Foxhunt self-tester

SERG Home brew competition entry 2025

Bruce VK3TJN

What is it for?

The foxhunt self-tester is used by our foxhunt system to perform a self-test, as much as is possible.

It has to be able work on any band we use foxhunting.

The need for the self tester grew out of a problem we observed whilst foxhunting in Melbourne, but also will be able to be employed:

a) when gear has just been put on the roof, and we need peace of mind that it is working prior to a hunt

b) during a hunt when things are not going to plan, and we can't seem to hear the fox at all, or the directions are not making sense. A self test will be the first thing to try, before contacting the fox.

What was the problem?

On two foxhunts in Melbourne, we could hear the fox well, but got very strange but consistently inaccurate bearings. On both hunts we ended up in a suburb a long way away when the hunts were called in. Just what was going on?

A few things we noted:

- Both hunts were on 2m
- Both were the first hunt of the evening. Later hunts worked fine.
- On both hunts, some other fatal problem had occurred with our hand-rotating *front* system, so we were relying purely on the *rear* system, which uses a motor rotated beam and an SDR receiver.
- The *rear* system was rebooted after the problematic hunts.

Clearly *something* was affecting the back system polar plot in rare circumstances, causing it to give skewed bearings.

Simple tests at home under controlled circumstances didn't reveal any issues. This was clearly going to take a bit more detective work!

I have detailed the investigation into the SDR system in Appendix 1. This is a fascinating tale, but doesn't really belong here.

The foxhunt self-test solution was actually based on an idea from our beam swinger on one of the mis-direction hunts in question, Graham VK3GA. "It'd be great if there was some sort of test source that works on any band, that could test both front and back systems automatically". Easier said than done, I thought, thinking initially of something like oscillators for each band crammed into a box.

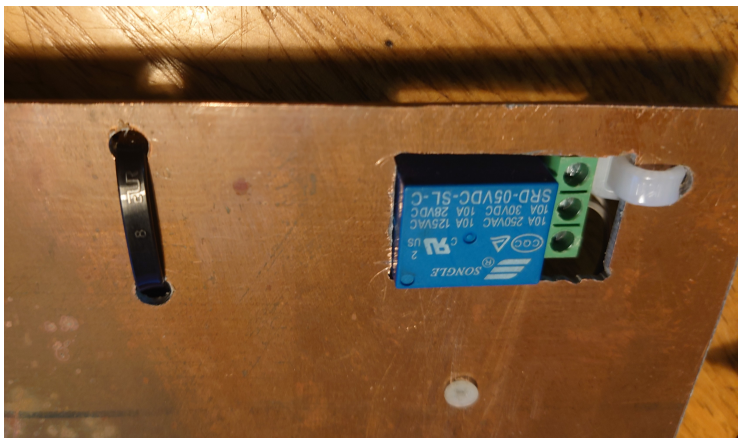
Later, I noticed that the LiIon battery in a little NanoVNA I'd bought a few years ago was expanding, and might crack the LCD display if I didn't take action and remove it. Oh well, at least I could still use it if required powered by USB, at least till I got a replacement battery…. I wonder if you can send it commands via the USB? Well, appendix 2 was the result of some googling. Maybe *this* can form the basis for a foxhunt self-test solution? Afterall, I'd already used the NanoVNA previously as a test signal for aligning antennas. Also, compared to other multi-band devices, they make quite a cheap programmable signal source.

The Tester box

The Tester box is a self-contained unit, powered and controlled by a single USB cable. Inside the box is a USB hub, a USB controlled relay and the NanoVNA (sans battery). The relay switches across the on/off switch, so the NanoVNA can be powered up only when required, but the USB-Serial of the Nano is always powered, avoiding unnecessary USB re-enumerations.

The S11 port (TX) is cabled externally for fitting an antenna to a BNC connector.

The relay had to be cut into the PCB mounting plate to fit into the box:



The Tester software

It's important that self-test can be performed easily on any band on demand when required, and this is where we can implement some smarts to make self-testing essentially a one button action.

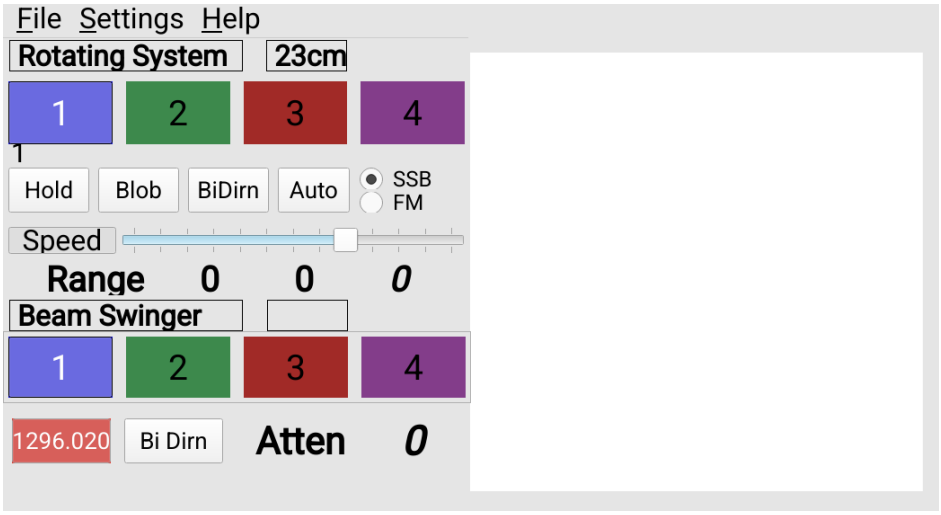
It's also important that testing does not happen on the fox frequency, as this could end up with teams inadvertently hunting us down! The solution is simple – it takes the current memory frequency and adds 20kHz. The NanoVNA is powered up, using serial commands sent to the USB relay, and the test frequency is sent to the Nano via USB. The SDR receiver is also set 20kHz above the current memory (as distinct from the current tuned fox frequency, which likely will have been tuned off the memory centre by now). Though the Nano has the ability to be set to output a CW signal, instead I have elected to output a short chirp with a 300Hz span around the desired frequency. This is to generate a distinctive test signal that could not be confused with a real fox, or other random spurious signals.

The tester times out after a while and reverts to normal foxhunt mode (and the previously tuned fox frequency), or testing can be exited immediately by pressing the test button again. The test button flashes red during self-test to avoid any confusion. The NanoVNA is powered off after, to avoid any

potential spurious signals when hunting.

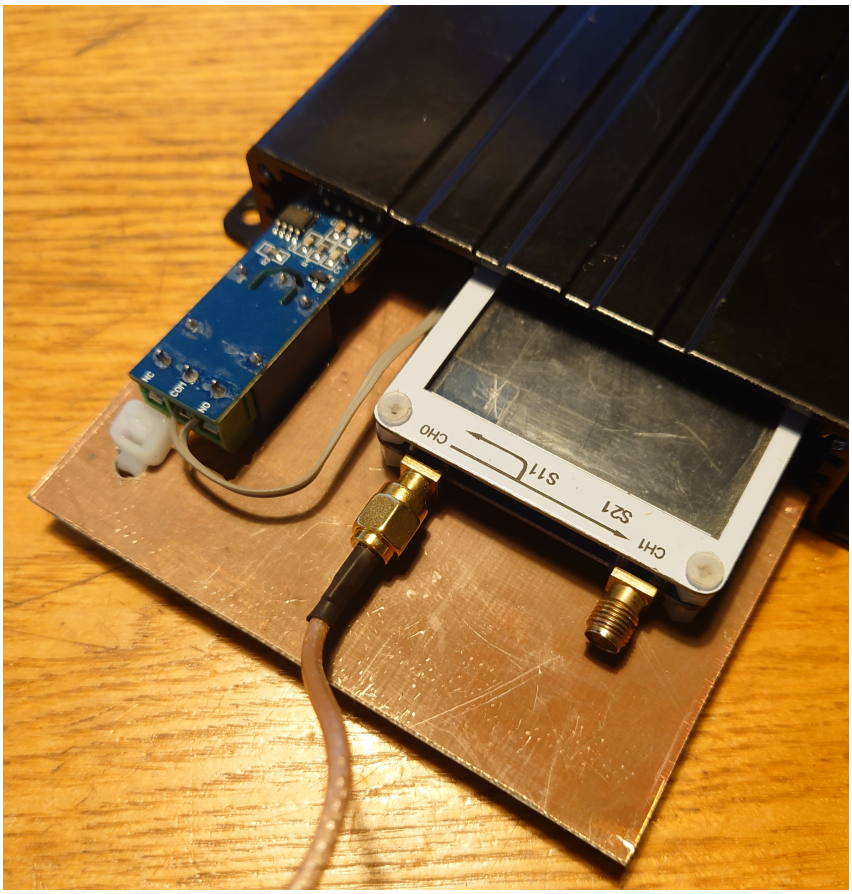
When in self-test, continue to rotate the antenna as normal, and check that the test signal polar plot points towards where the tester antenna is mounted. This proves that all the receive chain is working properly, and that there are no excess timing delays in the SDR system.

The front system can also request the self-test, using the in-dash app:



Note here the test frequency is set to 20kHz above fox1296.0Mhz on the bottom left self-test button. It displays the test frequency to remind the front system operator to tune up 20kHz (easily done with 2 clicks on 10KHz). Ensure the direction from the front system points back to the tester antenna.

Note: This does assume both systems are hunting the same band fox, which is usually the case, particularly at the start of a hunt, or when things are going pear shaped.



Tester software complications

During software development, which is done in the QT environment, I ran into a problem with USB enumeration. You can't predict what tty port number the USB relay serial emulation is going to end up on, as this depends on the order the USB devices enumerate, which can be a bit random. I also have another USB Serial device plugged into the SDR PC. Luckily there is a solution to work around this! You can direct the enumeration process to also provide a named device that is linked to the tty port based on information gleaned from the USB device, such as the manufacturer ID. My software therefore now talks to these named devices (that I name), and the enumerator provides a link to the actual tty port once it is plugged in.

There wasn't an issue with the NanoVNA Serial port, as this enumerated to ttyACM0, which was unique in my system.

When programming two buttons (one local, one remote) that access the same feature, I have to be careful to not create an infinite message loop!

The local interface on the SDR PC is a Linux target, whereas the in-dash screen app is an Android target. They communicate using UDP via the in-car WiFi, but I have my own self-discovery arrangement so we don't have to program in specific IP addresses.

The NanoVNA I have is one of the earlier 0-300MHz versions. Later firmware/hardware VNA versions exploit harmonics of the oscillator to push the upper frequency limit above 2GHz. They deliberate overdrive some gain stages to promote more harmonics. It's possible I could benefit by upgrading my NanoVNA firmware, but as it's an early PCB, I could equally end up bricking it! Instead, I have found the harmonic content is quite high enough as-is for my self-test purposes. For 70cm I program in $^1/_3$ the desired frequency (and a 100Hz span), and similarly for 23cm $^1/_9$ the frequency with a 33Hz span.

The foxhunt tester can be demonstrated, when convenient, on the VK3SOC foxhunt Pajero.

Bruce, VK3TJN

References:

1. Appendix 1: Investigating SDR delays
2. Appendix 2: NanoVNA Console Commands
3. QT environment: https://doc.qt.io/qt-5/
4. Appendix 3: USB relay commands
5. GoogleGroup: gqrx
6. SERG Homebrew entry 2018, New Foxhunt rotator, VK3TJN
7. SERG Homebrew entry 2019, A Software Defined Radio for Foxhunting,VK3TJN
8. NanoVNA: https://nanovna.com/

June Update

Some late additions to the entry!

- I built a special bull-bar mounting antenna, adapting a wire-wound whip found in the junk box. A mast elevates the antenna to the plane of the direction finding antennas. This antenna is as far away as possible but still be mounted on the car, to avoid near-field effects.
- I found that starting a test perturbed the polar plot display, with it initially pointing the wrong way (!!), but gradually catching up to point the right way, right at the test antenna. That issue turned out to be some sleep delays inserted in the code to program the tester properly when it starts up, holding up the timely processing of the input RSSI samples. The code is designed to gradually “eat up” any sample backlog, so it recovers in time, but this wasn’t a good result for what is meant to be a quick test. There are a few solutions to this – like starting a new thread to do the tester programming, but instead I just converted the tester startup sequence to “fall though” state machine programming.
- I found that sometimes, after a test, the usb relay device became unresponsive, and I had to unplug the tester and plug it in again to recover. At first I thought it might be due to insufficient software clean-up on the usb-serial port; but no, all was fine there. Then maybe timing issues? nope, more delays don’t help. Perhaps the long USB cable causing hardware voltage drop problems? plugging it into the USB port direct didn’t help. I’ve eventually just come to the conclusion the usb relay is just a bit flakey. My work-around is to call a script that does a full usb reset on a device after each tester run. The reset works best not on the usb relay itself, because that could become inaccessible - in order to perform the reset, but on the USB hub within the tester. That then causes all connected devices to re-enumerate on the USB bus, and all is fine.

Appendix 1:

Investigations into SDR delays

System Setup

Our rear motor rotated system uses an SDR receiver. The hardware is an SDRPlay RSP2. The software is my own modified version of the GQRX program. This has been covered in my previous Homebrew entries 2018, 2019.

The system doesn't care how long the delay is through the SDR receiver, as long as that delay can be easily characterised and, on average, does not change.

The best way to characterise the delay of the whole system is an end-to-end test. I setup a test signal and tuned it in. Then very very slowly (hand) rotate the beam to create a polar plot. This very slow rotation basically ignores the system delays as they are insignificant compared to the rotation speed, so this can be interpreted as a true polar plot.

Then rotate the antenna at the highest speed that will be used foxhunting (and this can be quite high, to allow for morse transmissions where the dah is still fairly short). Increase the delay programmed into the system until the polar plot at high speed points in the same direction as the slow speed plot. This delay is the total system delay. If this delay is set to zero, the polar plot will skew in a direction the opposite of the antenna rotation direction (ie. lagging), the skew proportional to the rotation speed.

Extra Delay?

Now in the scenario experienced on the two mis-directed hunts, it seemed clear the system delay had somehow changed, skewing the polar plot direction (in one case almost 180 degrees!).

The first checks were to ensure that the programmed delays were still correct, but all seemed to be correct. Then I checked and measured the average delays from the GQRX output signal strengths to the displayed polar plot, and though this is complicated by the multiple CPUs involved and associated serial communications, it seemed there were no expanding buffers or leakages happening there. So it seemed the issue was with the actual SDR, somewhere!

SDR the culprit?

My first hint came from a posting in a Google Groups GQRX forum:

I am observing significant latency creep (~1s per hour) in the buffer between a Perseus (USB) SDR front end and gqrx.This latency seems to be between the radio hardware and the FFT/waterfall,

in that after tuning the hardware front end, the delay is visible in the FFT/waterfall changing. Delay in the audio path seems much more reasonable (i.e. tuning the DSP passband has a fairly immediate affect on the audio output).
This latency does not clear when I stop and restart gqrx DSP - it only goes away when I completely exit gqrx and re start the program (which restarts the Perseus front end as well).

And a reply:

I have experienced much the same in the recent past, with GQRX on Linux (LMDE 5) working with a RSP1 SDR.

And:

The RSP1 and RSP2 have serious timing problems - Mine lose 12 seconds every twenty minutes.

Replication

A full antenna setup is just way too complicated to easily test for *changing* delays.

I setup a test jig to attempt to replicate the reported delays. My first attempt triggering a handheld was too unpredictable, as the handheld had quite a longish TX ramp up, long compared to the delays I'm actually trying to measure. I ended up using a small foxhunt transmitter setup to transmit for 50ms every 5s. The TX-enable is also fed into a USB Serial DCD line. The software is setup to time from the DCD change till the S-meter readings (from GQRX) exceed a set threshold.

This jig showed about **150ms** $^{+}/_{-}$ **50ms** in normal circumstances. The 50ms variability is more due to

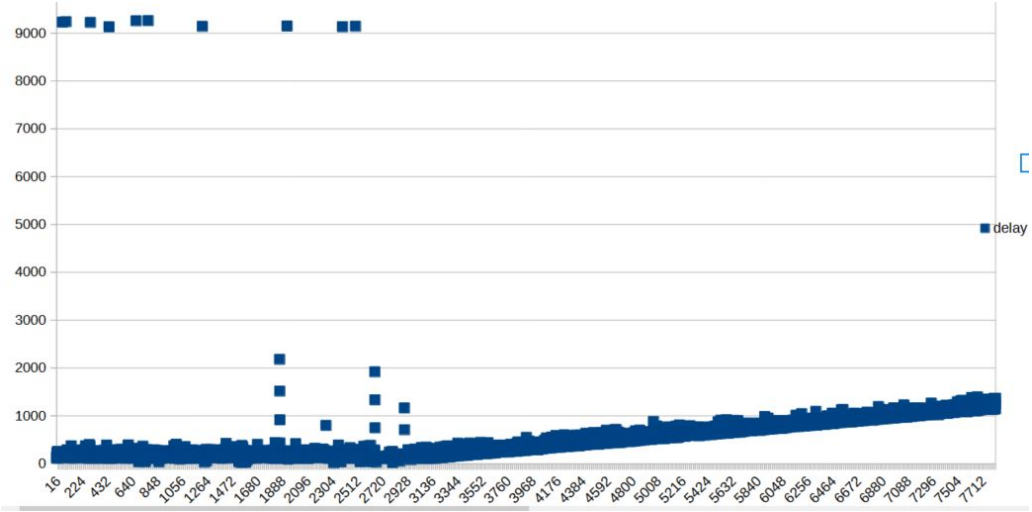
software latency variations in detecting the DCD transition and the GQRX somewhat blocky output stream, but the *average* delay is more what I'm interested in here.

After leaving it running overnight, I **did** observe an increase in the average delay, for example **1000ms extra**! It was difficult to reproduce the effect reliably, and I wasn't sure if it was a gradual change, relating to mismatched clocks, as hinted to above, or sudden steps, perhaps due to the screensaver activating, or something else?

The other oddity I observed was not only was there a step increase in the delay I was measuring, from transmission to GQRX S-meter (in effect, the same point as the waterfall observer above), but also, quite independently, I could sometimes hear an added delay between "the waterfall" and the audio output, which could add yet another second!

My discoveries so far had no feedback from the GQRX group, once reported, whatsover :(

I needed to somehow see how the delay was behaving over time. I added software to output each measured delay (every 5s) to append to a file, and then used LibreCalc to plot the data from overnight again (ignore the 9s readings – just missed transitions):



This was exciting!

Here I can observe clock drift directly. Note how the delay doesn't start to ramp up till around 3000 (3 hours) in. This can be simply explained. If the clock on the RSP hardware is drifting slower than the CPU sample clock used by GQRX, then GQRX will just experience very occasional underruns, which won't be noticed. Only if the RSP clock drifts faster than the CPU clock will buffers potentially start filling with extra samples.

It seems that GQRX simply wasn't designed all that conservatively, taking potential clock drifts into account. Starting from scratch, it is quite easy to either design in a slight oversample, or a simple semaphore protected reclocking block. In fact I do this myself to synchronise between the two processors mentioned earlier, dealing with the polar plot. It seems there are three potential clocking domains involved (RSP clock, CPU clock, sound-card clock).

Again resounding silence from the GQRX group. Oh well. Possibly over most heads?

Note above in the original statement: *This latency does not clear when I stop and restart gqrx DSP*

That comment sent me on quite a goose chase, thinking that the *expanding* buffer must therefore be in the RSP Linux driver (provided as a binary driver by SDRPlay, so the source can't be viewed) or perhaps in the USB subsystem. I chased up SDRPlay forums, but got nowhere. The Linux driver isn't particularly supported anyway.

It was only later I noticed, that for me, at least, restarting the GQRX DSP *did* remove the excess delay. So the *expanding* buffer was *within* GQRX afterall. That made more sense really, as other SDR programs didn't appear to be reporting delay issues.

Solutions

So, I had some choices:

1. Crawl through GQRX code looking for potential cross-clocking violations
2. Port everything over to another open source Linux SDR program
3. Force a periodic DSP restart on GQRX, resetting any growing delays

Both 1 & 2 represented a heap of work. Option 1 was messy because GQRX used many gnuradio sub-blocks and about 4 layers of hardware drivers. Option 2 as I'd spent significant hours customising GQRX for foxhunting, in particular adding a coherent signal S-meter (based around the original DSP algorithm used to process the FT290R audio used prior to 2018).

Option 3 it was! I added a new remote control function to my custom GQRX to not only cycle the DSP, but to restart the whole RSP via USB, just in case. In my own polar plot program, a timer calls this function every 5 minutes, or each time you change the memory to change to another fox leg. The drift within 5 minutes should be negligible.

Since then, we have not experienced a mis-direction hunt. Of course there are many reasons things can still go awry on a foxhunt, so the self-test is a means to quickly try a full end-to-end test. Hopefully, however, we won't be seeing clock drift issues again.

Further observations

The fact the two hunts we saw mis-direction on were the first hunts of the evening is significant. There is quite some time from when the SDR is first setup here in the driveway (say as early as 6pm), till the first hunt at perhaps 8:30pm. This is enough time for drift to have built up significant buffering, hence delay. If we rotate the antenna at say 2Hz, a 90deg error is only 125ms extra.

The temperature of the evening could also be having an effect. From the plot above, we can see the clocks can drift either way relative to each other. It would seem one clock source may be more temperature dependent than the other, so as it got cooler overnight in my test, we ended up with excess samples. Were we more susceptible to drift in winter?

The future

If there appear to be other compelling reasons to move to another SDR program, I may eventually do the porting, but for now GQRX will continue to serve.

USB Relay Commands

USB Relay Module USB Intelligent Control Switch USB Switch for LCUS-1 Type8941

DESCRIPTIONS

Protocol:
LC USB switch default communication baud rate: 9600Bps;
LC USB Switch Communication Protocol:
Data (1)----starting identifier (default: 0xA0)
Data (2)----Switch address code (default: 0x01, identified as the first switch)

Data (3)----Operation data (0x00 is "OFF", 0x01 is "ON")
Data (4)----check code
Example:
TURN ON: A0 01 01 A2
TURN OFF: A0 01 00 A1
HOW TO USE:
1, plug the USB relay into the computer, install the CH340 to serial port chip driver;
2, open STC-ISP, SSCOM32 and other serial debugging software, select the baud rate of 9600, send "A0 01 01 A2" turn on relay in hexadecimal (hex) form; send "A0 01" in decimal (hex) 00 " TURN OFF relay;
USB intelligent control switch.
Onboard high-performance micro-controller chip.
Board CH340 USB controller chip.
Onboard Power LED and relay status LED.
Onboard 5V, 10A / 250VAC, 10A / 30VDC relay, relay life long continuous pull 100,000 times.
Relay modules with overcurrent protection and freewheeling diode protection.
colour:blue
Material:PCB
size:49.6*16mm

Package Contents:
1 x USB relay module