algorithm, for example. Still, correcting the DC offset is often a good choice.

4.2 How should I set the gain controls for RX?

A good default setting to start with is RF=0 (off), IF=16, baseband=16. Increase or decrease the IF and baseband gain controls roughly equally to find the best settings for your situation. Turn on the RF amp if you need help picking up weak signals. If your gain settings are too low, your signal may be buried in the noise. If one or more of your gain settings is too high, you may see distortion (look for unexpected frequencies that pop up when you increase the gain) or the noise floor may be amplified more than your signal is.

4.3 What are the minimum system requirements for using HackRF?

The most important requirement is that you supply 500 mA at 5 V DC to your HackRF via the USB port. If your host computer has difficulty meeting this requirement, you may need to use a powered USB hub.

Most users will want to stream data to or from the HackRF at high speeds. This requires that the host computer supports Hi-Speed USB. Some Hi-Speed USB hosts are better than others, and you may have multiple host controllers on your computer. If you have difficulty operating your HackRF at high sample rates (10 Msps to 20 Msps), try using a different USB port on your computer. If possible, arrange things so that the HackRF is the only device on the bus.

There is no specific minimum CPU requirement for the host computer, but SDR is generally a CPU-intensive application. If you have a slower CPU, you may be unable to run certain SDR software or you may only be able to operate at lower sample rates.

4.4 Why isn't HackRF working with my virtual machine (VM)?

HackRF requires the ability to stream data at very high rates over USB. Unfortunately VM software typically has problems with continuous high speed USB transfers.

There are some known bugs with the HackRF firmware's USB implementation. It is possible that fixing these bugs will improve the ability to operate HackRF with a VM, but there is a very good chance that operation at higher sample rates will still be limited.

4.5 What LEDs should be illuminated on the HackRF?

When HackRF One is plugged in to a USB host, four LEDs should turn on: 3V3, 1V8, RF, and USB. The 3V3 LED indicates that the primary internal power supply is working properly. The 1V8 and RF LEDs indicate that firmware is running and has switched on additional internal power supplies. The USB LED indicates that the HackRF One is communicating with the host over USB.

The RX and TX LEDs indicate that a receive or transmit operation is currently in progress.

4.6 I can't seem to access my HackRF under Linux

If you run `hackrf_info` or any other command which tries to communicate with the HackRF and get one of the following error messages

```
hackrf_open() failed: HACKRF_ERROR_NOT_FOUND (-5)
```

or:

```
hackrf_open() failed: HACKRF_ERROR_LIBUSB (-1000)
```

there are a few steps you can try:

1. Make sure that you are running the latest version of libhackrf and hackrf-tools. HackRF One, for example, is only supported by release 2014.04.1 or newer. Try running `hackrf_info` again to see if the updates have addressed your issue.

2. Write a udev rule to instruct udev to set permissions for the device in a way that it can be accessed by any user on the system who is a member of a specific group.

   A normal user under Linux doesn't have the permissions to access arbitrary USB devices because of security reasons. The first solution would be to run every command which tries to access the HackRF as root which is not recommended for daily usage, but at least shows you if your HackRF really works.

   To write a udev rule, you need to create a new rules file in the `/etc/udev/rules.d` folder. I called mine `52-hackrf.rules`. Here is the content:

   ```
   ATTR{idVendor}=="1d50", ATTR{idProduct}=="604b", SYMLINK+="hackrf-
   ↪jawbreaker-%k", MODE="660",   GROUP="plugdev"
   ATTR{idVendor}=="1d50", ATTR{idProduct}=="6089", SYMLINK+="hackrf-one-%k",
   ↪ MODE="660", GROUP="plugdev"
   ATTR{idVendor}=="1fc9", ATTR{idProduct}=="000c", SYMLINK+="hackrf-dfu-%k",
   ↪ MODE="660", GROUP="plugdev"
   ```

   The content of the file instructs udev to look out for devices with Vendor ID and Product ID matching HackRF devices. It then sets the UNIX permissions to `660` and the group to `plugdev` and creates a symlink in `/dev` to the device.

   After creating the rules file you can either reboot or run the command `udevadm control --reload-rules` as root to instruct udev to reload all rule files. After replugging your HackRF board, you should be able to access the device with all utilities as a normal user. If you still can't access the device, make sure that you are a member of the plugdev group.

   (These instructions have been tested on Ubuntu and Gentoo and may need to be adapted to other Linux distributions. In particular, your distro may have a group named something other than plugdev for this purpose.)

3. Disable USB autosuspend for HackRF. A common problem for laptop users could power management enabling USB autosuspend, which is likely if `hackrf_info` returns an error of `hackrf_open() failed: Input/Output Error (-1000)` on the first execution, and works if you run it a second time directly afterwards. This can be confirmed by running `LIBUSB_DEBUG=3 hackrf_info` and checking that the error is a `broken pipe`.

   If you use the TLP power manager you can add the HackRF USB VID/PIDs to the `USB_BLACKLIST` line in `/etc/default/tlp` (under Archlinux create a file `/etc/tlp.d/10-usb-blacklist.conf`, under Ubuntu the config file can be found at `/etc/tlp.conf`):

   ```
   USB_BLACKLIST="1d50:604b 1d50:6089 1d50:cc15 1fc9:000c"
   ```

   and restart TLP using `tlp restart` or `systemctl restart tlp`.

4.7 The command hackrf_info failed with "hackrf_open() .. HACKRF_ERROR_NOT_FOUND"

This could be a problem of a kernel driver. Some ubuntu versions, like Ubuntu 15.04 with installed gnuradio has a kernel driver pre-installed. In this case you probably will get some syslog kernel messages like:

- kernel: [8932.297074] hackrf 1-9.4:1.0: Board ID: 02
- kernel: [8932.297076] hackrf 1-9.4:1.0: Firmware version: 2014.08.1
- kernel: [8932.297261] hackrf 1-9.4:1.0: Registered as swradio0
- kernel: [8932.297262] hackrf 1-9.4:1.0: SDR API is still slightly experimental and functionality changes may follow

when you plug in the the HackRF module. Use the command `dmesg` to check the last system log entries. If you try to start `hackrf_info` it will terminate with the error message and the system log will show a message like:

- kernel: [8967.263268] usb 1-9.4: usbfs: interface 0 claimed by hackrf while 'hackrf_info' sets config #1

To solve this issue check under root account if is there is a kernel module `hackrf` loaded: `lsmod | grep hackrf`. If there is a hackrf kernel module, try to unload it with `rmmod hackrf`. You must do this command as root, too. After this the command `hackrf_info` (and all other hackrf related stuff) should work and the syslog usbfs massage should vanish.

After a reset or USB unplug/plug this kernel module will load again and block the access again. To solve this you have to blacklist the hackrf kernel module in /etc/modprobe.d/blacklist(.conf) The current filename of the blacklist file may differ, it depends on the current ubuntu version. In ubuntu 15.04 it is located in /etc/modprobe.d/blacklist.conf. Open this file under root account with a text editor an add the following line at the end:

```
blacklist hackrf
```

After a system-restart, to get the updated modprobe working, the hackrf worked under ubuntu 15.04 with the upstream packages (Firmware version: 2014.08.1) out-of-the-box.

CHAPTER 5

Getting Help

Before asking for help with HackRF, check to see if your question is listed in the *FAQ* or has already been answered in GitHub issues or the mailing list archives.

For assistance with HackRF use or development, please look at the issues on the GitHub project. This is the preferred place to ask questions so that others may locate the answer to your question in the future.

Many users spend time in the #hackrf channel on Discord.

CHAPTER 6

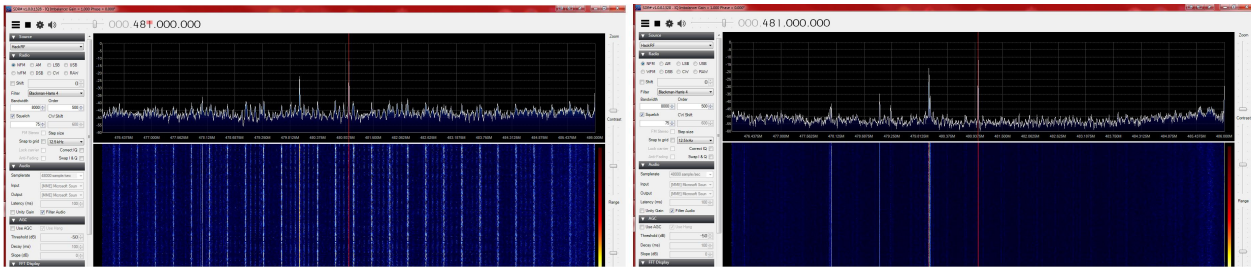
Tips and Tricks

6.1 USB Cables (and why to use a noise reducing one)

The USB cable you choose can make a big difference in what you see when using your HackRF and especially when using it around between 120 and 480 MHz where USB is doing all its work.

1. Use a shielded USB cable. The best way to guarantee RF interference from USB is to use an unshielded cable. You can test that your cable is shielded by using a continuity tester to verify that the shield on one connector has continuity to the shield on the connector at the other end of the cable.
2. Use a short USB cable. Trying anything larger than a 6ft cable may yield poor results. The longer the cable, the more loss you can expect and when making this post a 15ft cable was tried and the result was the HackRF would only power up half way.
3. For best results, select a cable with a ferrite core. These cables are usually advertised to be noise reducing and are recognizable from the plastic block towards one end.

Screenshot before and after changing to a noise reducing cable (view full size image):

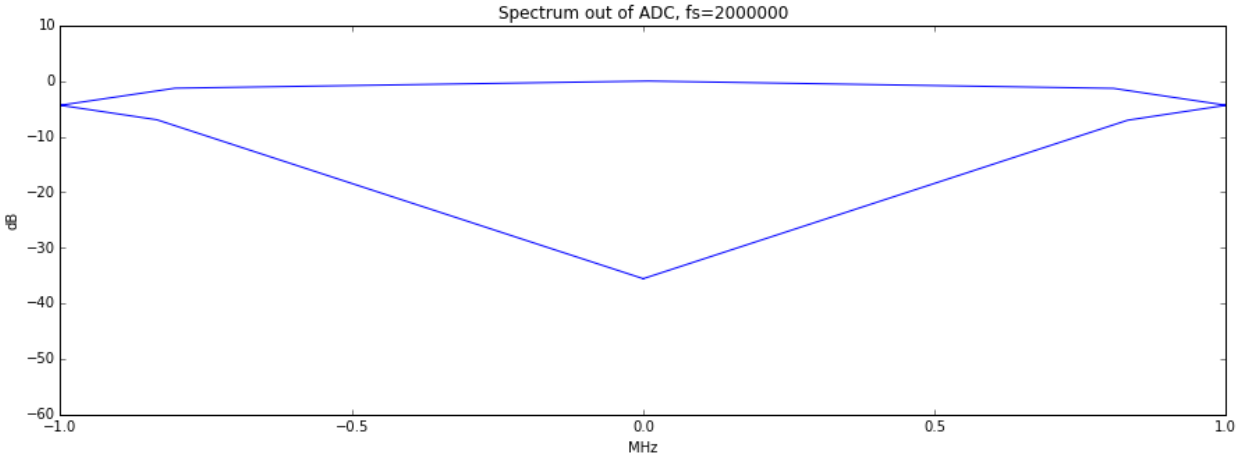


A shielded cable with ferrite core was used in the right-hand image.

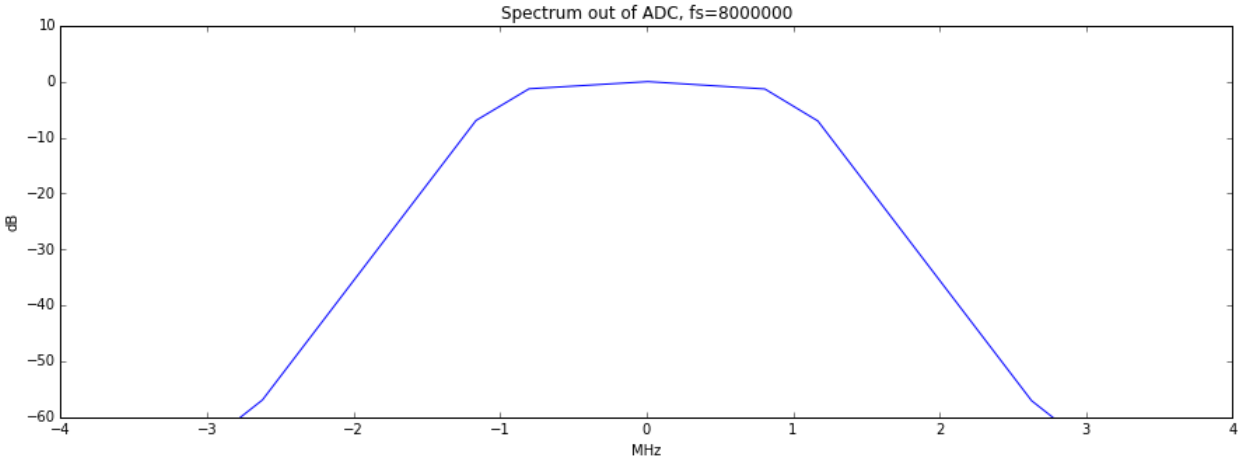
The before and after images were both taken with the preamp on and the LNA and VGA both set to 24db.

6.2 Sampling Rate and Baseband Filters

Using a sampling rate of less than 8MHz is not recommended. Partly, this is because the MAX5864 (ADC/DAC chip) isn't specified to operate at less than 8MHz, and therefore, no promises are made by Maxim about how it performs. But more importantly, the baseband filter in the MAX2837 has a minimum bandwidth of 1.75MHz. It can't provide enough filtering at 2MHz sampling rate to remove substantial signal energy in adjacent spectrum (more than +/-1MHz from the tuned frequency). The MAX2837 datasheet suggests that at +/-1MHz, the filter provides only 4dB attenuation, and at +/-2MHz (where a signal would alias right into the center of your 2MHz spectrum), it attenuates about 33dB. That's significant. Here's a picture:



At 8MHz sampling rate, and using the minimum 1.75MHz bandwidth filter, this is the response:



You can see that the attenuation is more than 60dB at +/-2.8MHz, which is more than sufficient to remove significant adjacent spectrum interference before the ADC digitizes the baseband. If using this configuration to get a 2MHz sampling rate, use a GNU Radio block after the 8MHz source that performs a 4:1 decimation with a decently sharp low pass filter (complex filter with a cut-off of <1MHz).

CHAPTER 7

HackRF Community Projects and Mentions

Have you done something cool with HackRF or mentioned HackRF in one of your presentations? Let us know and we might post a link here!

- HackRF vs. Tesla Model S (Sam Edwards)
- Jawbreaker/VFD spectrum analyzer (Jared Boone)
- LEGO car (Michael Ossmann)
- wireless microphones (Jared Boone)

7.1 Retired Projects

- Automotive Remote Keyless Entry Systems (Mike Kershaw)
- Decoding Pocsag Pagers With The HackRF (BinaryRF)
- Sniffing GSM with HackRF (BinaryRF)

CHAPTER 8

Installing HackRF Software

8.1 Install Using Package Managers

Unless developing or testing new features for HackRF, we highly recommend that most users use build systems or package managers provided for their operating system. **Our suggested operating system for use with HackRF is Ubuntu**.

8.1.1 FreeBSD

You can use the binary package: `# pkg install hackrf`

You can also build and install from ports:

```
# cd /usr/ports/comms/hackrf
# make install
```

8.1.2 Linux: Arch

```
pacman -S hackrf
```

8.1.3 Linux: Fedora / Red Hat

```
sudo dnf install hackrf -y
```

8.1.4 Linux: Gentoo

```
emerge -a net-wireless/hackrf-tools
```

8.1.5 Linux: Ubuntu / Debian

```
sudo apt-get install hackrf
```

8.1.6 OS X (10.5+): Homebrew

```
brew install hackrf
```

8.1.7 OS X (10.5+): MacPorts

```
sudo port install hackrf
```

8.1.8 Windows: Binaries

Binaries are provided as part of the PothosSDR project, they can be downloaded here.

8.2 Installing From Source

8.2.1 Linux / OS X / *BSD: Building HackRF Software From Source

Acquire the source for the HackRF tools from either a release archive or git: `git clone https://github.com/mossmann/hackrf.git`

Once you have the source downloaded, the host tools can be built as follows:

```
cd hackrf/host
mkdir build
cd build
cmake ..
make
sudo make install
sudo ldconfig
```

If you have HackRF hardware, you may need to *update the firmware* to match the host tools versions.

8.2.2 Windows: Prerequisites for Cygwin, MinGW, or Visual Studio

- cmake-2.8.12.1 or later from http://www.cmake.org/cmake/resources/software.html
- libusbx-1.0.18 or later from http://sourceforge.net/projects/libusbx/files/latest/download?source=files
- fftw-3.3.5 or later from http://www.fftw.org/install/windows.html
- **Install Windows driver for HackRF hardware or use Zadig see http://sourceforge.net/projects/libwdi/files/zadig**
  - If you want to use Zadig select HackRF USB device and just install/replace it with WinUSB driver.

Note for Windows build: You shall always execute hackrf-tools from Windows command shell and not from Cygwin or MinGW shell because on Cygwin/MinGW Ctrl+C is not managed correctly and especially for hackrf_transfer the Ctrl+C (abort) will not stop correctly and will corrupt the file.

8.2.3 Windows: Installing HackRF Software via Cygwin

```
mkdir host/build
cd host/build
cmake ../ -G "Unix Makefiles" -DCMAKE_LEGACY_CYGWIN_WIN32=1 -DLIBUSB_INCLUDE_DIR=/usr/
↪local/include/libusb-1.0/
make
make install
```

8.2.4 Windows: Installing HackRF Software via MinGW

```
mkdir host/build
cd host/build
cmake ../ -G "MSYS Makefiles" -DLIBUSB_INCLUDE_DIR=/usr/local/include/libusb-1.0/
make
make install
```

8.2.5 Windows: Installing HackRF Software via Visual Studio 2015 x64

Create library definition for MSVC to link to `C:\fftw-3.3.5-dll64> lib /machine:x64 /def:libfftw3f-3.def`

```
c:\hackrf\host\build> cmake ../ -G "Visual Studio 14 2015 Win64" \
-DLIBUSB_INCLUDE_DIR=c:\libusb-1.0.21\libusb \
-DLIBUSB_LIBRARIES=c:\libusb-1.0.21\MS64\dll\lib\libusb-1.0.lib \
-DTHREADS_PTHREADS_INCLUDE_DIR=c:\pthreads-w32-2-9-1-release\Pre-built.2\include \
-DTHREADS_PTHREADS_WIN32_LIBRARY=c:\pthreads-w32-2-9-1-release\Pre-built.
↪2\lib\x64\pthreadVC2.lib \
-DFFTW_INCLUDES=C:\fftw-3.3.5-dll64 \
-DFFTW_LIBRARIES=C:\fftw-3.3.5-dll64\libfftw3f-3.lib
```

CMake will produce a solution file named `HackRF.sln` and a series of project files which can be built with msbuild as follows: `c:\hackrf\host\build> msbuild HackRF.sln`

CHAPTER 9

Getting Started with HackRF and GNU Radio

We recommend getting started by watching the Software Defined Radio with HackRF video series. This series will introduce you to HackRF One, software including GNU Radio, and teach you the fundamentals of Digital Signal Processing (DSP) needed to take full advantage of the power of Software Defined Radio (SDR). Additional helpful information follows.

9.1 Try Your HackRF with Pentoo Linux

The easiest way to get started with your HackRF and ensure that it works is to use Pentoo, a Linux distribution with full support for HackRF and GNU Radio. Download the latest Pentoo .iso image from one of the mirrors listed at http://pentoo.ch/downloads/. Then burn the .iso to a DVD or use UNetbootin to install the .iso on a USB flash drive. Boot your computer using the DVD or USB flash drive to run Pentoo. Do this natively, not in a virtual machine. (Unfortunately high speed USB operation invariably fails when people try to run HackRF from a virtual machine.)

Once Pentoo is running, you can immediately use it to *update firmware* on your HackRF or use other HackRF command line tools. For a walkthrough, watch SDR with HackRF, Lesson 5: HackRF One.

To verify that your HackRF is detected, type `hackrf_info` at the command line. It should produce a few lines of output including "Found HackRF board." The 3V3, 1V8, RF, and USB LEDs should all be illuminated and are various colors.

You can type `startx` at the command line to launch a desktop environment. Accept the "default config" in the first dialog box. The desktop environment is useful for GNU Radio Companion and other graphical applications but is not required for basic operations such as firmware updates.

Now you can use programs such as gnuradio-companion or gqrx to start experimenting with your HackRF. Try the Examples below. If you are new to GNU Radio, an excellent place to start is with the SDR with HackRF video series or with the GNU Radio guided tutorials.

Alternative: GNU Radio Live SDR Environment

The GNU Radio Live SDR Environment is another nice bootable Linux .iso with support for HackRF and, of course, GNU Radio.

9.2 Software Setup

As mentioned above, the best way to get started with HackRF is to use Pentoo Linux. Eventually you may want to install software to use HackRF with your favorite operating system.

If your package manager includes the most recent release of libhackrf and gr-osmosdr, then use it to install those packages in addition to GNU Radio. Otherwise, the recommended way to install these tools is by using PyBOMBS.

See the *Operating System Tips* page for information on setting up HackRF software on particular Operating Systems and Linux distributions.

If you have any trouble, make sure that things work when booted to Pentoo. This will allow you to easily determine if your problem is being caused by hardware or software, and it will give you a way to see how the software is supposed to function.

9.3 Examples

A great way to get started with HackRF is the SDR with HackRF video series. Additional examples follow:

Testing the HackRF

1. Plug in the HackRF
2. run the hackrf_info command `$ hackrf_info`

If everything is OK, you should see something similar to the following:

```
hackrf_info version: 2017.02.1
libhackrf version: 2017.02.1 (0.5)
Found HackRF
Index: 0
Serial number: 0000000000000000################
Board ID Number: 2 (HackRF One)
Firmware Version: 2017.02.1 (API:1.02)
Part ID Number: 0x######## 0x########
```

FM Radio Example

This Example was derived from the following works:

- RTL-SDR FM radio receiver with GNU Radio Companion
- How To Build an FM Receiver with the USRP in Less Than 10 Minutes

1. Download the FM Radio Receiver python file here
2. Run the file `$ python ./fm_radio_rx.py`
3. You can find the GNU Radio Companion source file here

CHAPTER 10

HackRF Compatible Software

10.1 Software with HackRF Support

This is intended to be a list of software known to work with the HackRF. There are three sections, GNU Radio Based software, those that have direct support, and those that can work with data from the HackRF.

10.2 GNU Radio Based

GNU Radio Mode-S/ADS-B - https://github.com/bistromath/gr-air-modes

GQRX - http://gqrx.dk/

10.3 Direct Support

SDR# (Windows only) - https://airspy.com/download/

- Only nightly builds currently support HackRF One - http://sdrsharp.com/downloads/sdr-nightly.zip

SDR_Radio.com V2 - http://v2.sdr-radio.com/Radios/HackRF.aspx

Universal Radio Hacker (Windows/Linux) - https://github.com/jopohl/urh

QSpectrumAnalyzer - https://github.com/xmikos/qspectrumanalyzer

Spectrum Analyzer GUI for hackrf_sweep for Windows - https://github.com/pavsa/hackrf-spectrum-analyzer

Web-based APRS tracker https://xakcop.com/aprs-sdr

10.4 Can use HackRF data

Inspectrum https://github.com/miek/inspectrum

- Capture analysis tool with advanced features

Baudline http://www.baudline.com/ (Can view/process HackRF data, e.g. hackrf_transfer)

10.5 HackRF Tools

In addition to third party tools that support HackRF, we provide some commandline tools for interacting with HackRF. For information on how to use each tool look at the help information provided (e.g. `hackrf_transfer -h`) or the manual pages.

The first two tools (`hackrf_info` and `hackrf_transfer`) should cover most usage. The remaining tools are provided for debugging and general interest; beware, they have the potential to damage HackRF if used incorrectly.

- **hackrf_info** Read device information from HackRF such as serial number and firmware version.
- **hackrf_transfer** Send and receive signals using HackRF. Input/output can be 8bit signed quadrature files or wav files.
- **hackrf_max2837** Read and write registers in the Maxim 2837 transceiver chip. For most tx/rx purposes hackrf_transfer or other tools will take care of this for you.
- **hackrf_rffc5071** Read and write registers in the RFFC5071 mixer chip. As above, this is for curiosity or debugging only, most tools will take care of these settings automatically.
- **hackrf_si5351c** Read and write registers in the Silicon Labs Si5351C clock generator chip. This should also be unnecessary for most operation.
- **hackrf_spiflash** A tool to write new firmware to HackRF. This is mostly used for *Updating Firmware*.
- **hackrf_cpldjtag** A tool to update the CPLD on HackRF. This is needed only when *Updating Firmware* to a version prior to 2021.03.1.

10.6 Handling HackRF data

10.6.1 Matlab

```
fid = open('samples.bin',  'r');
len = 1000; % 1000 samples
y = fread(fid, 2*len, 'int8');
y = y(1:2:end) + 1j*y(2:2:end);
fclose(fid)
```

CHAPTER 11

libhackRF API

This document describes the functions, data structures and constants that libHackRF provides. It should be used as a reference for using libHackRF and the HackRF hardware.

If you are writing a generic SDR application, i.e. not tied to the HackRF hardware, we strongly recommend that you use either gr-osmosdr or SoapySDR to provide support for the broadest possible range of software defined radio hardware.

For example usage of many of these functions, see the hackrf_transfer tool.

11.1 Setup, Initialization and Shutdown

11.1.1 HackRF Init

Initialize libHackRF, including global libUSB context to support multiple HackRF hardware devices.

Syntax: `int hackrf_init()`

Returns: A value from the hackrf_error constants listed below.

11.1.2 HackRF Open

Syntax: `int hackrf_open(hackrf_device** device)`

Returns: A value from the hackrf_error constants listed below.

11.1.3 HackRF Device List

Retrieve a list of HackRF devices attached to the system. This function finds all devices, regardless of permissions or availability of the hardware.

Syntax: `hackrf_device_list_t* hackrf_device_list()`

Returns: A pointer to a hackrf_device_list_t struct, a list of HackRF devices attached to the system. The contents of the hackrf_device_list_t struct are decribed in the data structures section below.

11.1.4 HackRF Device List Open

Open and acquire a handle on a device from the hackrf_device_list_t struct.

Syntax: `int hackrf_device_list_open(hackrf_device_list_t* list, int idx, hackrf_device** device)`

Params:

`list` - A pointer to a hackrf_device_list_t returned by `hackrf_device_list()`

`idx` - The list index of the HackRF device to open

`device` - Output location for hackrf_device pointer. Only valid when return value is HACKRF_SUCCESS.

Returns: A value from the hackrf_error constants listed below.

11.1.5 HackRF Device List Free

Syntax: `void hackrf_device_list_free(hackrf_device_list_t* list)`

Params:

`list` - A pointer to a hackrf_device_list_t returned by `hackrf_device_list()`

11.1.6 HackRF Open By Serial

Syntax: `int hackrf_open_by_serial(const char* const desired_serial_number, hackrf_device** device)`

Returns:

11.1.7 HackRF Close

Syntax: `int hackrf_close(hackrf_device* device)`

Returns: A value from the hackrf_error constants listed below.

11.1.8 HackRF Exit

Cleanly shutdown libHackRF and the underlying USB context. This does not stop in progress transfers or close the HackRF hardware. `hackrf_close()` should be called before this to cleanly close the connection to the hardware.

Syntax: `int hackrf_exit()`

Returns: A value from the hackrf_error constants listed below.

11.2 Using the Radio

11.2.1 HackRF Start Rx

Syntax: `int hackrf_start_rx(hackrf_device*, hackrf_sample_block_cb_fn, void* rx_ctx)`

Params:

Returns: A value from the hackrf_error constants listed below.

11.2.2 HackRF Stop Rx

Syntax: `int hackrf_stop_rx(hackrf_device*)`

Params:

Returns: A value from the hackrf_error constants listed below.

11.2.3 HackRF Start Tx

Syntax: `int hackrf_start_tx(hackrf_device*, hackrf_sample_block_cb_fn, void* tx_ctx)`

Params:

Returns: A value from the hackrf_error constants listed below.

11.2.4 HackRF Stop Tx

Syntax: `int hackrf_stop_tx(hackrf_device*)`

Params:

Returns: A value from the hackrf_error constants listed below.

11.2.5 HackRF Set Baseband Filter Bandwidth

Syntax: `int hackrf_set_baseband_filter_bandwidth(hackrf_device*, const uint32_t bandwidth_hz)`

Params:

Returns: A value from the hackrf_error constants listed below.

11.2.6 HackRF Compute Baseband Filter BW

Compute best default value depending on sample rate (auto filter).

Syntax: `uint32_t hackrf_compute_baseband_filter_bw(const uint32_t bandwidth_hz)`

Params:

Returns: A valid baseband filter width available from the Maxim MAX2837 frontend used by the radio.

11.2.7 HackRF Compute Baseband Filter BW Round Down LT

Compute nearest freq for bw filter (manual filter)

Syntax: `uint32_t hackrf_compute_baseband_filter_bw_round_down_lt(const uint32_t bandwidth_hz)`

Params:

Returns: A valid baseband filter width available from the Maxim MAX2837 frontend used by the radio.

11.3 Reading and Writing Registers

11.3.1 HackRF MAX2837 Read

Read register values from the MAX2837 Baseband IC.

Syntax: `int hackrf_max2837_read(hackrf_device* device, uint8_t register_number, uint16_t* value)`

Params:

Returns:

11.3.2 HackRF MAX2837 Write

Write register values to the MAX2837 Baseband IC.

Syntax: `int hackrf_max2837_write(hackrf_device* device, uint8_t register_number, uint16_t value)`

Params:

Returns:

11.3.3 HackRF Si5351C Read

Read register values from the Si5351C clock generator IC.

Syntax: `int hackrf_si5351c_read(hackrf_device* device, uint16_t register_number, uint16_t* value)`

Params:

Returns:

11.3.4 HackRF Si5351C Write

Write register values to the Si5351C clock generator IC.

Syntax: `int hackrf_si5351c_write(hackrf_device* device, uint16_t register_number, uint16_t value)`

Params:

Returns:

11.3.5 HackRF RFFC5071 Read

Read register values from the RFFC5071 mixer IC.

Syntax: `int hackrf_rffc5071_read(hackrf_device* device, uint8_t register_number, uint16_t* value)`

Params:

Returns:

11.3.6 HackRF RFFC5071 Write

Write register values to the RFFC5071 mixer IC.

Syntax: `int hackrf_rffc5071_write(hackrf_device* device, uint8_t register_number, uint16_t value)`

Params:

Returns:

11.4 Updating Firmware

11.4.1 HackRF CPLD Write

Device will need to be reset after hackrf_cpld_write.

Syntax: `int hackrf_cpld_write(hackrf_device* device, unsigned char* const data, const unsigned int total_length)`

Params:

Returns:

11.4.2 HackRF SPI Flash Erase

Syntax: `int hackrf_spiflash_erase(hackrf_device* device)`

Params:

Returns:

11.4.3 HackRF SPI Flash Write

Syntax: `int hackrf_spiflash_write(hackrf_device* device, const uint32_t address, const uint16_t length, unsigned char* const data)`

Params:

Returns:

11.4.4 HackRF SPI Flash Read

Syntax: `int hackrf_spiflash_read(hackrf_device* device, const uint32_t address, const uint16_t length, unsigned char* data)`

Params:

Returns:

11.5 Board Identifiers

11.5.1 HackRF Board ID Read

Syntax: `int hackrf_board_id_read(hackrf_device* device, uint8_t* value)`

Params:

Returns:

11.5.2 HackRF Version String Read

Syntax: `int hackrf_version_string_read(hackrf_device* device, char* version, uint8_t length)`

Params:

Returns:

11.5.3 HackRF Board Part ID Serial Number Read

Syntax: `int hackrf_board_partid_serialno_read(hackrf_device* device, read_partid_serialno_t* read_partid_serialno)`

Params:

Returns:

11.6 Miscellaneous

11.6.1 HackRF Error Name

Syntax: `const char* hackrf_error_name(enum hackrf_error errcode)`

Params:

Returns:

11.6.2 HackRF Board ID Name

Syntax: `const char* hackrf_board_id_name(enum hackrf_board_id board_id)`

Params:

Returns:

11.6.3 HackRF USB Board ID Name

Syntax: `const char* hackrf_usb_board_id_name(enum hackrf_usb_board_id usb_board_id)`

Params:

Returns:

11.6.4 HackRF Filter Path Name

Syntax: `const char* hackrf_filter_path_name(const enum rf_path_filter path)`

Params:

Returns:

11.7 Data Structures

`typedef struct hackrf_device hackrf_device`

```
typedef struct {
        hackrf_device* device;
        uint8_t* buffer;
        int buffer_length;
        int valid_length;
        void* rx_ctx;
        void* tx_ctx;
} hackrf_transfer;
```

```
typedef struct {
        uint32_t part_id[2];
        uint32_t serial_no[4];
} read_partid_serialno_t;
```

```
typedef struct {
        char **serial_numbers;
        enum hackrf_usb_board_id *usb_board_ids;
        int *usb_device_index;
        int devicecount;

        void **usb_devices;
        int usb_devicecount;
} hackrf_device_list_t;
```

`typedef int (*hackrf_sample_block_cb_fn)(hackrf_transfer* transfer)`

11.8 Enumerations

11.8.1 Supported board versions

These values identify the board type of the connected hardware. This value can be used as an indicator of capabilities, such as frequency range, bandwidth or antenna port power.

Board	Frequency range	Bandwidth	Antenna port power
HackRF One	1MHz - 6Ghz	20MHz	Yes
Jawbreaker	10MHz - 6GHz	20MHz	No
Rad1o	50MHz - 4GHz	20MHz	Unknown
Jellybean	N/A	20MHz	No

Most boards will identify as HackRF One, Jawbreaker or Rad1o. Jellybean was a pre-production revision of HackRF. No hardware device should intentionally report itself with an invalid board ID.

```
enum hackrf_board_id {
        BOARD_ID_JELLYBEAN  = 0,
        BOARD_ID_JAWBREAKER = 1,
        BOARD_ID_HACKRF_ONE = 2,
        BOARD_ID_RAD1O = 3,
        BOARD_ID_INVALID = 0xFF,
};
```

11.8.2 USB Product IDs

```
enum hackrf_usb_board_id {
        USB_BOARD_ID_JAWBREAKER = 0x604B,
        USB_BOARD_ID_HACKRF_ONE = 0x6089,
        USB_BOARD_ID_RAD1O = 0xCC15,
        USB_BOARD_ID_INVALID = 0xFFFF,
};
```

11.8.3 Transceiver Mode

HackRF can operate in three main transceiver modes, Receive, Transmit and Signal Source. There is also a CPLD update mode which is used to write firmware images to the CPLD.

The transceiver mode can be changed with `hackrf_set_transceiver_mode` with the value parameter set to one of the following:

```
enum transceiver_mode_t {
        TRANSCEIVER_MODE_OFF = 0,
        TRANSCEIVER_MODE_RX = 1,
        TRANSCEIVER_MODE_TX = 2,
        TRANSCEIVER_MODE_SS = 3,
        TRANSCEIVER_MODE_CPLD_UPDATE = 4
};
```

Receive mode (TRANSCEIVER_MODE_RX) is used to stream samples from the radio to the host system. Use `hackrf_set_freq` to set the center frequency of receiver and `hackrf_set_sample_rate` to set the sample rate (effective bandwidth).

Transmit mode (TRANSCEIVER_MODE_TX) is used to stream samples from the host to the radio.

See hackrf_transfer for an example of setting transmit and receive mode and transferring data over USB.

11.8.4 Function return values

11.8.5 RF Filter Path

```
enum rf_path_filter {
        RF_PATH_FILTER_BYPASS = 0,
        RF_PATH_FILTER_LOW_PASS = 1,
        RF_PATH_FILTER_HIGH_PASS = 2,
};
```

CHAPTER 12

hackrf_sweep

12.1 Usage

```
[-h] # this help
[-d serial_number] # Serial number of desired HackRF
[-a amp_enable] # RX RF amplifier 1=Enable, 0=Disable
[-f freq_min:freq_max] # minimum and maximum frequencies in MHz
[-p antenna_enable] # Antenna port power, 1=Enable, 0=Disable
[-l gain_db] # RX LNA (IF) gain, 0-40dB, 8dB steps
[-g gain_db] # RX VGA (baseband) gain, 0-62dB, 2dB steps
[-n num_samples] # Number of samples per frequency, 8192-4294967296
[-w bin_width] # FFT bin width (frequency resolution) in Hz
[-1] # one shot mode
[-B] # binary output
[-I] # binary inverse FFT output
-r filename # output file
```

12.2 Output fields

`date, time, hz_low, hz_high, hz_bin_width, num_samples, dB, dB, ...`

Running `hackrf_sweep -f 2400:2490` gives the following example results:

Date	Time	Hz Low	Hz High	Hz bin width	Num Samples	dB	dB	dB	dB	dB
2019-01-03	11:57:34.9	2480500000	2405000000	1000000.00	20	-64.72	-63.36	-60.91	-61.74	-58.58
2019-01-03	11:57:34.9	2480500000	2415000000	1000000.00	20	-69.22	-60.67	-59.50	-61.81	-58.16
2019-01-03	11:57:34.9	2480500000	2410000000	1000000.00	20	-61.19	-70.14	-60.10	-57.91	-61.97
2019-01-03	11:57:34.9	2480500000	2420000000	1000000.00	20	-72.93	-79.14	-68.79	-70.71	-82.78
2019-01-03	11:57:34.9	2480500000	2425000000	1000000.00	20	-67.57	-61.61	-57.29	-61.90	-70.19
2019-01-03	11:57:34.9	2480500000	2435000000	1000000.00	20	-56.04	-59.58	-66.24	-66.02	-62.12

Two ranges of 5 MHz are analyzed at once from the same set of samples, so a single timestamp applies to the whole range.

The fifth column tells you the width in Hz (1 MHz in this case) of each frequency bin, which you can set with `-w`. The sixth column is the number of samples analyzed to produce that row of data.

Each of the remaining columns shows the power detected in each of several frequency bins. In this case there are five bins, the first from 2400 to 2401 MHz, the second from 2401 to 2402 MHz, and so forth.

CHAPTER 13

Updating Firmware

HackRF devices ship with firmware on the SPI flash memory. The firmware can be updated with nothing more than a USB cable and host computer.

These instructions allow you to upgrade the firmware in order to take advantage of new features or bug fixes.

If you have any difficulty making this process work from your native operating system, you can *use Pentoo or the GNU Radio Live DVD* to perform the updates.

13.1 Updating the SPI Flash Firmware

To update the firmware on a working HackRF One, use the hackrf_spiflash program:

```
hackrf_spiflash -w hackrf_one_usb.bin
```

You can find the firmware binary (hackrf_one_usb.bin) in the firmware-bin directory of the latest release package or you can compile your own from the source. For Jawbreaker, use hackrf_jawbreaker_usb.bin. If you compile from source, the file will be called hackrf_usb.bin.

The hackrf_spiflash program is part of hackrf-tools.

When writing a firmware image to SPI flash, be sure to select firmware with a filename ending in ".bin".

After writing the firmware to SPI flash, you may need to reset the HackRF device by pressing the RESET button or by unplugging it and plugging it back in.

If you get an error that mentions HACKRF_ERROR_NOT_FOUND, check out the *FAQ*. It's often a permissions problem that can be quickly solved.

13.2 Updating the CPLD

Older versions of HackRF firmware (prior to release 2021.03.1) require an additional step to program a bitstream into the CPLD.

To update the CPLD image, first update the SPI flash firmware, libhackrf, and hackrf-tools to the version you are installing. Then:

```
hackrf_cpldjtag -x firmware/cpld/sgpio_if/default.xsvf
```

After a few seconds, three LEDs should start blinking. This indicates that the CPLD has been programmed successfully. Reset the HackRF device by pressing the RESET button or by unplugging it and plugging it back in.

13.3 Only if Necessary: DFU Boot

DFU boot mode is normally only needed if the firmware is not working properly or has never been installed.

The LPC4330 microcontroller on HackRF is capable of booting from several different code sources. By default, HackRF boots from SPI flash memory (SPIFI). It can also boot HackRF in DFU (USB) boot mode. In DFU boot mode, HackRF will enumerate over USB, wait for code to be delivered using the DFU (Device Firmware Update) standard over USB, and then execute that code from RAM. The SPIFI is normally unused and unaltered in DFU mode.

To start up HackRF One in DFU mode, hold down the DFU button while powering it on or while pressing and releasing the RESET button. Release the DFU button after the 3V3 LED illuminates. The 1V8 LED should remain off. At this point HackRF One is ready to receive firmware over USB.

To start up Jawbreaker in DFU mode, short two pins on one of the "BOOT" headers while power is first supplied. The pins that must be shorted are pins 1 and 2 of header P32 on Jawbreaker. Header P32 is labeled "P2_8" on most Jawbreakers but may be labeled "2" on prototype units. Pin 1 is labeled "VCC". Pin 2 is the center pin. After DFU boot, you should see VCCLED illuminate and note that 1V8LED does not illuminate. At this point Jawbreaker is ready to receive firmware over USB.

You should only use a firmware image with a filename ending in ".dfu" over DFU, not firmware ending in ".bin".

13.4 Only if Necessary: Recovering the SPI Flash Firmware

If the firmware installed in SPI flash has been damaged or if you are programming a home-made HackRF for the first time, you will not be able to immediately use the hackrf_spiflash program as listed in the above procedure. Follow these steps instead:

1. Follow the DFU Boot instructions to start the HackRF in DFU boot mode.
2. Type `dfu-util --device 1fc9:000c --alt 0 --download hackrf_one_usb.dfu` to load firmware from a release package into RAM. If you have a Jawbreaker, use hackrf_jawbreaker_usb.dfu instead. Alternatively, use `make -e BOARD=HACKRF_ONE RUN_FROM=RAM program` to load the firmware into RAM and start it.
3. Follow the SPI flash firmware update procedure above to write the ".bin" firmware image to SPI flash.

13.5 Obtaining DFU-Util

On fresh installs of your OS, you may need obtain a copy of DFU-Util. For most Linux distributions it should be available as a package, for example on Debian/Ubuntu

```
sudo apt-get install dfu-util
```

If you are using a platform without a dfu-util package, build instruction can be found here on the dfu-util source forge build page.

```
cd ~
sudo apt-get build-dep dfu-util
sudo apt-get install libusb-1.0-0-dev
git clone git://git.code.sf.net/p/dfu-util/dfu-util
cd dfu-util
./autogen.sh
./configure
make
sudo make install
```

Now you will have the current version of DFU Util installed on your system.

CHAPTER 14

Firmware Development Setup

Firmware build instructions are included in the repository under firmware/README:

https://github.com/mossmann/hackrf/blob/master/firmware/README

CHAPTER 15

LPC43xx Debugging

Various debugger options for the LPC43xx exist.

15.1 Black Magic Probe

https://github.com/blacksphere/blackmagic

An example of using gdb with the Black Magic Probe:

```
arm-none-eabi-gdb -n blinky.elf
target extended-remote /dev/ttyACM0
monitor swdp_scan
attach 1
set {int}0x40043100 = 0x10000000
load
cont
```

It is possible to attach to the M0 instead of the M4 if you use jtag_scan instead of swdp_scan, but the Black Magic Probe had some bugs when trying to work with the M0 the last time I tried it.

15.2 LPC-Link

(included with LPCXpresso boards)

TitanMKD has had some success. See the tutorial in hackrf/doc/LPCXPresso_Flash_Debug_Tutorial.pdf or .odt (PDF and OpenOffice document) Doc Link [https://github.com/mossmann/hackrf/tree/master/doc)

15.3 ST-LINK/V2

15.3.1 Hardware Configuration

Start with an STM32F4-Discovery board. Remove the jumpers from CN3. Connect the target's SWD interface to CN2 "SWD" connector.

15.3.2 Software Configuration

I'm using libusb-1.0.9.

Install OpenOCD-0.6.0 dev

```
# Cloned at hash a21affa42906f55311ec047782a427fcbcb98994
git clone git://openocd.git.sourceforge.net/gitroot/openocd/openocd
cd openocd
./bootstrap
./configure --enable-stlink --enable-buspirate --enable-jlink --enable-maintainer-mode
make
sudo make install
```

OpenOCD configuration files

openocd.cfg

```
#debug_level 3
source [find interface/stlink-v2.cfg]
source ./lpc4350.cfg
```

lpc4350.cfg

```
set _CHIPNAME lpc4350
set _M0_CPUTAPID 0x4ba00477
set _M4_SWDTAPID 0x2ba01477
set _M0_TAPID 0x0BA01477
set _TRANSPORT stlink_swd

transport select $_TRANSPORT

stlink newtap $_CHIPNAME m4 -expected-id $_M4_SWDTAPID
stlink newtap $_CHIPNAME m0 -expected-id $_M0_TAPID

target create $_CHIPNAME.m4 stm32_stlink -chain-position $_CHIPNAME.m4
#target create $_CHIPNAME.m0 stm32_stlink -chain-position $_CHIPNAME.m0
```

target.xml, nabbed from an OpenOCD mailing list thread, to fix a communication problem between GDB and newer OpenOCD builds.

```
<?xml version="1.0"?>
<!DOCTYPE target SYSTEM "gdb-target.dtd">
<target>
  <feature name="org.gnu.gdb.arm.core">
```

(continues on next page)

(continued from previous page)

```
    <reg name="r0" bitsize="32" type="uint32"/>
    <reg name="r1" bitsize="32" type="uint32"/>
    <reg name="r2" bitsize="32" type="uint32"/>
    <reg name="r3" bitsize="32" type="uint32"/>
    <reg name="r4" bitsize="32" type="uint32"/>
    <reg name="r5" bitsize="32" type="uint32"/>
    <reg name="r6" bitsize="32" type="uint32"/>
    <reg name="r7" bitsize="32" type="uint32"/>
    <reg name="r8" bitsize="32" type="uint32"/>
    <reg name="r9" bitsize="32" type="uint32"/>
    <reg name="r10" bitsize="32" type="uint32"/>
    <reg name="r11" bitsize="32" type="uint32"/>
    <reg name="r12" bitsize="32" type="uint32"/>
    <reg name="sp" bitsize="32" type="data_ptr"/>
    <reg name="lr" bitsize="32"/>
    <reg name="pc" bitsize="32" type="code_ptr"/>
    <reg name="cpsr" bitsize="32" regnum="25"/>
  </feature>
  <feature name="org.gnu.gdb.arm.fpa">
    <reg name="f0" bitsize="96" type="arm_fpa_ext" regnum="16"/>
    <reg name="f1" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f2" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f3" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f4" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f5" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f6" bitsize="96" type="arm_fpa_ext"/>
    <reg name="f7" bitsize="96" type="arm_fpa_ext"/>
    <reg name="fps" bitsize="32"/>
  </feature>
</target>
```

15.4 Run ARM GDB

Soon, I should dump this stuff into a .gdbinit file.

```
arm-none-eabi-gdb -n
target extended-remote localhost:3333
set tdesc filename target.xml
monitor reset init
monitor mww 0x40043100 0x10000000
monitor mdw 0x40043100   # Verify 0x0 shadow register is set properly.
file lpc4350-test.axf    # This is an ELF file.
load                     # Place image into RAM.
monitor reset init
break main               # Set a breakpoint.
continue                 # Run to breakpoint.
continue                 # To continue from the breakpoint.
step                     # Step-execute the next source line.
stepi                    # Step-execute the next processor instruction.
info reg                 # Show processor registers.
```

More GDB tips for the GDB-unfamiliar:

```
# Write the variable "buffer" (an array) to file "buffer.u8".
dump binary value buffer.u8 buffer

# Display the first 32 values in buffer whenever you halt
# execution.
display/32xh buffer

# Print the contents of a range of registers (in this case the
# CGU peripheral, starting at 0x40050014, for 46 words):
x/46 0x40050014
```

And still more, for debugging ARM Cortex-M4 Hard Faults:

```
# Assuming you have a hard-fault handler wired in:
display/8xw args

# Examine fault-related registers:

# Configurable Fault Status Register (CFSR) contains:
# CFSR[15:8]: BusFault Status Register (BFSR)
#   "Shows the status of bus errors resulting from instruction
#   prefetches and data accesses."
#   BFSR[7]: BFARVALID: BFSR contents valid.
#   BFSR[5]: LSPERR: fault during FP lazy state preservation.
#   BFSR[4]: STKERR: derived bus fault on exception entry.
#   BFSR[3]: UNSTKERR: derived bus fault on exception return.
#   BFSR[2]: IMPRECISERR: imprecise data access error.
#   BFSR[1]: PRECISERR: precise data access error, faulting
#            address in BFAR.
#   BFSR[0]: IBUSERR: bus fault on instruction prefetch. Occurs
#            only if instruction is issued.
display/xw 0xE000ED28

# BusFault Address Register (BFAR)
# "Shows the address associated with a precise data access fault."
# "This is the location addressed by an attempted data access that
# was faulted. The BFSR shows the reason for the fault and whether
# BFAR_ADDRESS is valid..."
# "For unaligned access faults, the value returned is the address
# requested by the instruction. This might not be the address that
# faulted."
display/xw 0xE000ED38
```

CHAPTER 16

LPC43xx SGPIO Configuration

The LPC43xx SGPIO peripheral is used to move samples between USB and the ADC/DAC chip (MAX5864). The SGPIO is a peripheral that has a bunch of 32-bit shift registers. These shift registers can be configured to act as a parallel interface of different widths. For HackRF, we configure the SGPIO to transfer eight bits at a time. The SGPIO interface can also accept an external clock, which we use to synchronize transfers with the sample clock.

In the current HackRF design, there is a CPLD which manages the interface between the MAX5864 and the SGPIO interface. There are four SGPIO signals that control the SGPIO data transfer:

- Clock: Determines when a value on the SGPIO data bus is transferred.
- Direction: Determines whether the MAX5864 DA (ADC) data is driven onto the SGPIO lines, or if the SGPIO lines drive the data bus with data for the MAX5864 DD (DAC) signals.
- Data Valid: Indicates a sample on the SGPIO data bus is valid data.
- Transfer Enable: Allows SGPIO to synchronize with the I/Q data stream. The MAX5864 produces/consumes two values (quadrature/complex value) per sample period – an I value and a Q value. These two values are multiplexed on the SGPIO lines. This signal suspends data valid until the I value should be transferred.

16.1 Frequently Asked Questions

16.1.1 Why not use GPDMA to transfer samples through SGPIO?

It would be great if we could, as that would free up lots of processor time. Unfortunately, the GPDMA scheme in the LPC43xx does not seem to support peripheral-to-memory and memory-to-peripheral transfers with the SGPIO peripheral.

You might observe that the SGPIO peripheral can generate requests from SGPIO14 and SGPIO15, using an arbitrary bit pattern in the slice shift register. The pattern in the slice determines the request interval. That's a good start. However, how do you specify which SGPIO shadow registers are read/written at each request, and in which order those registers are transferred with memory? It turns out you can't. In fact, it appears that an SGPIO request doesn't cause any transfer at all, if your source or destination is "peripheral". Instead, the SGPIO request is intended to perform a memory-to-memory transfer synchronized with SGPIO. But you're on your own as far as getting data to/from the

SGPIO shadow registers. I believe this is why the SGPIO camera example in the user manual describes an SGPIO interrupt doing the SGPIO shadow register transfer, and the GPDMA doing moves from one block of RAM to another.

Perhaps if we transfer only one SGPIO shadow register, using memory-to-memory? Then we don't have to worry about the order of SGPIO registers, or which ones need to be transferred. It turns out that when you switch over to memory-to-memory transfers, you lose peripheral request generation. So the GPDMA will transfer as fast as possible – far faster than words are produced/consumed by SGPIO.

I'd really love to be wrong about all this, but all my testing has indicated there's no workable solution to using GPDMA that's any better than using SGPIO interrupts to transfer samples. If you want some sample GPDMA code to experiment with, please contact Jared (sharebrained on #hackrf in Discord or IRC).

CHAPTER 17

List of Hardware Revisions

17.1 HackRF One r1–r4

The first revision of HackRF One shipped by Great Scott Gadgets starting in 2014 was labeled r1. Subsequent manufacturing runs incremented the revision number up to r4 without modification to the hardware design. Manufacturing years: 2014–2020

17.2 HackRF One r5

This experimental revision has not been manufactured.

17.3 HackRF One r6

SKY13350 RF switches were replaced by SKY13453 due to component availability. Although the SKY13453 uses simplified control logic, it did not require a firmware modification. Manufacturing year: 2020

17.4 HackRF One r7

SKY13453 RF switches were reverted to SKY13350 due to component availability. USB VBUS detection resistor values were changed to better protect the LPC4320. Manufacturing year: 2021

17.5 HackRF One r8

SKY13350 RF switches were replaced by SKY13453 due to component availability. Manufacturing years: 2021–2022

17.6 Hardware Revision Identification

HackRF Ones manufactured by Great Scott Gadgets have the revision number printed on the PCB top silkscreen layer near the MAX5864 (U18).

CHAPTER 18

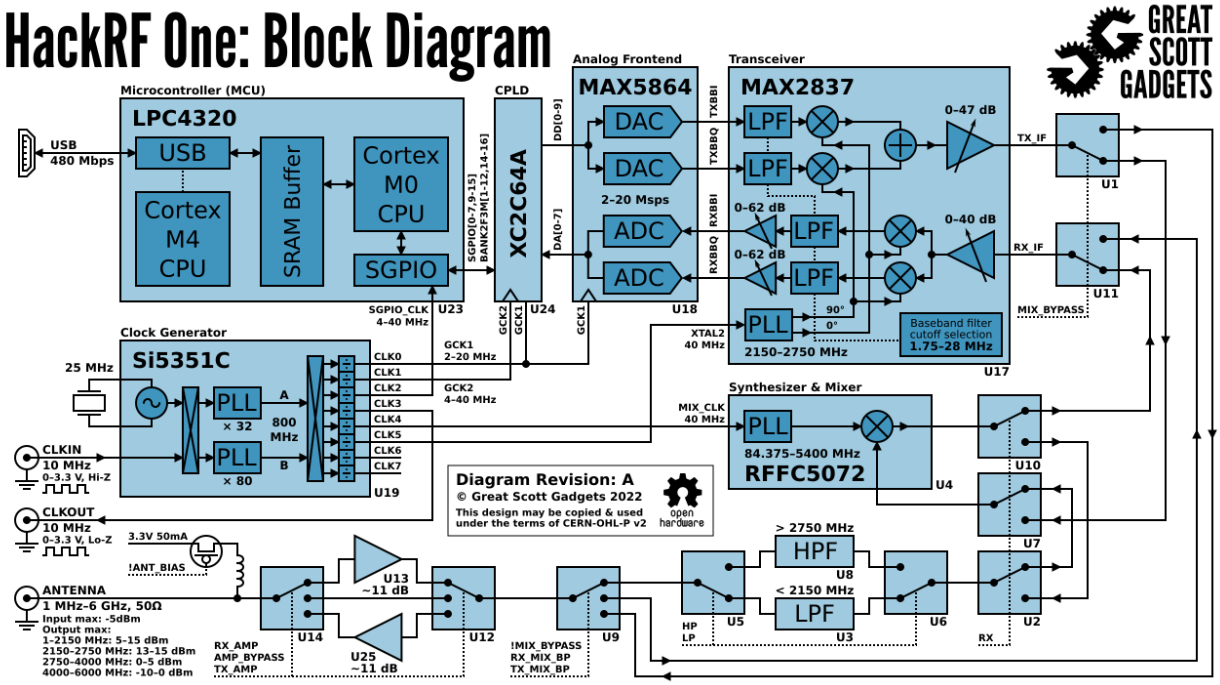
Hardware Components

Major parts used in HackRF One:

- **MAX2837 2.3 to 2.7 GHz transceiver**
  - Datasheet
  - There's also a register map document that Mike received directly from Maxim. Send an email to Mike or submit a support request to Maxim if you want a copy.
- **MAX5864 ADC/DAC**
  - Datasheet
- **Si5351 clock generator**
  - AN619: Manually Generating an Si5351 Register Map
  - Datasheet - this document is a mess of typos, and best used in conjunction with AN619, which has its own typos. Usually, you can reconcile what's true by comparison and a bit of thought.
  - Other Documentation - includes application notes, user guides, and white papers.
- CoolRunner-II CPLD
- **LPC43xx ARM Cortex-M4 microcontroller**
  - User Manual
  - Datasheet
  - Other Documentation (LPC4330FBD144) - includes errata and application notes.
  - ARM-standard JTAG/SWD connector pinout
  - BSDL file for the LPC43xx (For boundary scan)
- **RFFC5072 mixer/synthesizer**
  - Datasheet
  - Other Documentation ; click "Technical Documents" - includes programming guides and application notes.

- W25Q80BV 8M-bit Flash

18.1 Block Diagram



CHAPTER 19

Enclosure Options

The commercial version of HackRF One from Great Scott Gadgets ships with an injection molded plastic enclosure, but it is designed to fit two optional enclosures:

- Hammond 1455J1201: HackRF One fits this extruded aluminum enclosure and other similar models from Hammond Manufacturing. In order to use the enclosure's end plates, you will have to drill them. An end plate template can be found in the HackRF One KiCad layout.
- Acrylic sandwich: You can also use a laser cut acrylic enclosure with HackRF One. This is a good option for access to the expansion headers. A design can be found in the HackRF One hardware directory. Use any laser cutting service or purchase from a reseller.

CHAPTER 20

HackRF One's Buttons

The RESET button resets the microcontroller. This is a reboot that should result in a USB re-enumeration.

The DFU button invokes a USB DFU bootloader located in the microcontroller's ROM. This bootloader makes it possible to unbrick a HackRF One with damaged firmware because the ROM cannot be overwritten.

To invoke DFU mode: Press and hold the DFU button. While holding the DFU button, reset the HackRF One either by pressing and releasing the RESET button or by powering on the HackRF One. Release the DFU button.

The DFU button only invokes the bootloader during reset. This means that it can be used for other functions by custom firmware.

CHAPTER 21

External Clock Interface (CLKIN and CLKOUT)

HackRF One produces a 10 MHz clock signal on CLKOUT. The signal is a 10 MHz square wave from 0 V to 3 V intended for a high impedance load.

The CLKIN port on HackRF One is a high impedance input that expects a 0 V to 3 V square wave at 10 MHz. Do not exceed 3.3 V or drop below 0 V on this input. Do not connect a clock signal at a frequency other than 10 MHz (unless you modify the firmware to support this). You may directly connect the CLKOUT port of one HackRF One to the CLKIN port of another HackRF One.

HackRF One uses CLKIN instead of the internal crystal when a clock signal is detected on CLKIN. The switch to or from CLKIN only happens when a transmit or receive operation begins.

To verify that a signal has been detected on CLKIN, use `hackrf_debug --si5351c -n 0 -r`. The expected output with a clock detected is *[0] -> 0x01*. The expected output with no clock detected is *[0] -> 0x51*.

CHAPTER 22

Clocking Signals

HackRF clock signals are generated by the Si5351. The plan so far:

- crystal frequency: 25 MHz (supports 25 or 27 MHz)
- optional clock input frequency: 10 MHz recommended (supports 10 to 40 MHz, or higher with division)
- VCO frequency: 800 MHz (supports 600 to 900 MHz)
- MAX2837 clock: 40 MHz
- preferred MAX5864 clocks: 8, 10, 12.5, 16, 20 MHz
- A clock at double the MAX5864 rate will be delivered to the CPLD and SGPIO.
- LPC43xx clock: 12 MHz (from separate crystal so the ROM-based USB DFU will work)

Lemondrop+Jellybean Si5351 output mapping:

- CLK0 -> MAX2837
- CLK1 -> MAX5864/CPLD
- CLK2 -> CPLD
- CLK3 -> CPLD
- CLK4 -> LPC4330
- CLK5 -> RFFC5072
- CLK6 -> extra
- CLK7 -> extra

Jawbreaker output mapping:

- CLK0 -> MAX5864/CPLD
- CLK1 -> CPLD
- CLK2 -> SGPIO
- CLK3 -> external clock output

- CLK4 -> RFFC5072
- CLK5 -> MAX2837
- CLK6 -> none
- CLK7 -> LPC4330 (but LPC4330 will start up on its own crystal)

CHAPTER 23

Expansion Interface

The HackRF One expansion interface consists of headers P9, P20, P22, and P28. These four headers are installed on the commercial HackRF One from Great Scott Gadgets.

23.1 P9 Baseband

A direct analog interface to the high speed dual ADC and dual DAC.

Pin	Function
1	GND
2	GND
3	GND
4	RXBBQ-
5	RXBBI-
6	RXBBQ+
7	RXBBI+
8	GND
9	GND
10	TXBBI-
11	TXBBQ+
12	TXBBI+
13	TXBBQ-
14	GND
15	GND
16	GND

23.2 P20 GPIO

Providing access to GPIO, ADC, RTC, and power.

Pin	Function
1	VBAT
2	RTC_ALARM
3	VCC
4	WAKEUP
5	GPIO3_8
6	GPIO3_0
7	GPIO3_10
8	GPIO3_11
9	GPIO3_12
10	GPIO3_13
11	GPIO3_14
12	GPIO3_15
13	GND
14	ADC0_6
15	GND
16	ADC0_2
17	VBUSCTRL
18	ADC0_5
19	GND
20	ADC0_0
21	VBUS
22	VIN

23.3 P22 I2S

I2S, SPI, I2C, UART, GPIO, and clocks.

Pin	Function
1	CLKOUT
2	CLKIN
3	RESET
4	GND
5	I2C1_SCL
6	I2C1_SDA
7	SPIFI_MISO
8	SPIFI_SCK
9	SPIFI_MOSI
10	GND
11	VCC
12	I2S0_RX_SCK
13	I2S_RX_SDA
14	I2S0_RX_MCLK
15	I2S0_RX_WS
16	I2S0_TX_SCK
17	I2S0_TX_MCLK
18	GND
19	U0_RXD
20	U0_TXD
21	P2_9
22	P2_13
23	P2_8
24	SDA
25	CLK6
26	SCL

23.4 P28 SD

SDIO, GPIO, clocks, and CPLD.

Pin	Function
1	VCC
2	GND
3	SD_CD
4	SD_DAT3
5	SD_DAT2
6	SD_DAT1
7	SD_DAT0
8	SD_VOLT0
9	SD_CMD
10	SD_POW
11	SD_CLK
12	GND
13	GCK2
14	GCK1
15	B1AUX14
16	B1AUX13
17	CPLD_TCK
18	BANK2F3M2
19	CPLD_TDI
20	BANK2F3M6
21	BANK2F3M12
22	BANK2F3M4

Additional unpopulated headers and test points are available for test and development, but they may be incompatible with some enclosure or expansion options.

Refer to the schematics and component documentation for more information.

CHAPTER 24

Multiple Device Hardware Level Synchronization

24.1 Purpose

This page describes the modifications required to get multiple HackRF hardware-level synchronisation working. Synchronisation is required for many applications where a single HackRF isn't sufficient:

- phase correlation
- oversampling using multiple devices
- 40MHz (or more) protocols such as WiFi

The HackRFs will start transmitting USB packets at the same time, which results in an inter-device offset of ~50 samples at a sample rate of 20MSps. Without this synchronisation, the offset is in the range of thousands to tens of thousands of samples. This is due to the USB start command being called sequentially for each device, along with USB buffering, OS-level timing etc.

BE WARNED you will have to open your HackRFs, which is most likely going to destroy the plastic case it comes in. You will also be electrically connecting them together. If you do this incorrectly, there is a good chance one or all of the devices will be permanently destroyed.

24.2 Related work

"bardi_" on the #hackrf channel pointed out his paper on synchronising HackRFs. This uses the HackRF CPLD to synchronise multiple devices.

24.3 Requirements

For this to work you will need:

- at least two devices

- a clock sync cable
- some connecting cables (pin header-type)
- a breadboard

24.4 Opening your HackRF

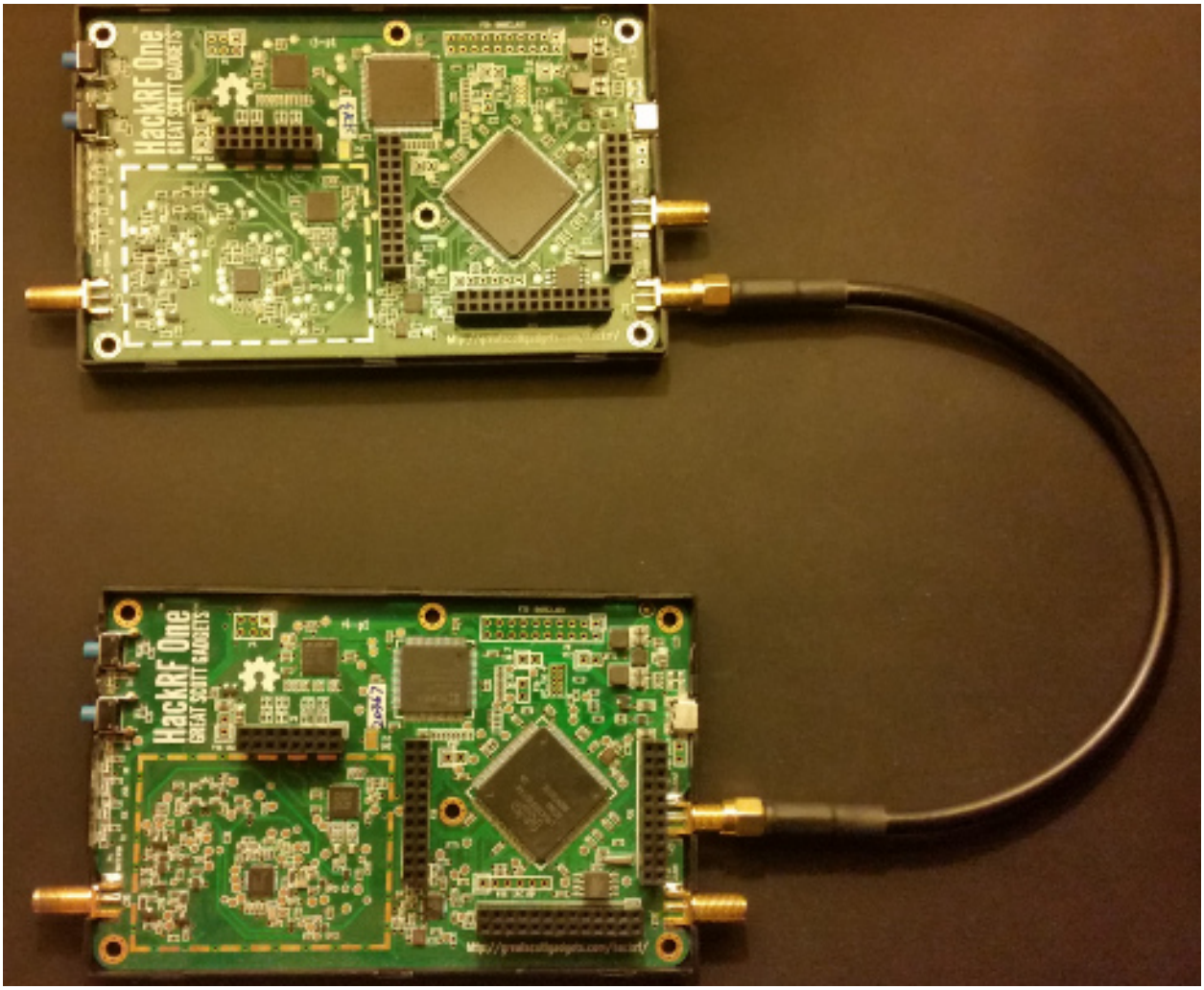
The HackRF case has small plastic clips holding it together. These are usually destroyed when the case is opened. Please follow the instructions in this video by Jared Boone.

24.5 Connect the clocks

Connecting the HackRF clocks together will force them to sample at precisely the same rate. The individual samples will most likely be sampled at slightly different times due to phase offsets in the clock ICs, but for most purposes this is acceptable.

Choose a **primary** HackRF, and connect the clock sync cable from the clock out connector to the clock in connector of the **second** HackRF. If you're using another HackRF, connect the second HackRF's clock out to the **third** HackRF's clock in.

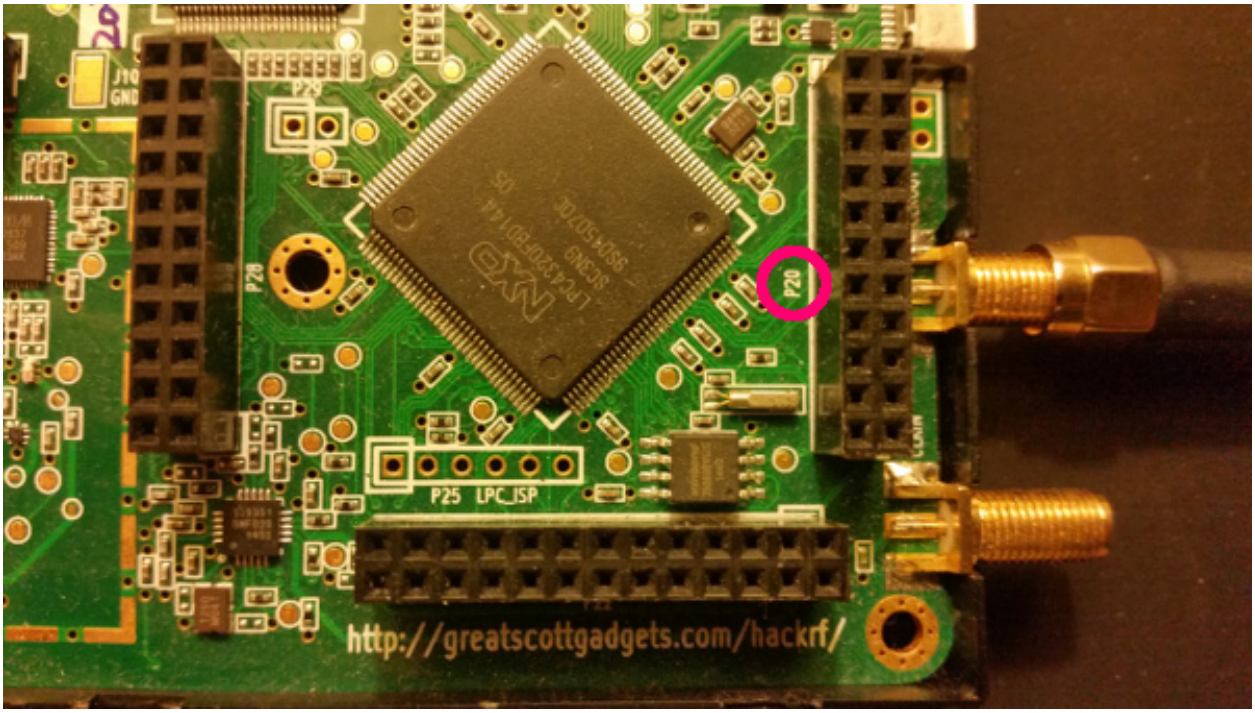
Your HackRFs should look like this:



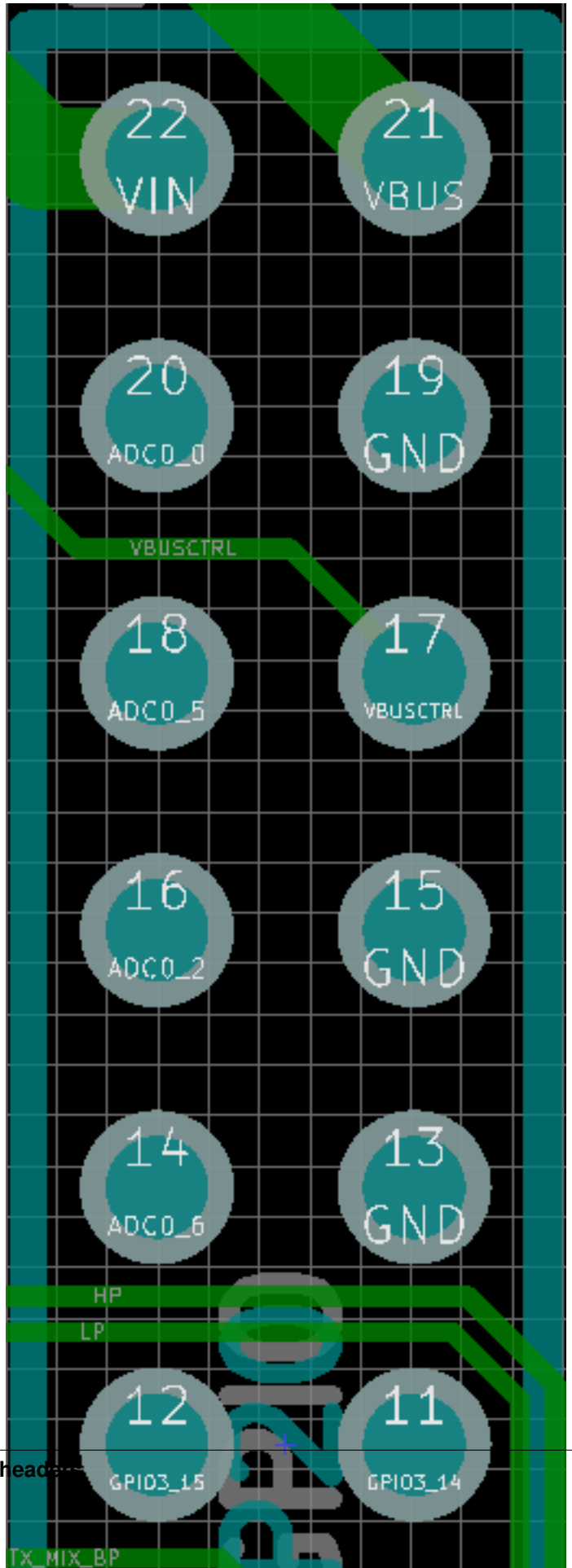
24.6 Identify the pin headers

Firstly, this has only been tested on official HackRF Ones. If you have a jawbreaker, HackRF blue or another HackRF-inspired device, you will have to figure out how to connect the devices correctly, using the schematics.

The hackrf has four pin headers, three of which are arranged in a 'C' shape. On the board these are marked as *P28*, *P22* and *P20*. *P20* is the header closest to the *clock in/clock out* connectors. For this exercise we will only be discussing *P20*. The hackrf schematics are a very good reference for this activity. The relevant part can been seen in the following image:



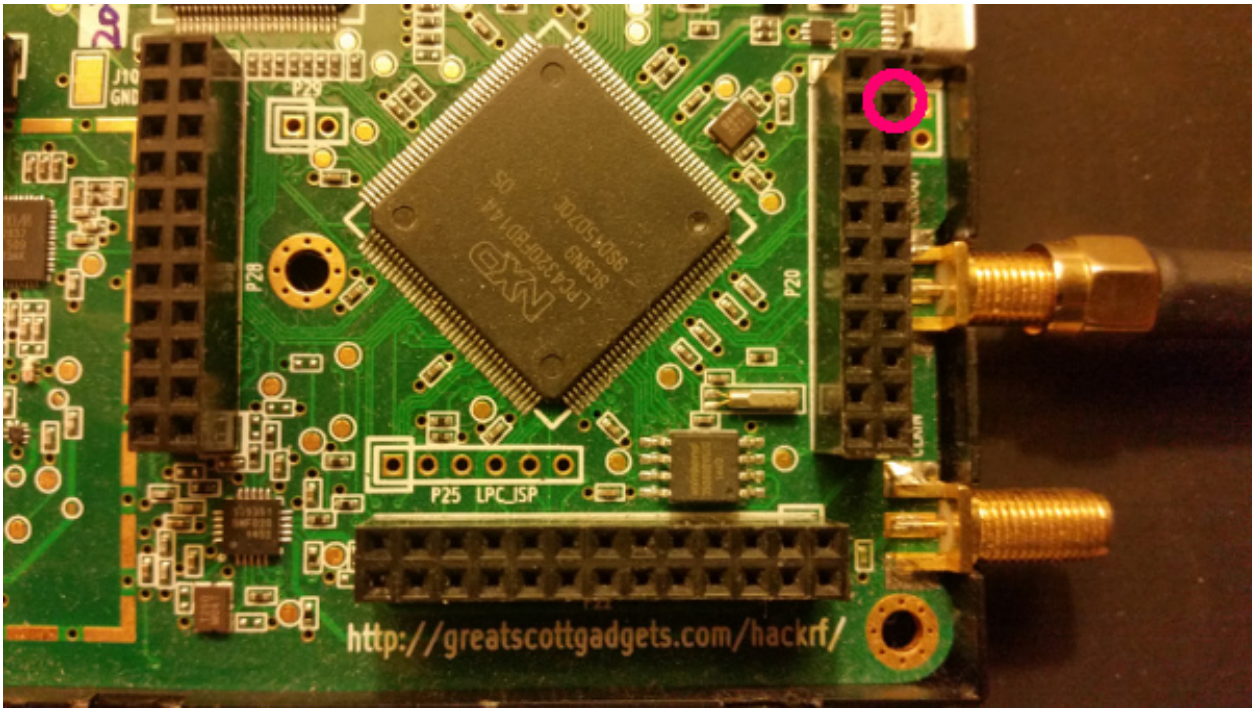
This is the P20 schematic diagram:

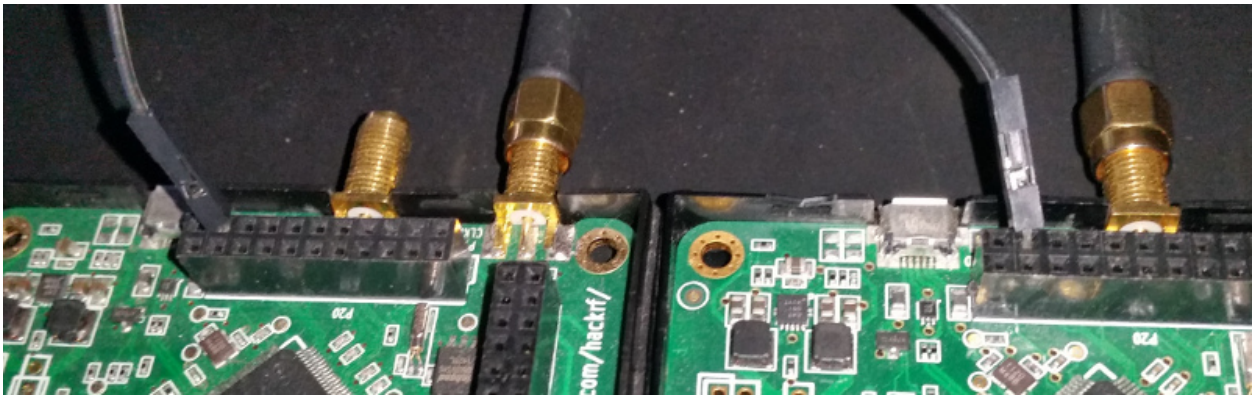

22
VIN
21
VBUS
20
ADC0_0
19
GND
VBUSCTRL
18
ADC0_5
17
VBUSCTRL
16
ADC0_2
15
GND
14
ADC0_6
13
GND
HP
LP
12
GPIO3_15
11
GPIO3_14
TX_MIX_BP

24.7 Wire up the pin headers

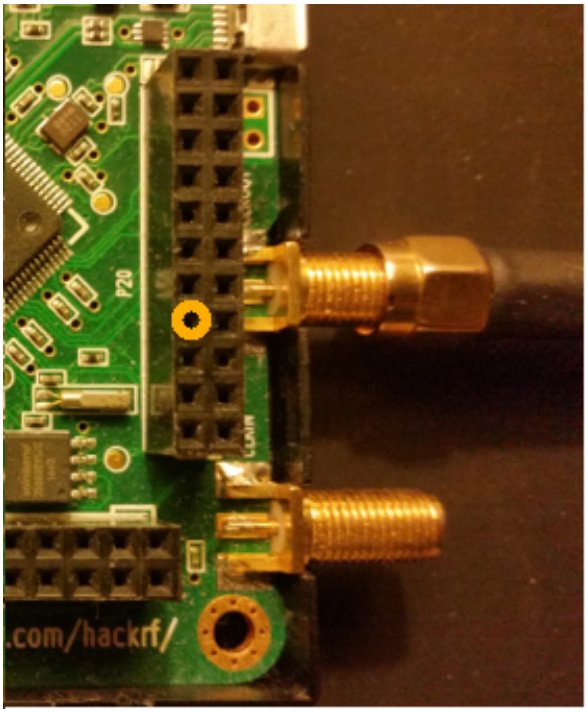
As mentioned before **BE WARNED**, this step could easily result in **one or all** of your HackRFs being **permanently damaged**.

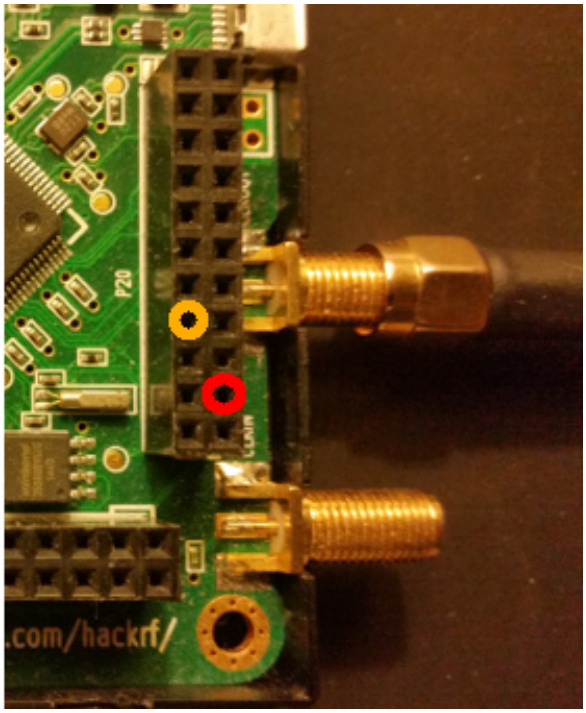
Now that's out of the way, let me describe what we're doing here. The first part of this exercise is to give both devices a common ground. This is really important for any inter-device electrical connections, as it prevents ICs from seeing slight differences in the respective GND levels as legitimate signals. As shown on the schematic, many of the pins in P20 are GND pins. We use P20-PIN19 on both devices and connect them together like so:

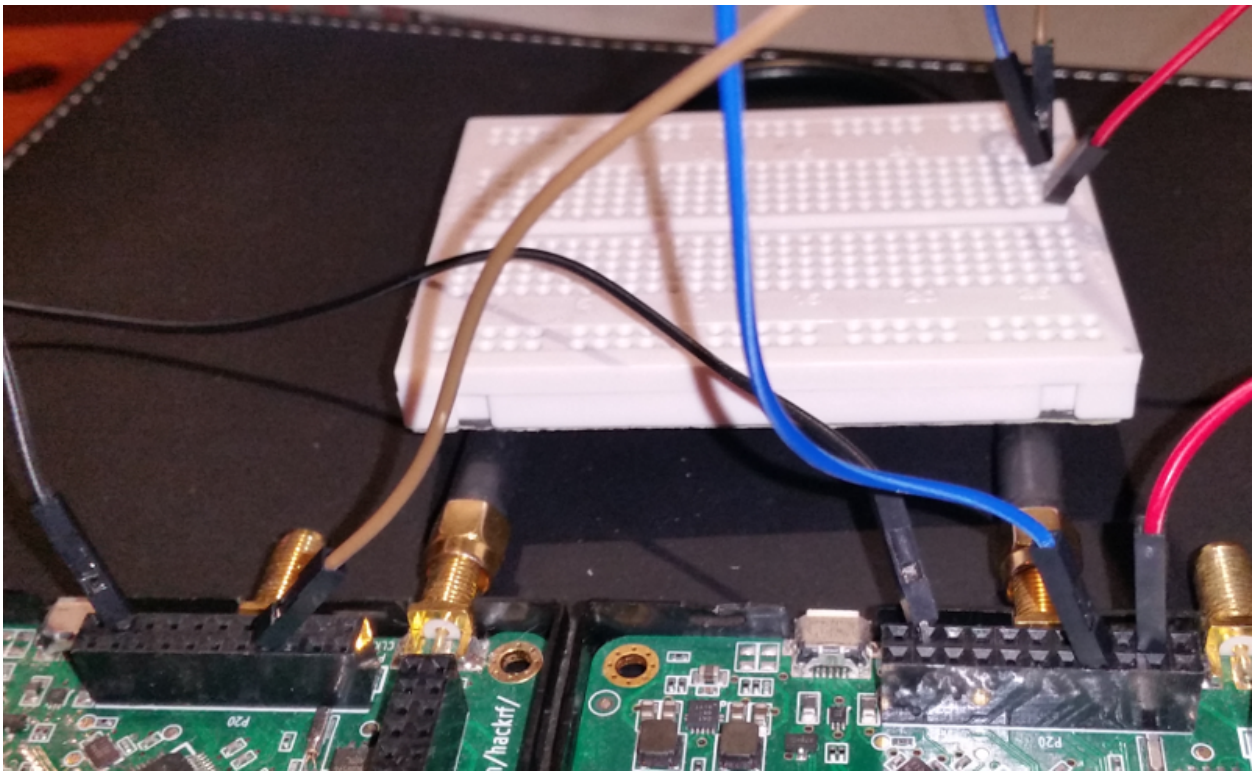




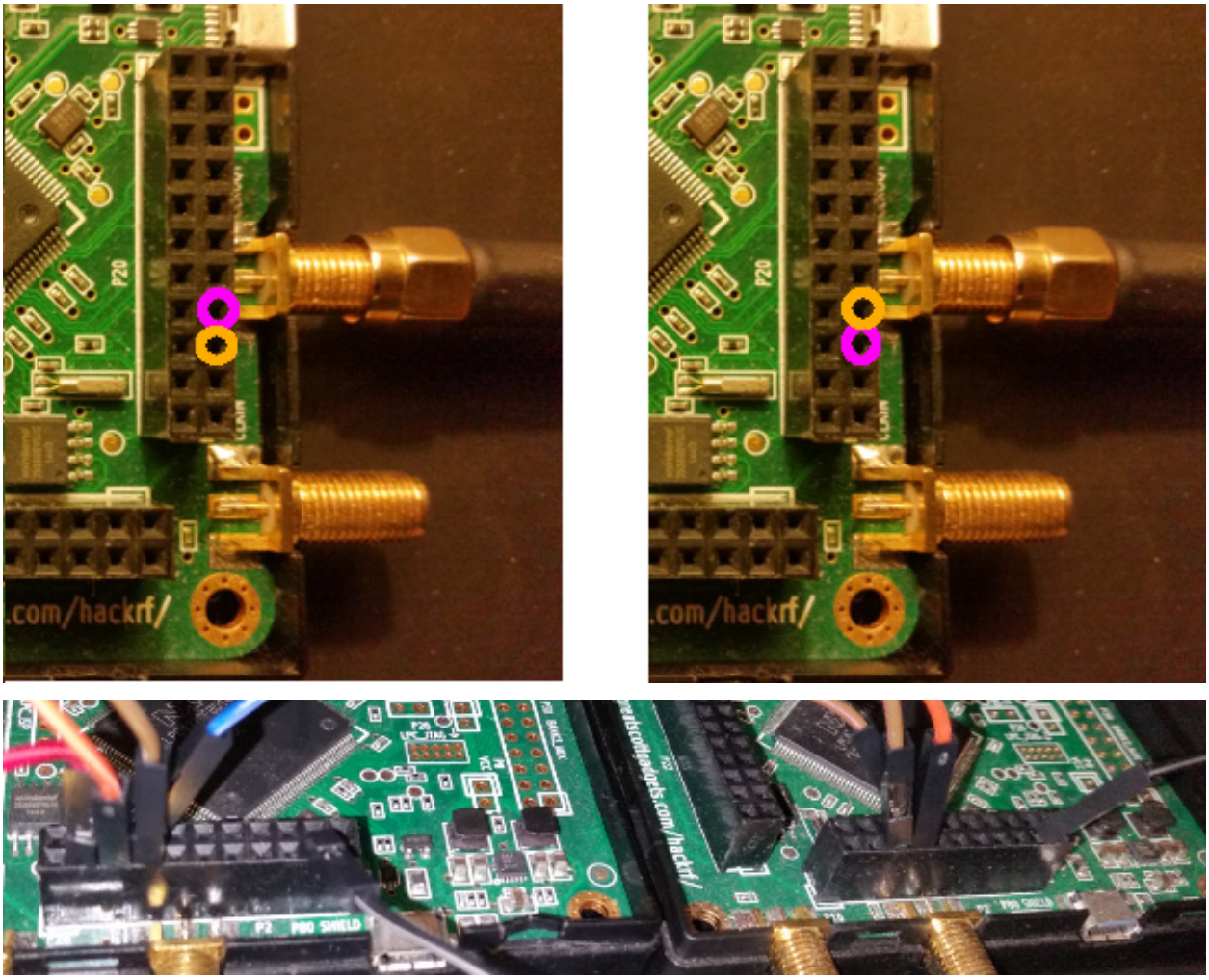
We then need a *positive* (+5v) connection to 'fake' the *third* hackrf if it's not present. We use *P20-PIN3* from the **primary** hackrf for this, and bring it down to the breadboard. *primary:P20-PIN8* and *secondary:P20-PIN8* are ready input GPIO pins for the missing third HackRF. Connect these to the breadboard *positive* line. After this your setup should look like so:







Next we connect the *primary:P20-PIN7 ready* GPIO pin input to the *secondary:P20-PIN5 ready* GPIO pin output, and the *primary:P20-PIN5 ack* GPIO pin output to the *secondary:P20-PIN7* ack GPIO pin input. This is the final step, and should look as follows:



24.8 Upgrade

Now that the hardware is setup, you need to upgrade your HackRFs' firmware, and your *libhackrf* to at least v2017.02.1 as per *this documentation page*.

24.9 Testing with hackrf_transfer

The latest version of *hackrf_transfer* includes the '-H' flag, which will activate hardware synchronisation (via libhackrf via the firmware). Testing this way is a little tricky because neither HackRF will start sending data until they are synched, and *hackrf_transfer* will time out if it hasn't received any data within one second. So the test requires that **two** copies of _hackrf_transfer are started within 1 second of each other. My approach is to have two terminal windows with the relevant commands waiting, and quickly run them.

This test will fail if:

- your hackrf firmware or libhackrf are out of date
- your connectors are incorrectly set up
- your timing is too slow when running hackrf_transfer

Run the following command:

- hackrf_transfer -d <device A> -r <filename-A> -H &; hackrf_transfer -d <device B> -r <filename-B> -H

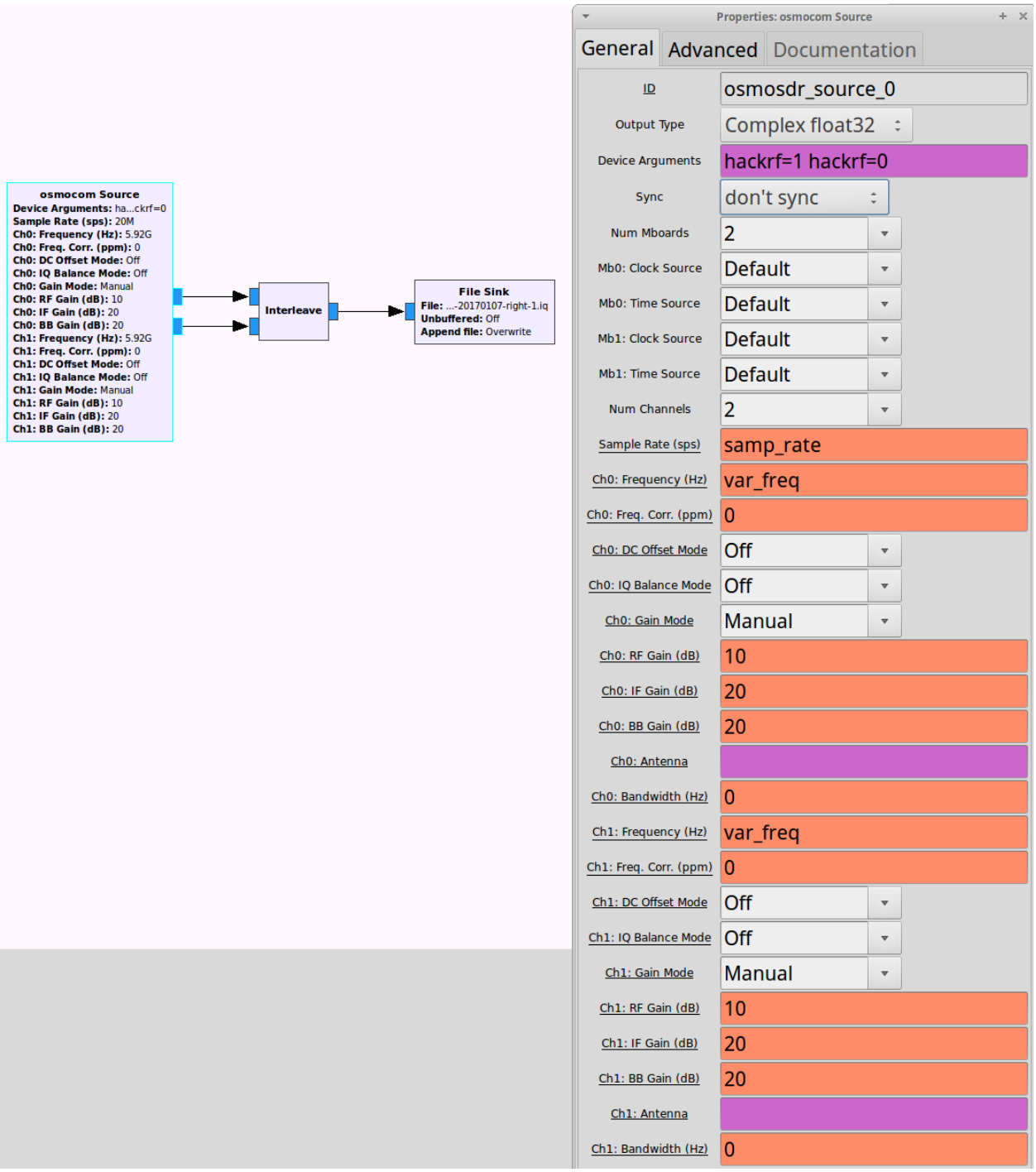
If the test runs correctly, you have successfully streamed synchronised data from two HackRFs!

The two streams can be merged into one using GnuRadio, and then viewed using this hacky piece of PyQt.

24.10 What next?

Obviously the method of wiring up multiple HackRFs described above is fragile and prone to error. Perhaps a PCB could be designed that will connect up to four HackRFs together by plugging into the 'C-shape' pin headers.

Usually the *Osmocom source* can be used for multi-device streaming, as it can be configured to pull from more than one device. Unfortunately the current version does not have hardware synchronisation built in. Work is being done to make the *Osmocom source* compatible with these changes.


osmocom Source
Device Arguments: ha...ckrf=0
Sample Rate (sps): 20M
Ch0: Frequency (Hz): 5.92G
Ch0: Freq. Corr. (ppm): 0
Ch0: DC Offset Mode: Off
Ch0: IQ Balance Mode: Off
Ch0: Gain Mode: Manual
Ch0: RF Gain (dB): 10
Ch0: IF Gain (dB): 20
Ch0: BB Gain (dB): 20
Ch1: Frequency (Hz): 5.92G
Ch1: Freq. Corr. (ppm): 0
Ch1: DC Offset Mode: Off
Ch1: IQ Balance Mode: Off
Ch1: Gain Mode: Manual
Ch1: RF Gain (dB): 10
Ch1: IF Gain (dB): 20
Ch1: BB Gain (dB): 20
Interleave
File Sink
File: ...-20170107-right-1.iq
Unbuffered: Off
Append file: Overwrite
Properties: osmocom Source
General
Advanced
Documentation
ID
osmosdr_source_0
Output Type
Complex float32
Device Arguments
hackrf=1 hackrf=0
Sync
don't sync
Num Mboards
2
Mb0: Clock Source
Default
Mb0: Time Source
Default
Mb1: Clock Source
Default
Mb1: Time Source
Default
Num Channels
2
Sample Rate (sps)
samp_rate
Ch0: Frequency (Hz)
var_freq
Ch0: Freq. Corr. (ppm)
0
Ch0: DC Offset Mode
Off
Ch0: IQ Balance Mode
Off
Ch0: Gain Mode
Manual
Ch0: RF Gain (dB)
10
Ch0: IF Gain (dB)
20
Ch0: BB Gain (dB)
20
Ch0: Antenna
Ch0: Bandwidth (Hz)
0
Ch1: Frequency (Hz)
var_freq
Ch1: Freq. Corr. (ppm)
0
Ch1: DC Offset Mode
Off
Ch1: IQ Balance Mode
Off
Ch1: Gain Mode
Manual
Ch1: RF Gain (dB)
10
Ch1: IF Gain (dB)
20
Ch1: BB Gain (dB)
20
Ch1: Antenna
Ch1: Bandwidth (Hz)
0

CHAPTER 25

Jawbreaker

HackRF Jawbreaker is the beta test hardware platform for the HackRF project.

25.1 Features

- half-duplex transceiver
- operating freq: 30 MHz to 6 GHz
- supported sample rates: 8 Msps to 20 Msps (quadrature)
- resolution: 8 bits
- interface: High Speed USB (with USB Micro-B connector)
- power supply: USB bus power
- portable
- open source

25.2 Set your Jawbreaker Free!

Jawbreaker has an SMA antenna connector but also includes a built-in PCB antenna intended for operation near 900 MHz. It isn't a very good antenna. Seriously. A paperclip stuck into the SMA connector would probably be better. You can free your Jawbreaker to operate with better antennas by cutting the PCB trace to the PCB antenna with a knife. This enables the SMA connector to be used without interference from the PCB antenna.

A video that demonstrates the antenna modification is on YouTube: HackRF Antenna Modification

The trace to be cut is between the two solder pads inside a box labeled R44 in the assembly diagram. There is an arrow pointing to it printed on the board.

Due to a manufacturing error, there is solder on R44. R44 may appear as a single solder blob. If you have a soldering iron and solder wick/braid, use a soldering iron and fine solder wick to remove as much solder as you can from the

two R44 pads. Then, use a pen knife to gently cut away the area between the two R44 pads. Make multiple, gentle cuts, instead of one or two forceful cuts. As you cut, you'll break through the black solder mask, then the copper trace between the pads, and stop when you reach fiberglass. Remove the copper trace completely, so just the two R44 pads remain. Use a multimeter or continuity tester to verify that the two R44 pads are no longer connected.

If you don't have a soldering iron, you can cut through the copper trace and the solder blob all at once, but it requires a bit more effort.

The only reason not to do this is if you want to try Jawbreaker but don't have any antenna with an SMA connector (or adapter).

If you want to restore the PCB antenna for some reason, you can install a 10 nF capacitor or a 0 ohm resistor on the R44 pads or you may be able to simply create a solder bridge.

25.3 SMA, not RP-SMA

Some connectors that appear to be SMA are actually RP-SMA. If you connect an RP-SMA antenna to Jawbreaker, it will seem to connect snugly but won't function at all because neither the male nor female side has a center pin. RP-SMA connectors are most common on 2.4 GHz antennas and are popular on Wi-Fi equipment.

25.4 Transmit Power

The maximum TX power varies by operating frequency:

- 30 MHz to 100 MHz: 5 dBm to 15 dBm, increasing as frequency decreases
- 100 MHz to 2300 MHz: 0 dBm to 10 dBm, increasing as frequency decreases
- 2300 MHz to 2700 MHz: 10 dBm to 15 dBm
- 2700 MHz to 4000 MHz: -5 dBm to 5 dBm, increasing as frequency decreases
- 4000 MHz to 6000 MHz: -15 dBm to 0 dBm, increasing as frequency decreases

Overall, the output power is enough to perform over-the-air experiments at close range or to drive an external amplifier. If you connect an external amplifier, you should also use an external bandpass filter for your operating frequency.

Before you transmit, know your laws. Jawbreaker has not been tested for compliance with regulations governing transmission of radio signals. You are responsible for using your Jawbreaker legally.

25.5 Hardware Documentation

Schematic diagram, assembly diagram,and bill of materials can be found at https://github.com/mossmann/hackrf/tree/master/doc/hardware

25.6 Expansion Interface

25.6.1 LPC

Boot config

Default boot configuration is SPIFI. Install headers and jumpers (and optionally resistors) to reconfigure.

Pin	P43	P32	P42	P27
1	VCC	VCC	VCC	VCC
2	P2_9	P2_8	P1_2	P1_1
3	GND	GND	GND	GND

The table below shows which pins to short per header for a given selection.

Selection	P43	P32	P42	P27
USART0	2-3	2-3	2-3	2-3
SPIFI	2-3	2-3	2-3	1-2
USB0	2-3	1-2	2-3	1-2
USSP0	2-3	1-2	1-2	1-2
USART3	1-2	2-3	2-3	2-3

P19 SPIFI Intercept header

Traces may be cut to install header and jumpers or use off-board SPI flash.

Pin	Function
1	Flash DO
2	SPIFI_MISO
3	Flash DI
4	SPIFI_MOSI
5	Flash CLK
6	SPIFI_SCK
7	Flash CS
8	SPIFI_CS
9	Flash Hold
10	SPIFI_SIO3
11	Flash WP
12	SPIFI_SIO2

P20 GPIO

Pin	Function
1	GPIO3_8
2	GPIO3_9
3	GPIO3_10
4	GPIO3_11
5	GPIO3_12
6	GPIO3_13
7	GPIO3_14
8	GPIO3_15
9	GND
10	GND

P21 Analog

Pin	Function
1	GND
2	ADC0_6
3	GND
4	ADC0_2
5	GND
6	ADC0_5
7	GND
8	ADC0_0

P22 I2S

Pin	Function
1	VCC
2	I2S0_TX_SDA
3	I2S0_TX_WS
4	I2S0_TX_SCK
5	I2S0_TX_MCLK
6	GND

P25 LPC_ISP

Pin	Function
1	GND
2	ISP
3	NC
4	U0_RXD
5	U0_TXD
6	RESET

P26 LPC_JTAG

Pin	Function
1	VCC
2	TMS
3	GND
4	TCK
5	GND
6	TDO
7	NC
8	TDI
9	GND
10	RESET

P28 SD

Pin	Function
1	GND
2	VCC
3	SD_CD
4	SD_DAT3
5	SD_DAT2
6	SD_DAT1
7	SD_DAT0
8	SD_VOLT0
9	SD_CMD
10	SD_POW
11	SD_CLK
12	NC

25.6.2 CPLD

P29 CPLD_JTAG

Pin	Function
1	CPLD_TMS
2	CPLD_TDI
3	CPLD_TDO
4	CPLD_TCK
5	GND
6	NCC

P30 BANK2_AUX

Pin	Function
1	B2AUX1
2	B2AUX2
3	B2AUX3
4	B2AUX4
5	B2AUX5
6	B2AUX6
7	B2AUX7
8	B2AUX8
9	B2AUX9
10	B2AUX10
11	B2AUX11
12	B2AUX12
13	B2AUX13
14	B2AUX14
15	B2AUX15
16	B2AUX16

P31 BANK1_AUX

Pin	Function
1	B1AUX9
2	B1AUX10
3	B1AUX11
4	B1AUX12
5	B1AUX13
6	B1AUX14
7	B1AUX15
8	B1AUX16
9	GND
10	GND

25.6.3 External clock

P2 CLKOUT

Install C165 and R92 as necessary to match output. For CMOS output, install 0 ohm resistor in place of C165; do not install R92.

Pin	Function
1	CLKOUT
2	GND
3	GND
4	GND
5	GND

P16 CLKIN

Install C118, C164, R45, R84 and R85 as necessary to match input.

For CMOS input, install 0 ohm resistors in place of C118 and C164; do not install R45, R84, or R85.

Pin	Function
1	CLKIN
2	GND
3	GND
4	GND
5	GND

P17 CLKIN_JMP

Cut P17 short (trace) to enable external clock input. If short is cut, a jumper should be used on P17 at all times when an external clock is not connected to P16.

Pin	Function
1	GND
2	CLKIN

25.6.4 More

Additional headers are available. See the board files for additional details.

25.7 Differences between Jawbreaker and HackRF One

Jawbreaker was the beta platform that preceded HackRF One. HackRF One incorporates the following changes and enhancements:

- Antenna port: No modification is necessary to use the SMA antenna port on HackRF One.
- PCB antenna: Removed.
- Size: HackRF One is smaller at 120 mm x 75 mm (PCB size).
- Enclosure: The commercial version of HackRF One from Great Scott Gadgets ships with an injection molded plastic enclosure. HackRF One is also designed to fit other enclosure options.
- Buttons: HackRF One has a RESET button and a DFU button for easy programming.
- Clock input and output: Installed and functional without modification.
- USB connector: HackRF One features a new USB connector and improved USB layout.
- Expansion interface: More pins are available for expansion, and pin headers are installed on HackRF One.
- Real-Time Clock: An RTC is installed on HackRF One.
- LPC4320 microcontroller: Jawbreaker had an LPC4330.
- RF shield footprint: An optional shield may be installed over HackRF One's RF section.
- Antenna port power: HackRF One can supply up to 50 mA at 3.3 V DC on the antenna port for compatibility with powered antennas and other low power amplifiers.
- Enhanced frequency range: The RF performance of HackRF One is better than Jawbreaker, particularly at the high and low ends of the operating frequency range. HackRF One can operate at 1 MHz or even lower.

CHAPTER 26

Design Goals

Eventually, the HackRF project may result in multiple hardware designs, but the initial goal is to build a single wideband transceiver peripheral that can be attached to a general purpose computer for software radio functions.

Primary goals:

- half-duplex transceiver
- operating freq: 100 MHz to 6 GHz
- maximum sample rate: 20 Msps
- resolution: 8 bits
- interface: High Speed USB
- power supply: USB bus power
- portable
- open source

Wish list:

- full-duplex (at reduced max sample rate)
- external clock reference
- dithering
- parallel interface for external FPGA, etc.

If there is a primary goal we miss, it will probably be the operating frequency range. The wideband front end is the part of the design furthest from completion. At an absolute minimum, the board should do 900 MHz and 2.4 GHz.

The design is FPGA-less. There will be a tiny bit of DSP capability (ARM Cortex-M4), but mostly we're just trying to get samples to and from a host computer.

We are trading resolution and DSP capability for cost, portability, and frequency range. Considering that we'll be able to support oversampling for many applications and that we should be able to implement AGC, it should be a pretty good trade.

CHAPTER 27

Future Hardware Modifications

Things to consider for post-Jawbreaker hardware designs:

27.1 Antenna

The PCB antenna on Jawbreaker was included to facilitate beta testing. Future designs likely will not include a PCB antenna.

SMA connectors will be PCB edge-mounted.

27.2 Baseband

The interfaces between the MAX2837 and MAX5864 have some signals inverted. Theoretically, that's fine if compensated for in software. However, I'm theorizing that RX/ADC DC offset compensation assumes that both channels have the same DC polarity. I've fixed the inversion in the CPLD. However, a PCB experiment should be conducted to see if the DC offset is reduced by un-inverting the RX Q channel connections to the MAX5864.

27.3 CPLD

The CPLD could be removed, but some sort of multiplexer would be needed to meet the MAX5864 i/o requirements. Depending on the particular LPC43xx part used, it might be possible to use the System Control Unit (SCU) for this.

27.4 Clocking

The clock signal from the Si5351C to the LPC43xx's GP_CLKIN pin may need different passives, but the documentation on that clock input is thin (acceptable peak-to-peak voltage anyone?).

An unpopulated footprint for a 32.768 kHz RTC crystal would be nice. Also break out RTC battery pins to an expansion header.

27.5 USB

Would support for host mode on the second USB PHY be useful somehow? This is only possible with a larger LPC43xx package that exposes the second PHY's ULPI signals. Unless, of course, a mere full-speed PHY is acceptable.

27.6 Power Management

The MAX5864 appears to come up in "Tx" or "Rcvr" mode – I have observed that the part will pass DA bus data to ID/QD without any SPI configuration. If we're worried about USB power and minimizing current consumption, it might be good to have this device on a power regulator with an ENABLE pin, or have a FET power switch. Yes, let's add a high side switch for the whole RF section.

27.7 Regulators

U21 (the TPS62410) FB1 pin is connected on the far side of jumper P8 (VCC), which puts the jumper inside the feedback path. If the jumper trace is cut, the regulator may go nuts because the FB pin is floating.

27.8 Buttons

Add a reset button (for the LPC43xx). Maybe add a DFU button too.

27.9 Shielding

Maybe add a can around the RF section.

27.10 Footprints

Tighten up holes for USB connector support legs to improve placement consistency. Make some of the QFN pads bigger (especially on the RF switches) for better soldering.

27.11 Shield Support

If support for add-on shields is considered valuable, here are some tweaks I'd suggest:

Any reason P28 (SD) pin 12 isn't grounded or doing something useful? Same goes for P25 (LPC_ISP) pin 3 – maybe make it VCC, the signaling voltage for the ISP interface? The SPIFI connector could also use a reference voltage (GND?).

I'd like to see an I2C bus exposed somewhere, and perhaps an I2S0_RX_SDA signal, so I don't have to steal it from the CPLD interface. The I2S0 will function in "four-wire mode" with only one more pin (RX_SDA), so why not?

Provide a way to inject a supply voltage into the board? Having diodes managing multiple voltage sources would be lossy, so a more expensive solution would be necessary on the Jawbreaker board, adding cost.

If an LPC43xx package with a higher pin-count is used, it would be stellar to expose the LCD interface and quadrature encoder peripheral pins.

The RTC would be handy for stand-alone use. This would require a crystal (32.768kHz) between RTCX1 and RTCX2, and exposing VBAT to a shield for battery backup (disconnecting it from VCC) or providing a coin cell footprint on the HackRF PCB.

Coalesce separate headers into fewer, larger banks of headers, to reduce the number of unique, small header receptacles required for mating? Reducing the header count will also increase the amount of board space around the perimeter of a shield for components and connectors.

CHAPTER 28

Lemondrop Bring Up

Board draws approximately 24mA from +3V3 when power is applied. This seems a bit high, but may be expected if not all parts are capable of low-power mode, or aren't configured for low power at power-on. I need to review the schematic and datasheets and see what can be done.

When I put my finger on the MAX2837, current consumption goes up. This suggests there may be floating nodes in that region of the circuit.

28.1 Si5351 I2C

Attached crystal is 25MHz. For now, I'm assuming 10pF "internal load capacitance" is good enough to get the crystal oscillating. The crystal datasheet should be reviewed and measurements made. . .

Be sure to reference Silicon Labs application note 619 (AN619). The datasheet is a terrible mess (typos and lack of some details). AN619 appears to be less of a mess, on the whole. And as a bonus, AN619 has PDF bookmarks for each register.

28.1.1 Connections

- Bus Pirate GND to P7 pin 1
- Bus Pirate +3V3 to P7 pin 2 (through multimeter set to 200mA range)
- Bus Pirate CLK to P7 pin 3
- Bus Pirate MOSI to P7 pin 5

28.1.2 Bus Pirate I2C Initialization

```
# set mode
m
# I2C mode
4
# ~100kHz speed
3
# power supplies ON
W
# macro 1: 7-bit address search
(1)
Searching I2C address space. Found devices at:
0xC0(0x60 W) 0xC1(0x60 R)
```

I2C A0 address configuration pin (not available on QFN20 package) is apparently forced to "0".

28.1.3 Reading registers

```
# Read register 0
I2C>[0xc0 0[0xc1 r]]
...
# Register 0: SYS_INIT=0, LOL_B=0, LOL_A=0, LOS=1, REVID=0
READ: 0x10
...
# Read 16 registers, starting with register 0
I2C>[0xc0 0[0xc1 r:16]]
...
READ: 0x10  ACK 0xF8  ACK 0x03  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x8F  ACK 0x01  ACK
      0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x90  ACK 0x00
...
# Read 16 registers, starting with register 16
I2C>[0xc0 16[0xc1 r:16]]
...
READ: 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK
      0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00  ACK 0x00
...
```

28.1.4 Writing registers

Simple XTAL passthrough to CLK0

```
# Disable all CLKx outputs.
[0xC0 3 0xFF]

# Turn off OEB pin control for all CLKx
[0xC0 9 0xFF]

# Power down all CLKx
[0xC0 16 0x80 0x80 0x80 0x80 0x80 0x80 0x80 0x80]

# Register 183: Crystal Internal Load Capacitance
# Reads as 0xE4 on power-up
# Set to 10pF (until I find out what loading the crystal/PCB likes best)
[0xC0 183 0xE4]
```

(continues on next page)

(continued from previous page)

```
# Register 187: Fanout Enable
# Turn on XO fanout only.
[0xC0 187 0x40]

# Register 15: PLL Input Source
# CLKIN_DIV=0 (Divide by 1)
# PLLB_SRC=0 (XTAL input)
# PLLA_SRC=0 (XTAL input)
[0xC0 15 0x00]

# Registers 16 through 23: CLKx Control
# CLK0:
#   CLK0_PDN=0 (powered up)
#   MS0_INT=1 (integer mode)
#   MS0_SRC=0 (PLLA as source for MultiSynth 0)
#   CLK0_INV=0 (not inverted)
#   CLK0_SRC=0 (XTAL as clock source for CLK0)
#   CLK0_IDRV=3 (8mA)
[0xC0 16 0x43 0x80 0x80 0x80 0x80 0x80 0x80 0x80]

# Enable CLK0 output only.
[0xC0 3 0xFE]
```

Clocking Scheme (Work In Progress)

From AN619: If Fxtal=25MHz, Fvco = Fxtal * (a + (b / c)). If we want Fvco = 800MHz, a = 32, b = 0, c = don't care.

```
MSNA_P1[17:0] = 128 * a + floor(128 * b / c) - 512
              = 128 * a + floor(0) - 512
              = 128 * 32 + 0 - 512
              = 3584 = 0xE00
MSNA_P1[17:16] (register 28) = 0x00
MSNA_P1[15: 8] (register 29) = 0x0E
MSNA_P1[ 7: 0] (register 30) = 0x00
MSNA_P2[19:0] = 128 * b - c * floor(128 * b / c)
              = 128 * 0 - 0 * floor(128 * 0 / X)
              = 0
MSNA_P3[19:0] = 0
```

MultiSynth0 should output 40MHz (800MHz VCO divided by 20):

```
a = 20, b = 0, c = X
MS0_P1[17: 0] = 128 * a + floor(128 * b / c) - 512
              = 2048 = 0x800
MS0_P1[17:16] (register 44) = 0x00
MS0_P1[15: 8] (register 45) = 0x08
MS0_P1[ 7: 0] (register 46) = 0x00
MS0_P2[19:0]  = 0
MS0_P3[19:0]  = 0
```

MultiSynth1 should output 20MHz (800MHz VCO divided by 40) or some smaller integer fraction of the VCO:

```
a = 40, b = 0, c = X
MS1_P1[17: 0] = 128 * a + floor(128 * b / c) - 512
```

(continues on next page)

(continued from previous page)

```
              = 4608 = 0x1200
MS1_P1[17:16] (register 52) = 0x00
MS1_P1[15: 8] (register 53) = 0x12
MS1_P1[ 7: 0] (register 54) = 0x00
MS1_P2[19:0]  = 0
MS1_P3[19:0]  = 0
```

Initialization:

```
# Disable all CLKx outputs.
[0xC0 3 0xFF]

# Turn off OEB pin control for all CLKx
[0xC0 9 0xFF]

# Power down all CLKx
[0xC0 16 0x80 0x80 0x80 0x80 0x80 0x80 0x80 0x80]

# Register 183: Crystal Internal Load Capacitance
# Reads as 0xE4 on power-up
# Set to 10pF (until I find out what loading the crystal/PCB likes best)
[0xC0 183 0xE4]

# Register 187: Fanout Enable
# Turn on XO and MultiSynth fanout only.
[0xC0 187 0x50]

# Register 15: PLL Input Source
# CLKIN_DIV=0 (Divide by 1)
# PLLB_SRC=0 (XTAL input)
# PLLA_SRC=0 (XTAL input)
[0xC0 15 0x00]

# MultiSynth NA (PLL1)
[0xC0 26 0x00 0x00 0x00 0x0E 0x00 0x00 0x00 0x00]

# MultiSynth NB (PLL2)
...

# MultiSynth 0
[0xC0 42 0x00 0x00 0x00 0x08 0x00 0x00 0x00 0x00]

# MultiSynth 1
[0xC0 50 0x00 0x00 0x00 0x12 0x00 0x00 0x00 0x00]

# Registers 16 through 23: CLKx Control
# CLK0:
#   CLK0_PDN=0 (powered up)
#   MS0_INT=1 (integer mode)
#   MS0_SRC=0 (PLLA as source for MultiSynth 0)
#   CLK0_INV=0 (not inverted)
#   CLK0_SRC=3 (MS0 as input source)
#   CLK0_IDRV=3 (8mA)
# CLK1:
#   CLK1_PDN=0 (powered up)
#   MS1_INT=1 (integer mode)
#   MS1_SRC=0 (PLLA as source for MultiSynth 1)
```

(continues on next page)

(continued from previous page)

```
#   CLK1_INV=0 (not inverted)
#   CLK1_SRC=3 (MS1 as input source)
#   CLK1_IDRV=3 (8mA)
[0xC0 16 0x4F 0x4F 0x80 0x80 0x80 0x80 0x80 0x80]

# Enable CLK0 output only.
[0xC0 3 0xFC]
```

28.1.5 Si5351 output phase relationships

Tested CLK4 and CLK5 (integer division only):

With CLK4 set to MS4 and CLK5 set to MS5, even with both multisynths configured identically, there was no consistent phase between the two. Once started, the clocks maintained relative phase with each other, but when stopped and restarted the initial phase offset was unpredictable.

With CLK4 and CLK5 both set to MS4, the phase of both outputs was identical when no output (R) divider was selected. When an output divider was selected on both MS4 and MS5, the relative phase became predictable only within the constraints of the divider (e.g. with R=2 the relative phase was always either 0 or half a cycle, with R=4 the relative phase was always either 0, a quarter, a half, or three quarters of a cycle). With R=1 on MS4 and R=2 on MS5, the two outputs were consistently in phase with each other. The output (R) dividers supposedly tied to the multisynths are actually tied to the outputs.

CHAPTER 29

LPC4350 SGPIO Experimentation

The NXP LPC43xx microcontrollers have an interesting, programmable serial peripheral called the SGPIO (Serial GPIO). It consists of a slew of counters and shift registers that can be configured to serialize and deserialize many channels of data. Channels can be grouped to create multi-bit parallel data streams.

The current HackRF design entails using the SGPIO peripheral to move quadrature baseband receive and transmit data between the USB interface and the baseband ADC/DAC IC. Because the baseband ADC/DAC IC (MAX5864) uses DDR signaling, we expect to use a CPLD to convert bus signaling. The CPLD may also help manage bus turnaround (between transmit and receive modes) or interfacing two narrower but faster interfaces to the LPC43xx to facilitate full-duplex.

Because the Jellybean board wasn't completed at the time of these experiments, I used the Diolan LPC-4350-DB1-A development board. Despite using an LPC4350 in an BGA256 package, the SGPIO peripheral's signals can be mapped to many different pins. So reworking code to a new set of SGPIO pins should be a trivial matter of switching the SGU configuration for the affected pins.

29.1 SGPIO Examples

Some SGPIO peripheral examples can be found in the LPCWare repository. All source I've found so far is focused on generating many I2S interfaces, which is not very similar to HackRF's needs. But reviewing the code is still valuable in grasping how the SGPIO peripheral operates.

There are a few common details to setting up the SGPIO peripheral:

```
// Configure the PLL to generate a reasonable clock. The SGPIO
// will operate with a clock of up to 204MHz (the same as the
// M4 clock.
CGU_SetPLL1(12);

// Set the BASE_PERIPH clock to come from PLL1.
CGU_EnableEntity(CGU_BASE_PERIPH, ENABLE);
CGU_EntityConnect(CGU_CLKSRC_PLL1, CGU_BASE_PERIPH);
```

(continues on next page)

(continued from previous page)

```
// Compute and commit clock configuration changes.
CGU_UpdateClock();
```

29.1.1 Jiggle SGPIO Pins From GPIO Mode

My first test was to ensure I had the right pin(s) hooked up to my scope:

```
// Jiggle one of the SGPIO pins in GPIO mode, to make sure
// I'm looking at the right pin on the scope.
scu_pinmux(9,  0, MD_PLN_FAST, 0);

GPIO_SetDir(4, 1L << 12, 1);

while(1) {
    volatile int i;
    GPIO_SetValue(4, 1L << 12);
    for(i=0; i<1000; i++);
    GPIO_ClearValue(4, 1L << 12);
    for(i=0; i<1000; i++);
}
```

29.1.2 Jiggle Pins from SGPIO Mode

You can also control SGPIO pins, GPIO-style, from within the SGPIO peripheral. This helped me understand the basics of operating the SGPIO output mux.

```
// Set pin to SGPIO mode, toggle output using SGPIO
// peripheral registers.
scu_pinmux(9,  0, MD_PLN_FAST, 6);  // SGPIO0

// P_OUT_CFG = 4, gpio_out
// P_OE_CFG = X
LPC_SGPIO->OUT_MUX_CFG[0] = (0L << 4) | (4L << 0);
LPC_SGPIO->GPIO_OENREG |= (1L << 0);

while(1) {
    volatile int i;
    LPC_SGPIO->GPIO_OUTREG |= (1L << 0);
    for(i=0; i<1000; i++);
    LPC_SGPIO->GPIO_OUTREG &= ~(1L << 0);
    for(i=0; i<1000; i++);
}
```

29.1.3 Serializing Data With Slice Clock Source

My first full-on SGPIO experiment involved serializing a data pattern from slice A, using slice D to generate a SGPIO_CLK/2 data rate. I derived the code from examples that configured the SGPIO as I2S interfaces:

```
// Disable all counters during configuration
LPC_SGPIO->CTRL_ENABLED = 0;
```

(continues on next page)

(continued from previous page)

```
// Configure pin functions.
scu_pinmux(9,  0, MD_PLN_FAST, 6);  // SGPIO0
scu_pinmux(2,  3, MD_PLN_FAST, 0);  // SGPIO12

// Enable SGPIO pin outputs.
LPC_SGPIO->GPIO_OENREG =
    (1L << 12) |    // SGPIO12
    (1L <<  0);     // SGPIO0

// SGPIO pin 0 outputs slice A bit 0.
LPC_SGPIO->OUT_MUX_CFG[0] =
    (0L <<  4) |    // P_OE_CFG = X
    (0L <<  0);     // P_OUT_CFG = 0, dout_doutm1 (1-bit mode)

// SGPIO pin 12 outputs slice D bit 0.
LPC_SGPIO->OUT_MUX_CFG[12] =
    (0L <<  4) |    // P_OE_CFG = X
    (0L <<  0);     // P_OUT_CFG = 0, dout_doutm1 (1-bit mode)

// Slice A
LPC_SGPIO->SGPIO_MUX_CFG[0] =
    (0L << 12) |    // CONCAT_ORDER = 0 (self-loop)
    (1L << 11) |    // CONCAT_ENABLE = 1 (concatenate data)
    (0L <<  9) |    // QUALIFIER_SLICE_MODE = X
    (0L <<  7) |    // QUALIFIER_PIN_MODE = X
    (0L <<  5) |    // QUALIFIER_MODE = 0 (enable)
    (0L <<  3) |    // CLK_SOURCE_SLICE_MODE = 0, slice D
    (0L <<  1) |    // CLK_SOURCE_PIN_MODE = X
    (0L <<  0);     // EXT_CLK_ENABLE = 0, internal clock signal (slice)

LPC_SGPIO->SLICE_MUX_CFG[0] =
    (0L <<  8) |    // INV_QUALIFIER = 0 (use normal qualifier)
    (0L <<  6) |    // PARALLEL_MODE = 0 (shift 1 bit per clock)
    (0L <<  4) |    // DATA_CAPTURE_MODE = 0 (detect rising edge)
    (0L <<  3) |    // INV_OUT_CLK = 0 (normal clock)
    (0L <<  2) |    // CLKGEN_MODE = 0 (use clock from COUNTER)
    (0L <<  1) |    // CLK_CAPTURE_MODE = 0 (use rising clock edge)
    (0L <<  0);     // MATCH_MODE = 0 (do not match data)

LPC_SGPIO->PRESET[0] = 1;
LPC_SGPIO->COUNT[0] = 0;
LPC_SGPIO->POS[0] = (0x1FL << 8) | (0x1FL << 0);
LPC_SGPIO->REG[0] = 0xAAAAAAAA;     // Primary output data register
LPC_SGPIO->REG_SS[0] = 0xAAAAAAAA;  // Shadow output data register

// Slice D (clock for Slice A)
LPC_SGPIO->SGPIO_MUX_CFG[3] =
    (0L << 12) |    // CONCAT_ORDER = 0 (self-loop)
    (1L << 11) |    // CONCAT_ENABLE = 1 (concatenate data)
    (0L <<  9) |    // QUALIFIER_SLICE_MODE = X
    (0L <<  7) |    // QUALIFIER_PIN_MODE = X
    (0L <<  5) |    // QUALIFIER_MODE = 0 (enable)
    (0L <<  3) |    // CLK_SOURCE_SLICE_MODE = 0, slice D
    (0L <<  1) |    // CLK_SOURCE_PIN_MODE = X
    (0L <<  0);     // EXT_CLK_ENABLE = 0, internal clock signal (slice)

LPC_SGPIO->SLICE_MUX_CFG[3] =
```

(continues on next page)

(continued from previous page)

```
    (0L <<  8) |    // INV_QUALIFIER = 0 (use normal qualifier)
    (0L <<  6) |    // PARALLEL_MODE = 0 (shift 1 bit per clock)
    (0L <<  4) |    // DATA_CAPTURE_MODE = 0 (detect rising edge)
    (0L <<  3) |    // INV_OUT_CLK = 0 (normal clock)
    (0L <<  2) |    // CLKGEN_MODE = 0 (use clock from COUNTER)
    (0L <<  1) |    // CLK_CAPTURE_MODE = 0 (use rising clock edge)
    (0L <<  0);     // MATCH_MODE = 0 (do not match data)

LPC_SGPIO->PRESET[3] = 0;
LPC_SGPIO->COUNT[3] = 0;
LPC_SGPIO->POS[3] = (0x1FL << 8) | (0x1FL << 0);
LPC_SGPIO->REG[0] = 0xAAAAAAAA;     // Primary output data register
LPC_SGPIO->REG_SS[0] = 0xAAAAAAAA;  // Shadow output data register

// Start SGPIO operation by enabling slice clocks.
LPC_SGPIO->CTRL_ENABLED =
    (1L <<  3) |    // Slice D
    (1L <<  0);     // Slice A
```