Tiny Python Panadapter
Martin Ewing, AA6E

Supplementary Information for QST Article [update 3, 9 June 2014]

This document is an expanded and revised version of the article printed in QST (April, 2014). Red text corresponds to latest revisions. It has been updated, but it is probably not fully current, since the project has continued in development after publication.

The most up-to-date information on the TPP project is to be found on-line:

1. Program files, git repository, and related technical material are permanently stored at SourceForge.net - https://sourceforge.net/projects/tinypythonpanadapter/ .

2. Project news, discussion of user issues is available on a SourceForge mailing list. You can mail questions to tinypythonpanadaptor-discussion@lists.sourceforge.net .

For a free subscription, sign up at https://lists.sourceforge.net/lists/listinfo/tinypythonpanadapter-discussion .

3. A "lab notebook" about the TPP project with lots of technical and installation information is provided at http://www.aa6e.net/wiki/Tiny_Python_Panadapter .

4. Text files (e.g. Python source) in the enclosed zip archive may be in Linux format. They can be read with Microsoft Word or similar software or converted to Windows format using the gedit text editor (available for Linux or Windows at https://wiki.gnome.org/Apps/Gedit).

Getting the Latest Program Version

The normal way to get the latest released version is by downloading through https://sourceforge.net/projects/tinypythonpanadapter. Code developers may also wish to access the latest contributions in the GIT repository https://sourceforge.net/p/tinypythonpanadapter/code/ci/master/tree/.

Raspberry Pi: First Steps [expanded]

Here is a step-by-step list of things to do to set up the Pi for the panadapter project. Many of these are useful for anything you may want to do with your Pi.

- Using an SD card (suggested 8 GB class 4 or above) and a PC, download the NOOBS system according to the website above. (We assume you have an Ethernet connection through your home router to the Internet.)
- Select Raspbian, download that system, and reboot as directed.
- You will be presented with a menu of configuration options. Use the tab key to skip to desired options and the enter key to choose one. (If you drop out of this menu and want to get back at any time, you can type the *raspi-config* command.) Recommended choices:

    - Change your password. Choose a good personal password to ensure some level of security. The username (pi) does not change. (You can also set up a completely independent account with a new user name. See info for the BeagleBone Black below.)
    - Don't enable boot directly to desktop, unless you know that you want to. The Pi will boot to the command line, and you can do what you need with the command line, or use the *startx* command to manually start the graphical desktop.
    - Under advanced options, select the item that enables SSH server. This will allow an outside computer (e.g., your PC) to connect to the Pi using the secure SSH protocol. (Outgoing SSH is always allowed.)
    - Select *finish* and hit Enter when you are done with the menu.
- The initial Raspbian download is set up for UK operation – timezone, keyboard, locale, etc. You may want to set up for a different country and timezone.
    - To select your timezone, type *sudo dpkg-reconfigure tzdata* and follow the menus. In the end you will set up something like "America/New_York", which will appear afterwards in the file /etc/timezone.
    - Set up your locale (rules for character sets, lexical conventions, etc.). Type *sudo dpkg-reconfigure locales.* For a US location, make sure that the "en_US.UTF-8 UTF-8" locale is enabled. (Turn on the asterisk by hitting the space bar.) When you reach the screen about default locale, be sure to set "en_US.UTF-8 UTF-8" as your default.
    - You may want to set up a standard US keyboard. Type *sudo dpkg-reconfigure keyboard configuration*. You probably want the "Generic 104 key" option (not International). You probably want keyboard layout "English US", "No AltGr Key", and "No Compose Key". You can choose either presented option for Control+Alt+Backspace.
    - Reboot the computer (*sudo reboot*) to make all the changes effective.

BeagleBone Black: First Steps [Expanded]

The BBB requires a bit more work to configure for our project than the Pi, but in the end we will have a system that is configured much like the Pi above. We will give instructions for bringing up a Debian 7.0.0 "Wheezy" (or later) Linux system on the BeagleBone Black (BBB). This system uses the kernel provided by Robert C. Nelson, called *"Linux arm 3.8.13-bone21 #1 SMP Thu Jun 13 23:52:15 UTC 2013 armv7l GNU/Linux"*. A later kernel should be OK. Various other Linux distributions are available for the BBB. They might work for our project, but we haven't tried them.

- Read http://elinux.org/BeagleBoardDebian. This page gives a number of options that more technical readers can use to get a basic working Debian system, including how to build your own kernel.
- Obtain an 8 or 16 GB microSD card for the Debian installation. (We will not be using the BBB's on-board 2 GB "eMMC" flash memory. I recommend keeping the Angstrom Linux on eMMC that comes with the BBB from the factory for later testing.)
- Follow instructions to download Debian system on your local Linux PC (using a micro to full-

size SD adapter if needed) and install on microSD.

- Attach an HDMI (or DVI-D) monitor, USB hub, and keyboard and mouse to your BBB. Insert the microSD in your BBB and boot. You should see a photo of Tux, the Linux penguin, followed shortly by a text login (user:debian, password:temppwd).
- Check that your Internet connection is up and running. (*ifconfig eth0; ping arrl.org*)
- Type command *sudo apt-get task-lxde-desktop*. If you want a different desktop system you could ask for a different "task", such as "task-gnome-desktop". Be patient: many packages will be downloaded and installed. It's a good time to organize your shack.
- When finished, reboot the computer (*sudo reboot*), and you should see an LXDE login screen, where you can select user "debian" and enter password "temppwd".
- The Debian LXDE desktop should appear!
- Change password for account "debian" to something private for you, using a terminal emulator window and the "*passwd*" command.
- If needed, you can use LXDE's "X" (start) menu → Other → LXRandR to specify a preferred screen resolution. Sometimes a lower setting can provide more stable video. Click "Save" to make it permanent.

Continuing with your Pi or BBB [Expanded]

Most of the following is the same in either the Pi or BBB.

- Install several packages: "*sudo apt-get install python-pygame python-libhamlib2 python-dev portaudio19-dev python-numpy*". (Some of these may already have been installed.) If asked whether it's OK to remove libjack-jackd2-0, answer *yes*.
- We want PyAudio software also. There may be a version in your repository (python-pyaudio), but we need version 0.2.7 or later, and the repository may not be up to that level. [The current version is 0.2.8, as viewed 5/11/2014.] If so, we need to download and install PyAudio "by hand":
  - Use web browser Midori (Pi's standard browser) or IceWeasel (BBB's) and go to http://people.csail.mit.edu/hubert/pyaudio/
  - Scroll down to "PyAudio Source" and click on "PyAudio Tarball.[1]". Download the compressed archive pyaudio-0.2.8.tar.gz to a convenient folder, such as your home directory.
  - Change directory (*cd*) to that folder and type *tar xzf pyaudio-0.2.8.tar.gz* to decompress and extract the folder PyAudio-0.2.8. Change (*cd*) to this folder and give the command *sudo python setup.py install*. The process should complete in a minute or so.
  - Verify the installation by running Python and trying to import PyAudio:

    Type "*python"* followed by the Enter key. Python will start and give its prompt: ">>>"). Then type *import pyaudio* followed by the Enter key.

    If there is no error message, you have successfully loaded PyAudio. To see the actual version number, you can type *pyaudio.__version__*.

- (Optional) If you want support for the RTL2832U-based DVB-T dongle (RTL-SDR):
  - You may need to run *sudo apt-get install autoconf libtool make libusb-1.0-0-dev* to install those packages.
  - Download, build, and install rtl-sdr using instructions at *http://sdr.osmocom.org/trac/wiki/rtl-sdr*. Use the "autotools" instructions on that site, unless you have another preference. Install udev rules as suggested. With RTL dongle installed, run *rtl_test* to verify correct installation.
  - Download and install *pyrtlsdr* using *git clone https://github.com/roger-/pyrtlsdr.git* and then (in pyrtlsdr directory) *sudo python setup.py install*.
- (Optional) Install ntp package for accurate network-based timekeeping. (Ntp is already set up on the Raspberry Pi.)
- (Optional) Create a normal user account for yourself. If your name is Hiram: (*sudo adduser hiram*) Give your account the following groups: sudo, dialout, audio. (*sudo adduser hiram sudo*, *sudo adduser hiram audio*, etc.)
- You may wish to customize your computer name and set up your home network.
  - To change your computer's name, edit the file /etc/hostname and reboot. Use LeafPad or your favorite text editor.
  - To make your other local computers reachable by name, edit the file /etc/hosts, giving the numerical IP address, then spaces, then the computer name for that address. (Check *man hosts* for more information.)
- (Optional) Install "gedit" -- a better editor than LXDE's LeafPad, especially for programming in Python. A lot of additional packages will be pulled in. (On the Pi, installing gedit may remove certain other packages, including ruby, clang, llvm, etc. You can reinstall them later, if needed.)

Run-time Issues

The programming challenge in a project like this is mainly to keep the CPU demand low enough to fit the capabilities of our small computer boards. It is a good idea to keep the CPU average utilization below about 75% for reliable operation. Linux is not a real-time operating system, which means that your application can be interrupted from time to time for system housekeeping and other tasks. Python also has its limitations for real-time operation. It has multi-threading capability, which we use, but it does not understand process-level multiprogramming or threading across multiple CPUs or cores. Sometimes you may find that the Linux *nice* command with negative "niceness" will boost the priority of *iq.py* enough to keep it running more smoothly. You can try invoking the program as

nice -20 python iq.py ...

Many factors can reduce CPU demand, including increased data chunk size (*n_buffers*), lower sample rate (44100 Hz or less), reduced FFT size, rejecting some data (reducing *take*), avoiding the waterfall option, etc. In general, you have a trade-off between display update rate, sensitivity, and frequency resolution. Feel free to experiment with these parameters to get the most pleasing results.

FFT Size

Another question is how to choose the best FFT size. The greater the FFT size, the finer the frequency resolution will be at the expense of greater CPU demand. However, there is no reason to use a size greater than the width of the display spectrum, in pixels. Additionally, the FFT algorithm works much faster for some sizes than others. The favorable sizes are numbers that are "highly composite" – the product of small integers, like 2, 3, 5, etc. (The worst choices will be prime numbers – numbers that are only divisible by themselves and by 1, like 127 or 257.) Table 4 gives some examples of some sizes you might choose, along with measured Numpy FFT execution time on the BBB and Pi. For example 224 = 7 x 2 x 2 x 2 x 2 = 7 x 2**5 is one reasonable choice, but powers of 2 (256, 512, etc.) will likely be fastest.

Factors	**Number**	**FFT (avg uS)**	
		BBB	**RPi**
7 x 2**5	224	261	580
2**8	256	232	576
prime	257	2840	3450
3*2 x 2**5	288	303	637
5 x 2**6	320	305	634
3 x 2**7	384	375	751
7 x 2**6	448	476	914
2**9	512	468	897
3**2 x 2**6	576	566	989

Table 4. Some highly composite numbers and FFT timing for BeagleBone Black and Raspberry Pi.

The FFT load is usually not a major concern for *iq.py*, if you choose a reasonable size. Graphics and other functions dominate.

Graphics Options

Our code uses PyGame's graphics, but PyGame supports various graphics drivers. During development, when you are probably running with a large format video monitor, you may want to work in a window under your normal desktop environment (usually X11). This is PyGame's default mode. But if you intend to run the *iq.py* panadapter in a dedicated environment with a smaller display, like the LCD4, you may not want all the complexity of the X11 desktop, which takes CPU time and memory to do things you won't need.

PyGame will also work with non-X11 drivers. We have tried directfb with some success. If you have the libdirectfb package installed, you can boot your system to the command line without any graphical desktop (before typing *startx*) and simply run *iq.py*. Directfb should find the monitor and display the panadapter window. In some cases, you may need to set an environment variable before running *iq.py*, using the command

export SDL_VIDEODRIVER=directfb .

Directfb needs some configuration files to work with the LCD4 display. Create the file '.directfbrc' in your home directory (or create /etc/directfbrc) with the contents:

```
system=sdl
mode=480x272
depth=16
```

Add the following to the file /etc/fb.modes:

```
mode "480x272"
    geometry 480 272 480 272 0
endmode
```

To use directfb, you need to boot the BBB into text mode (without X11). One way to make that happen is to edit the file /boot/uboot/uEnv.txt to change the line

```
optargs=
```

to

```
optargs=text
```

After this when you reboot, you will always get the command line without X11. Then you would use SSH to connect to the BBB and then invoke *iq.py* as root. For example,

sudo python iq.py --LCD4 …

If you want your computer to automatically boot into *iq.py,* you can place the *python iq.py* command into /etc/rc.local.

Directfb can also be used on the Pi, but the LCD4 display is not available. You can use directfb with any size monitor that is connected through the HDMI port or that has a direct connect to the Pi (such as the Adafruit 2.8" touch screen), as long as appropriate driver software is installed. For alternate displays, you may want to change the "480x272" pixel configuration above to the actual pixel size of your device.

Frequency Readout & Control Options

The original TPP release allowed frequency to be read and controlled and two ways: 1) via Hamlib and a radio's CAT port, if available, or 2) via the built-in control interface in the RTL-SDR dongle. In version 0.3.6, we add the ability to control and read out frequency in radios that provide a USB-connected Si570 DDS VFO chip, as available in the SoftRock RxTx Ensemble, for example.

Special Problem with Raspberry Pi

The Raspberry Pi has serious problems running with a high-speed USB soundcard. I have found that there is no "stable" (hangup-free) configuration with the following configuration: latest Raspian (2014-01-07), X11 (startx), 48,000 Hz sampling, UCA202 or iMic soundcards, HDMI display. Using 44,100 Hz sampling may be a little better, but 22,050 works fine. (The Pi appears to be just too slow to handle the continuous data flow. This has been noted for data output. See this, but this.)

Confirming these Internet hints, it turns out the key problem is that the Pi cannot run USB 2.0 reliably for our purposes -- continuous 48 kHz or higher sampling rates. (The data rate is about 48K x 2 x 2 = 192 kB/s, much less than USB2.0's theoretical maximum of 35 MB/s. Still, it seems to be too fast for

the Pi to control reliably.)

We can slow down the USB transactions dramatically by dropping back to USB 1.1 (theoretical maximum speed 1.5 MB/s). This is accomplished by editing the Pi's boot file /boot/cmdline.txt. Insert the parameter 'dwc_otg.speed=1' at the beginning of the line. (dwc_otg.speed=0 signifies default USB 2.0 behavior.) In my case, I would have

```
dwc_otg.speed=1 dwc_otg.lpm_enable=0 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200
console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline rootwait
```

You will need to reboot to have this become effective.

With this change, the Raspberry Pi will run TPP very much better. But there are a couple of catches:

- With your Pi's USB bus on version 1, many keyboards and mice will not operate. They often assume USB 2.0. Fortunately, the Ethernet connection is still available. If you can't find a compatible keyboard and mouse, you will have to connect with SSH from another device.
- If you want to use the RTL-SDR option, chances are that your DVB-T / RTL dongle requires USB 2.0! So if you want to mix RTL-SDR and Audio soundcard 48 kHz operation, you have a problem. For now, you need to edit /boot/cmdline.txt and reboot to switch from one to the other. Or spend $35 for another Raspberry Pi...

Operating Controls

Most of the operating conditions for iq.py are set up through the command-line options discussed above. After the program starts, though, it is convenient to be able to change the display scaling and some other features.

When you have only the minimal BBB + LCD4 configuration, your only controls are the 5 push buttons on the LCD4. So most of the commands are defined in terms of these 5 buttons and their ASCII equivalent characters.

The Enter button brings up a help screen that overlays the spectrum display in two areas. The top overlay gives a list of commands you have available, and the bottom overlay shows a real-time display of some operating parameters.

Each time you click Enter, you get a different help screen with other options, until finally you arrive back at the normal spectrum display with no help overlay. At present there are only three help screens, but any number can be added.

The screen-shot in Fig. 4 was taken on a Raspberry Pi connected to an HDMI display and a Sound Blaster SB1240 USB sound card accepting I/Q data from a KX-3. The command was

python --size=384 --n_buffers=15 --take=4 --WATERFALL --index=2 .

Following is a detailed list of the screen overlays that are particularly useful for the LCD4 display with its limited set of buttons.

Overlay 1: Keyboard Controls

These are the keys you can use if you have a full ASCII keyboard available.

- "R" resets the display to initial scale settings
- "Q" quits the program.

These commands affect the scaling of the "2D" spectrum in 10 dB increments:

- "U" increases and "u" decreases the upper dB limit. (Default -20)
- "L" increases and "l" decreases the lower dB limit. (Default -120)

The following 4 commands affect the waterfall scaling in 10 dB increments:

- "B" increases and "b" decreases the upper limit of the palette (Default -20) [The "bright" limit]
- "D" increases and "d" decreases the lower limit of the palette (Default -120) [The "dim" limit]

If Hamlib is active, you will have these radio control selections:

- RIGHT (→) - If shift, +large freq. step, otherwise +small freq. step.
- LEFT (←) - If shift, -large freq. step, otherwise -small freq. step.

Overlay 2: Spectrum Adjustments

These functions can be invoked through a keyboard or LCD4 buttons.

These buttons adjust the "2D" spectrum plot scaling:

- UP (↑) - increase upper screen level 10 dB
- DOWN (↓) - decrease upper screen level 10 dB
- RIGHT (→) - increase lower screen level 10 dB
- LEFT (←) - decrease lower screen level 10 dB

Overlay 3: Waterfall Palette Adjustments

Only available if --WATERFALL option was selected.

These functions can be invoked through a keyboard or LCD4 buttons.

These buttons adjust the waterfall palette thresholds.

- UP (↑) - increase upper threshold 10 dB
- DOWN (↓) - decrease upper threshold 10 dB
- RIGHT (→) - increase lower threshold 10 dB
- LEFT (←) - decrease lower threshold 10 dB

Status: (Shown at bottom of display for overlays 1-3)

The following are the current values set through the help screens above:

- dB scale for 2D spectrum graph (min, max)
- dB scale for waterfall (min, max)

The following stay the same while the program runs:

- Fs - Sample Rate
- Res - frequency resolution (bandpass divided by size)
- Chans - Size of FFT
- width - width of spectrum in pixels

- acc - accumulation time per plot

The following are updated periodically:

- ADC max (I and Q) - maximum digital counts from A-D converter. Useful to verify appropriate input and ALSA gain levels.
- Load - Current CPU loading reported by Linux (user state, system state, and 1 minute load average) User state over 0.75 may cause problems.

Command-Line Options [supplemental]

Table 3 is a full list of the command line options you can use.

Option	Default	Action
Boolean options (no arguments):		
--help	n/a	Show all available options.
--FULLSCREEN	(normal window)	Use full screen for display.
--HAMLIB	(no hamlib)	Use Hamlib to read/set receiver frequency. (Not for RTL-SDR)
--LAGFIX	(normal operation)	Compensate for R/L lag bug in some PCM290x sound cards.
--LCD4	(HDMI display)	Use LCD4 cape on BeagleBone Black
--REV	(Not reversed)	Flip display freq. axis – equiv. to swapping I&Q
--RTL	(sound card input)	Use RTL-SDR dongle for input
--Si570	(no Si570)	Use Si570 VFO control via USB
--WATERFALL	(no waterfall)	Include waterfall display below spectrum graph.
Options with one argument:		
--cpu_load_intvl=<float>	3.0	Sleep (secs) between CPU load checks
--hamlib_device=<str>	"/dev/ttyUSB0"	Address for Hamlib serial port adapter
--hamlib_intvl=<float>	1.0	Sleep (secs) between Hamlib commands
--hamlib_rig=<int>	229	Hamlib ID for rig type (default Elecraft K3)
--index=<int>	-1 (use default device)	PyAudio device index for input
--lcd4_brightness=<int>	75	Brightness, 0-100. (Setting requires root status.)
--n_buffers=<int>	12	No. of sample buffers per input "chunk"
--pulse_clip=<int>	10	Relative clipping level for large pulses
--rate=<int>	48,000 (sound card) 1,024,000 (RTL_SDR)	Set sampling rate in Hz.
--skip=<int>	0	If >0, skip every Nth buffer. If <0, skip all *except* (-N)th buffer.
--rtl_freq=<int>	146.e6[2]	RTL center frequency, Hz.
--rtl_gain=<int>	0 (auto)	RTL gain seeting
--si570_frequency	7050.0	Initial frequency for Si570, if selected (kHz)
--size=<int>	384	FFT dimension (# data frames / buffer)
--sp_max=<int>	-20	dB level for top of graticule
--sp_min=<int>	-120	dB level for bottom of graticule
--v_max=<int>	-20	dB level for maximum palette value
--v_min=<int>	-120	dB level for minimum palette value
--waterfall_acc=<int>	4	No. of spectra to make 1 waterfall line
--waterfall_palette=<int>	2	Color mapping for waterfall (1 or 2)

Table 3. Program Options

Exploring the Code

While you can have some fun just loading the program and running the panadapter, we hope you will take a look at the Python code, figure out how it works, improve it (fix the bugs!), and adapt it for your particular interests. It's all open source under the GPL (Gnu Public License, *www.gnu.org/licenses/gpl.html*), meaning that you are free to use it, extend it, and distribute your work, as long as you provide your code to everyone on the same terms, as open source.

It will help a great deal if you familiarize yourself first with the Python language and the basics of Object Oriented Programming. There are many books and web resources you can consult.[3] [4]

The remainder of this section is a narrative description of the code. It's meant to be read alongside the code listing itself. I apologize to my expert readers if some of the comments here are obvious and elementary, but I hope to make the program accessible even to relatively new Python programmers. As you get into it, you will see that the code is really open-ended – there are lots of areas where you could make changes to tailor things to your particular interests and your particular hardware.

These notes refer to the code in version 0.30. There may be slight differences if you have a later version. I welcome readers' comments and bug reports.

Main Program & Graphics – iq.py

The main program is structured as a PyGame application (*www.pygame.org/docs/*). PyGame is a simple graphical programming environment that is oriented toward games. It turns out this is appropriate for the panadapter project, because it is relatively simple and fast compared with full-featured GUI systems like wxWidgets or GTK+. Games want to maximize the display frames per second, and so do we. We only use the graphics and keyboard processing parts of PyGame, ignoring other features such as audio.

The program begins by importing the modules[5] that will be needed. (Hamlib will be imported later, if requested.) Then, a range of colors is defined for convenience.

When *iq_opt* was imported, that module assembled all the options from the command line and created an object that has named variables corresponding to each option value you specify – or a default value. This object *options* is copied to a local variable *opt* in *iq.py* for convenience, and then most options are printed to your terminal window.

Next, there are a number of function and class definitions:

- *quit_all* is a function to quit the program gracefully in case of error.
- *LED* is a class that provides a graphical object (a PyGame "surface") which is a visual representation of an LED (a colored disk with a black outline).
- *Graticule* is a class that makes the oscilloscope-like graticule to calibrate the spectrum display. Usually, it is shown as faint red lines with numerical calibrations in dB and kHz.

There are two functions that are meant to be executed in separate Python threads. These are functions that involve blocking I/O that should not be allowed to interrupt the main signal processing and display loop:

- *updatefreq* is defined (but started later) to manage communications with your radio via Hamlib. (It is only defined if you have requested it via the --HAMLIB option.) It will set the current frequency into the global *rigfreq*. If the main program has set a different frequency in global *rigfreq_request*, Hamlib will command the radio to change to that frequency. Hamlib I/O usually requires sending a command via serial I/O and waiting for a reply from the radio. When these "slow" operations block the thread, control returns to the main program keeping things running smoothly.
- *cpu_load* will periodically get current average system load information from Linux and place it in the global *cpu_usage*. This changes only slowly. It is only used for diagnostics purposes when using the special screen overlays described above.

Now, we need to initialize the PyGame display. *SCREEN_SIZE* is set to the hardware size of the LCD4 display if the --LCD4 option was given. In the LCD4 mode, you can also set the display brightness, but only if you are running as root. Otherwise, *SCREEN_SIZE* and *SCREEN_MODE* are set to standard values. Graphics ultimately are drawn to the PyGame surface *surf_main*, which is created here. We make a number of calculations for the screen layout, and we ensure that there is at least one display pixel for each spectrum point, reducing the *size* parameter if necessary.

We instantiate a *dataIn* object that will get input from our radio device. The modules *iq_rtl* and *iq_af* respectively define the two classes *rtl.RTL_In* for the RTL-SDR dongle or *af.DataInput* for sound card input.

The object *myDSP* instantiates the class *dsp.DSP*, which will does most of the numerical calculations. We create empty surfaces for the 2D live spectrum, and for the waterfall. There are two LEDs to signal "buffer under-run" and "pulse clip". Three fonts are defined for later use.

If a waterfall is requested, we create a waterfall object *mywf*. If Hamlib is requested, we initialize the rig connection and start the Hamlib thread. We start the CPU load monitor thread, and then instantiate a graticule and make a graticule surface for the display.

Finally, we compute a *parms_matter* surface to hold the parameter data which will not change in the main loop. This will be used later in the overlay screens.

Now we are ready to begin the main infinite "while" loop:

Clear *surf_main*. In the remainder of the loop, our job is to build a new main display frame.

If we are using Hamlib or RTL-SDR input, we display the current operating frequency.

If we are using a sound card,

- Check for an under-run signal from *iq_af* and set the LED accordingly.
- Check for a led_clip signal from *iq_dsp* and set the LED accordingly.

If we are using RTL-SDR for input, read a chunk of data in blocking mode. This is probably necessary because we'd have trouble keeping up with the maximum RTL data rate of 2 MHz in non-blocking mode.

If not RTL-SDR (i.e., if using sound card):

- Pass unless there are at least 2 chunks available in the data queue. (The data is enqueued the

iq_af module.) When ready, read a chunk from the queue as a raw string of bytes. The data queue is the mechanism for synchronization between the incoming real-time audio data stream (interrupt/call-back driven) and the display loop.

- Convert the byte string consisting of interleaved "right" (I) and "left" (Q) channel 16-bit samples into two floating arrays, *re_d* and *im_d*. This is efficiently handled with just 3 Numpy calls.
- If needed, apply a special fix for PCM290x chip, rotating (delaying) *im_d* by 1 clock. (Minor problem: one sample will be processed incorrectly out of the chunk.)
- Set *stats* array with the maximum values of *re_d* and *im_d*, and then combine the two arrays into a single complex array *iq_data_cmplx*.

Compute the log power spectrum *sp_log* using *myDSP.GetLogPowerSpectrum*.

If we are handling RTL-SDR input, we need to boost the output by (say) 60 dB, because RTL data is normalized to +/- 1. This helps get it onto our spectrum scale.

Copy ("blit[6]") the current graticule to *surf_2d*.

Using PyGame's *draw.lines* function, plot the live 2D spectrum on *surf_2d*.

Copy/blit *surf_2d* to *surf_main*.

If a waterfall is requested, calculate a new waterfall surface (*surf_wf*) and blit to *surf_main*.

Check *info_phase*. If positive, we will need to apply a help overlay. Set up a *help_matter* surface with help info according to *info_phase*. Set up *live_surface* with live (dynamically updated) status information (scale info, I/Q data maximums, cpu load). Blit both surfaces to *surf_main*, covering over some spectrum and/or waterfall data.

If needed, show labels for LCD4 buttons at right side of screen.

Check for any PyGame events (keyboard presses). Use key commands to change plot scale settings, etc. We may need to recalculate the graticule.

Having computed our new *surf_main*, we issue PyGame's *display.update* function to update the user's display.

This ends the main loop.

Audio Input – iq_af.py

The *iq_af* module (file *iq_af.py*) contains all the code that interacts with PyAudio. It manages the input from your sound card.

The main part is the class *DataInput* which initializes the data input process. First, the PyAudio subsystem must be initialized. Unfortunately, the underlying PortAudio library always seems to emit many error or warning messages about devices that are unknown or that can't be found. They can be ignored.

We set up *af_using_index* either as the default ALSA input device or as the user-requested index. Then we verify that the requested input mode is valid. If so, we open a PyAudio stream *afiqstream* (or

self.afiqstream[7]), specifying a call-back routine *pa_call-back_iqin* that will receive control (in a separate thread, effectively an interrupt handler) when a full chunk is received.

We create the queue *dataqueue* to serve as an interface between the call-back thread and the signal processing and display that are in the main control thread.

The call-back function *pa_call-back_iqin* should be kept very simple, as you would for an interrupt handler. It checks for an under-run error condition (which will signal via a display "LED"). It will then enqueue the chunk *in_data* to be handed over to the main thread. An error is detected if the queue overflows. That will terminate the program with an error message. Generally this means that there was not enough time to process the prior chunks. That is, the CPU can't handle the load. (Try again with less demanding command-line options.)

Other methods of *DataInput* provide for starting, stopping, and termination of the audio stream.

Signal Processing – iq_dsp.py

The panadapter software is typical of many modern computer projects. Most of the code complexity and cpu time is associated with the graphical user interface. But from the point of view of receiver engineering and software defined radio, the "interesting" part is the digital signal processing, which is mainly in the *DSP* class in the module *iq_dsp.py*.

Let's describe what happens in *DSP*. We will keep the math as simple as possible here. If you're interested in going further, the full details are available in many DSP references[8]. A more introductory overview is also available[9].

When the *DSP* class is instantiated (in the main program, *iq.py*), the *__init__* function is called to establish some counters, arrays, and constants. In particular, since we know the size of the FFT array (passed in as *opt.size*), we can compute *db_adjust* to scale the FFT data so that 0 dB in the log power spectrum corresponds to the theoretical maximum signal we can see in a 16 bit sample. (A pure input sine wave that is just below clipping level will produce this maximum value in a single frequency bin.) This constant will be used to offset (normalize) the output data.

Although it's debatable whether it is needed, we also compute a "windowing" function (a so-called Hanning window) that will taper the input data before the FFT. This controls the "sidelobes" of any narrow-frequency structure. In practice, with a typical noise environment, we don't see much effect.

The DSP *GetLogPowerSpectrum* method does all the work on a chunk of input data. At this point, the input data is complex floating point samples, with the real part being "I" and the imaginary part being "Q". A chunk has *opt.buffers* buffers, with each buffer having *opt.size* samples. (The FFT works on a single buffer at a time.)

Now we are beginning to use some interesting Numpy functions that operate on arrays of data. For the first buffer of a chunk, we compute *td_median* which will be the median value of the absolute value of the first data buffer. Numpy gives us *np.median* and *np.abs* that make this simple.

Why do we need such a median value? It turns out that the noise we encounter on the HF bands has a lot of impulse noise. (Think lightning crashes.) A broadband noise pulse will add a huge amount of power to the incoming data stream and will upset our average power spectrum calculation if allowed to go through. Instead, we use *td_threshold*, which is *opt.pulse* (a command line parameter) times

td_median. If the maximum data value in a buffer has a magnitude (absolute value) greater than *td_threshold*, we ignore it and we set *led_clip_ct* which will turn on our graphical "clip LED". In practice, this process helps to noticeably clean up the spectrum display.

If the buffer is not thrown out by the threshold test, it is multiplied by the window function *w*. (Note that the Numpy arrays are multiplied together element by element.) Next, the Numpy *fft.fft* function converts the buffer of time samples into frequency channels, i.e. *fd_spectrum*. By convention, the first spectrum point (the zero channel) corresponds to zero frequency (DC) with positive frequencies increasing from 0 up to *size*/2 and negative frequencies going downward, wrapping into the top half of the FFT output array.

For the panadapter display, we want the output to have zero at the center with negative frequencies on the left and positive on the right. Numpy's *fft.fftshift* function rotates the spectrum array appropriately.

The FFT gives a voltage-like complex output in each frequency channel. The display wants to show the power spectrum, i.e., the power of the incoming signal in each frequency channel. Numpy doesn't give us just the right function for this, but it's simple to compute the power. It's the complex spectrum channel times its conjugate value.[10] (This reduces to the square of the voltage for a real-valued spectrum.) We convert the power spectrum into a real-valued array and accumulate it in the array *power_spectrum*.

We could try to display *power_spectrum* directly, but our signals generally cover a large dynamic range, making it difficult to see smaller signals in the presence of strong ones, and also making it difficult to set the plot scales appropriately. Instead, we will show the logarithm (log) of *power_spectrum*. The new array will be *log_power_spectrum* in dB.

Finally, we apply *sdb_adjust* to normalize the signal so that it will never exceed 0 dB.

Waterfall – iq_wf.py

The palette is used to translate a power spectrum value into an RGB color for display. Many strategies are possible. We have defined two, but users can add as many functions as they like! The function *palette_color* accepts a palette ID number (1 or 2), the value to be converted, and the minimum and maximum values that are expected. It returns a tuple of 3 integers for red, green, and blue in the range 0 - 255.

The class *Wf* represents a waterfall object. It is initialized with *opt* (the command-line options), *vmin* and *vmax* – minimum and maximum spectrum values to establish a range, *nsteps* to determine the number of color steps that are used to cover the range, and *pxsz* a tuple which gives the width and height of a single "pixel" of the waterfall – one frequency channel.

The palette is initialized (or reinitialized) by creating a list of pixel-size surfaces, each filled with the computed color for one step of the *palette_color* function. One of these surfaces will later be blitted in to the waterfall surface for each frequency point, depending on its value.

The *calculate* method does most of the work each time a new row is to be added to the top of the waterfall display. The waterfall actually accumulates *nsum* log power spectra per waterfall line in the accumulator *wfacc*. We then blit the waterfall *surface* into itself, one row down, i.e. rolling the waterfall down one step. Then the new line is blitted in at the top.

RTL-SDR Mode – iq_rtl.py

Module *iq_rtl* is a simple interface with the rtlsdr module. Class *RTL_In* sets up the RTL device according to the options requested in the command line. The method *ReadSamples* gets a requested number of 8-bit I/Q samples and returns a complex floating array, normalized so that its maximum values are +/- 1.

Command-Line Options – iq_opt.py

The *iq_opt* module uses the standard Python module *optparse* to scan the command line for the options described in Table 3 above. If an option is not specified, a default value is used. The option values are collected in the special object *opt*, which will be referenced in the main program *iq.py* as *options.opt*.

1 A tarball is a file created by the Linux tar utility (or other compatible software). It usually combines multiple files into one for handling convenience. Often, it is compressed using gzip or another compression program.

2 Frequencies are stored in integer Hz, but a floating point variable will be understood.

3 The basic reference is *www.python.org*, but I recommend reading one of the many available Python books. Beginners might choose Mark Lutz's *"Learning Python"*, 5th ed., 2013, O'Reilly Media. More experienced readers might consider David M. Beazley's *"Python Essential Reference"*, 4th ed., 2009, Addison Wesley.

4 We are using Python version 2.7. The latest versions (3.x) are not fully compatible with version 2. Version 3 is gaining ground, but some Python modules are not yet available in Version 3.

5 Recall that a Python module is generally a ".py" file that contains a bunch of Python code that will be run or compiled into its own address space.

6 In computer graphics, a "blit" operation (block level transfer) is an efficient way to transfer an image from one image buffer (PyGame surface) to another.

7 If you are new to Python or Object Oriented Programming, you may be puzzled about the prefix "*self.*" that appears with variables in a class definition. If so, you need to study up some more! It's important. In a nutshell, the code in a class can be "instantiated" many times, making many objects. Each object has its own local variables, like *afiqstream.* The "*self.*" prefix keeps them separate. In our narrative description, we may drop the "*self.*", but it has to be there in the code.

8 Two key DSP references for Amateur Radio: A Series of QEX articles by Gerald Youngblood AC5OG, *"A Software-Defined Radio for the Masses"* (Jul/Aug, 2002, Sept/Oct, 2002, Nov/Dec, 2002, and Mar/Apr, 2003. And the book by Doug Smith KF6DX *"Digital Signal Processing Technology: Essentials of the Communications Revolution"*, ARRL, 2003.

9 Martin Ewing AA6E, *"The ABCs of Software Defined Radio"*, ARRL, 2012.

10 If x = a + jb, for real values a and b, the conjugate of x is defined as a – jb.