Maximum Supported Hopping Rate Measurements using the Universal Software Radio Peripheral Software Defined Radio

Richard Bell RICHARD.BELL1@NAVY.MIL
Space and Naval Warfare Systems Center Pacific (SSC Pacific), 53560 Hull St, San Diego, CA 92152 USA

Abstract

The goal of this paper is to quantify the maximum usable frequency hopping rate for a variety of Ettus Research Universal Software Radio Peripheral (USRP) software defined radios (SDRs). A table of measured average and maximum hop rates will be presented for the following combinations of motherboards and daughterboards (MB+DB), USRP N210+WBX, N210+SBX, B100+WBX, B100+SBX, B200, X310+UBX.

1. Introduction

Frequency hopping (George & Kiesler, 1942) is a central component of the Link 16 (Gruman, 2014) waveform and as such will be the driving motivation of discussion in this paper. Link 16 is a military tactical data exchange network used by U.S. and NATO Nations. This waveform is used for wireless communication between, air, ground and sea units and relies on frequency hopping to ensure secure and robust data links. The driving question to be answered is: "Can a USRP and personal computer support the hopping rate required by the Link 16 specification?" If the answer is yes, then it is possible to provide low cost surrogates to expensive military hardened radios for laboratory/prototyping use. If the answer is no, but a clear characterization of the rates they support is available, then these systems may be able to support design and testing in some other way.

When designing a new wireless communications system, it is desirable to prototype the system before manufacturing custom hardware to minimize design costs. With cost minimization in mind, it is also desirable to use only general purpose processors (GPPs), avoiding field programmable gate arrays (FPGAs) when possible, due to the high level of expertise needed to design/implement/test/maintain FPGA systems. Today this is possible using SDRs like the Ettus Research USRP (Ettus, 2016), BladeRF (BladeRF, 2016), HackRF (HackRF, 2016), etc. and signal processing software packages like GNU Radio (GNURadio, 2016), MATLAB/Simulink or LabVIEW.

In theory, any communications system can be implemented in a GPP. It is simply an exercise in programming. What prevents us from doing this to all systems by default is the requirement that they operate in real time. For example, suppose you are trying to fly a remote controlled airplane. If the time between when you command the airplane to turn right by moving a joystick to when the plane actually turns right is 3 seconds, then you will have a hard time not crashing the airplane. As a second example, think back to news broadcasts where a correspondent is in a remote area using satellite communications to report back to a live news broadcaster. There is typically a 3-5 second delay between when the correspondent speaks to when they are heard on the television. This often leads to the broadcaster and correspondent speaking over each other and makes coherent two way communications more difficult and time consuming. Thus, for analogous reasons, one might be forced to manufacture custom hardware to avoid these pitfalls. Typically, an all GPP system will be less expensive than a GPP with FPGA system, and this will be less expensive than an application specific integrated circuit (ASIC) system, unless very large production quantities are expected (think iPhone) (Adams, 2002).

What is being detailed in the above two examples is the age old trade off between performance and cost in designing systems. Typically as engineers, the optimal design is the design that just meets a specification. Over-designing the system leads to increased cost, which one typically wants to minimize. Though an FPGA or ASIC based system will have better performance than a GPP based system, if the GPP based system can meet specifications, it will also minimize costs, and therefore should be considered the optimal solution (Farrell, 2009).

Link 16 is a complicated waveform. Among its various specifications is a nominal hop rate greater than 77 kHz. The term "hop" is used to refer to the process of changing the carrier frequency from frequency $X = 1000$ MHz to frequency $Y = 1003$ MHz, for example. Standalone military Link 16 radio units can cost between $100,000 and

Proceedings of the 6^{th} GNU Radio Conference, Copyright 2016 by the author(s).

$1,000,000 dollars, depending on the desired application and features. If an SDR like the USRP, which costs between $2,000 and $5,000 can support the required hop rate, then one can remove the high costs of production radios from test scenarios and test the waveform against new usage scenarios, requiring only software updates that someone familiar with C++ or Python could implement. A typical question that arises in wireless communications systems that would be easier to answer with an SDR GPP only setup is: "How will a specific interference source affect our current radio implementation, and can we do something to fix it if needed?" Answering this question is simplified when there is only a GPP in the loop because FPGA development is a much less agile process (Doyle, 2009). Therefore, with this as motivation, we wish to answer the maximum hop rate question for several commercially available and low cost SDRs.

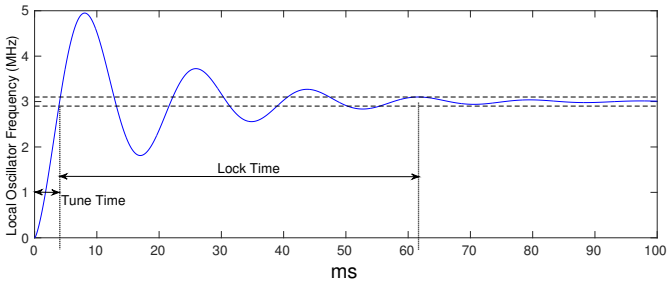


Figure 1: An example of how t_{tune} and t_{lock} are defined. The scale of the two values shown in the figure should not be assumed typical in any sense, they depend on the systems transfer function specifics.

Retuning a radios carrier or center frequency can be achieved in either the RF/Analog domain, the DSP domain, or a combination of the two. For reasons related to the size of the Link 16 band, focus will be directed at RF/Analog retuning in the following discussions. Some comments and results related to DSP tuning will be made.

When retuning a radio to a new frequency, there are several phases that must occur before data can be transmitted. These will be represented using two times. First, the radio frequency (RF) front end analog circuitry must change its current local oscillator (LO) frequency, which is instantiated as a phase locked loop (PLL), to the new frequency. The time required to change frequencies will be called the tune time, t_{tune}. Second, the PLL must declare it has locked at the new frequency before the radio can begin transmitting data, which is called the lock time, t_{lock}. Fig. 1 shows a typical PLL frequency verses time plot when an input of 3 MHz is applied. What these variables represent and how they can change depending on PLL settings is made clear from this figure. The band of frequencies between the dashed lines in the figure represent a lock zone. As soon as the amplitude enters this zone and does not leave again, lock is declared. This is a standard definition used in control theory (Dorf & Bishop, 2016). For more information on how the lock zone is defined as it relates to the RF daughterboards used by the USRP, please see (Forbes & Collins, 2006).

The dwell time, t_{dwell}, is a user defined quantity which determines the frequency hopping rate. It represents the amount of time the center frequency remains, or dwells, at each frequency. Therefore

$$f_{hop} = \frac{1}{t_{dwell}}\,, \tag{1}$$

where f_{hop} is the frequency hopping rate. For a hop to be successful, the CPU processing time, t_{cpu}, plus tune time, t_{tune}, plus lock time, t_{lock}, plus data packet time, t_{data}, is required to be less than the dwell time, t_{dwell}, or else the next hop will be missed

$$t_{cpu} + t_{tune} + t_{lock} + t_{data} \leq t_{dwell}\,. \tag{2}$$

To support Link 16, this imposes the following restriction on the tune time and lock time of the particular radio that is used

$$t_{tune} + t_{lock} \leq t_{dwell} - t_{data} - t_{cpu}\,. \tag{3}$$

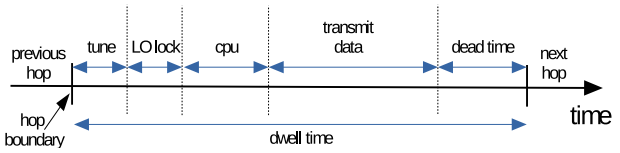
As mentioned earlier, Link 16 requires a hop rate of at least 77 kHz, corresponding to a dwell time of $1/77000 = 0.000013$ seconds, or 13 μs. Using a Link 16 single pulse data packet, which has a length of 6.4 μs, this leads to $t_{dwell} - t_{data} = 6.6$ μs. Therefore

$$t_{tune} + t_{lock} \leq 6.6 - t_{cpu}\,, \tag{4}$$

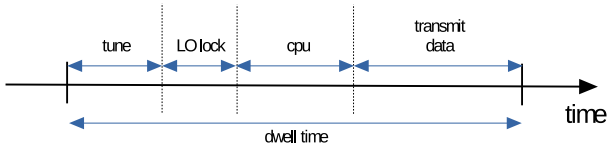
where time units are expressed in μs.

The quantity t_{cpu} represents the overhead time of the CPU processing commands (i.e. the processing done by your laptop or desktop). In the case of a true Link 16 transmitter implemented as an SDR, this would almost entirely represent the baseband processing time required to turn raw binary data into the inphase/quadrature (I/Q) data the USRP requires as input. Because we are interested in characterizing the SDR hardware constraints, and not the processor that is being used, a simple baseband tone that requires very little baseband processing will be used as input. Thus, t_{cpu} will represent the overhead of issuing simple commands, processing conditional if/else statements and calls to timer functions. It is expected in this case for t_{cpu} to be very small, almost negligible. This will be quantified more in the next section.

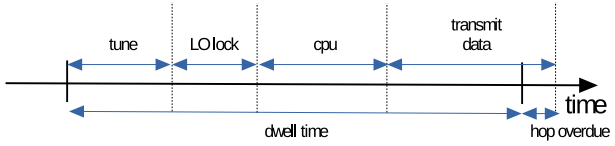
A visualization of when a hop is successful verses when it is not successful is shown in Fig. 2. The critical task is to transmit a full data packet, labeled transmit data in



(a) Successful Hop



(b) Successful Hop



(c) Missed Hop

Figure 2: Three timelines describing three different scenarios. In (a) the CPU time plus retune time plus lock time plus data time is less than the dwell time. In (b), they are equal. In (c) they are greater than the dwell time.

the figure, in the current time slot, which is defined by the hop boundaries. The length of transmit data is predefined ($6.4\mu s$) and static and where it occurs in the slot does not matter, so long as it is fully contained in the slot. In Fig. 2(a), the transmit data begins and ends in the slot with time to spare, labeled dead time. While in the dead time region, the SDR is idle waiting for the next hop time to occur. In Fig. 2(b), there is no dead time, but the hop is successful, just barely. In Fig. 2(c) the data packet spans two time slots, which means the hop was not successful and this information is lost.

The times in Fig. 2 have been presented in a serial fashion, as though retune must complete before LO lock can begin, which happens to always be true, but subsequently that the CPU cannot begin processing before the LO has locked, which is not true. It is understood that some of these times may be occurring in parallel and the Python script will take this into account naturally. For the sake of explaining the process clearly, the times are presented as they have been.

2. Approach

GNU Radio, which is an open source signal processing tool that is used to implement communications baseband processing, is used to interface with and control the USRP hardware via the Universal Hardware Driver (UHD) and a Python script. The Python script is responsible for the following:

1. Instantiate a GNU Radio flowgraph connecting a signal source object to a USRP object. All essential radio settings are set at this point, including antenna port, transmit gain and sample rate.
2. Start the flowgraph, sending samples to the USRP transmitter. This flowgraph is given its own thread with priority over non-essential system processes, so processing time should not be effected by the next steps.
3. Start a timer and command a retune of the USRP transmit center frequency to the next hop frequency.
4. Wait for the local oscillator to lock at the new frequency.
5. Find the difference between the current timer value and the timer value in Step 3.
6. If the timer difference is less than the dwell time, wait until dwell time has elapsed and command the next center frequency, otherwise stop the simulation and declare the hop rate unsupportable.

Before an evaluation of whether the USRP can meet the hop rate required by Link 16 can be made, an estimate of t_{cpu} in Eq. (4) is needed. First an estimate of the smallest unit of time that can be reliably measured by our processing system (the OS and laptop/desktop hardware) will be made. Ubuntu 14.04 LTS on HP EliteBook 8570w with Intel Core i7-3820QM processor and 16 GB RAM was used to make all measurements. To estimate t_{cpu}, the following Python code is used

```
import numpy as np
import timeit
numRealizations = 1000
step1 = np.zeros(numRealizations)
for nn in xrange(numRealizations):
    start_timer = timeit.default_timer()
    time.sleep(0.001) #time in seconds
    step1[nn] = \
        timeit.default_timer()-start_timer
```

A sleep statement is used in the loop so that the timer function will always return a nonzero value. Without this

sleep, the timer often returns 0.0 seconds, which is unrealistic. The sleep statement itself introduces its own source of error in that the computer is estimating 0.001 seconds. For our purposes, as will be shown, it is good enough. We also tested this using the time.time() and time.clock() function calls, both from the Python time module. The time.clock() function measures processing time of the CPU itself (on Linux, Windows may be different), meaning, a sleep time of 1 second would only produce a very small number using time.clock(), because it is not measuring wall time elapsed, it is measuring the amount of time it actually took to execute a sleep statement, which is independent of the argument passed to it. Thus, we will choose not to use time.clock(). The timeit module produced estimates very close to the time.time() function call, but the time.time() function has a consistently smaller standard deviation, though not by a lot ($2\mu s$).

Inspecting Table 1, we see that the two different modules perform similarly over a large number of runs. The table was populated by making 100,000 elapsed time estimates using the aforementioned block of code. Though the values given in the table may lead you to conclude that time.time() should be used, there is not a large enough difference between the two to justify losing the portability of the timeit module.

Table 1: Minimum time resolution statistics using time.time() and timeit.default_timer(). The sleep time was set to $1000\mu s$.

Python Module	Mean (μs)	Std Dev (μs)	Max (μs)	Min (μs)	Max/Min Diff (μs)
time	1093	19	1585	1004	580
timeit	1091	21	2493	1014	1477

The difference between the min and max for each module is surprising and serves to illustrate another important point. The purpose of an Operating System (OS) is to allow the physical CPU to be shared amongst several processes in parallel, creating a schedule that determines which process gets to use the CPU and for how much time. In the course of running programs, the OS will weave processes (and threads) in and out thousands of times per second, creating what appears to be seamless parallel operation from a users perspective. In reality, the parallelization is only as high as the number of cores on the machine. If we assume a single core CPU without hyper-threading, there is no parallelization taking place and yet a user would still be able to listen to music at the same time as downloading files and browsing the internet. This is because the OS and more specifically the scheduler, is switching back and forth between tasks so fast it makes it look like parallel processing is occurring, when in fact it is all serial.

Normally, this is exactly how a user wants the OS to behave. Unfortunately, when attempting to make precise measurements for a single process as is being done here, it is preferred that the OS not switch out our process for another until our process has completed. The OS should put everything else on hold until the radio process is completed. This level of control is not possible using a standard OS such as Ubuntu. The best that can be hoped for is to minimize the effect by assigning higher priority to the radio processes and threads. It is believed that these large outliers are accounted for by this fact. The script took 331 seconds to run through 100,000 iterations and in this time, the OS may have given other processes control of the CPU, forcing ours to idle and causing the large time difference to occur.

Given the results of Table 1, we can conclude that to within 97% confidence the timeit.default_timer() resolution will be within six standard deviations of the mean, or $21 \times 6 = 126\ \mu s$, ignoring the thread priority outliers. We use Chebyshev's inequality (Stark & Woods, 2002) to make this statement, which holds for any arbitrary underlying distribution, not only normal (Gaussian) distributions. If the underlying distribution was known to be Gaussian, this result could be improved upon. This lets us conclude that our processing system (the laptop/desktop portion) will surely not support Link 16, because it requires on the order of microsecond resolution. It is also noted that one cannot hope to measure t_{tune} or t_{lock} to any time finer then $126\ \mu s$. When designing a processing system for Link 16, it would be desirable to use a real time operating system (RTOS) with sub microsecond timing capabilities. It is unknown at this time whether the many free RTOS[1] options would perform well enough for Link 16.

Let us now move on to estimating the tune time and the lock time of the USRP. Retuning the USRP is accomplished by calling the following command

```
tb.usrp.set_center_freq(target_freq)
```

and determining when the USRP locks is accomplished by calling

```
tb.usrp.get_sensor("lo_locked")
```

Using these commands, a timer is started right before calling *set_center_freq()*. A while loop is then entered and the program remains here until *get.sensor("lo_locked")* returns true. Upon leaving this while loop we measure the elapsed time between the start of the timer to when the LO locked,

[1] PREEMPT-RT is an example of an RTOS Linux kernel patch that can be applied to standard Linux, see https://www.linux.com/blog/intro-real-time-linux-embedded-developers. This is not the only solution

which gives us our estimate of $t_{tune} + t_{lock}$, to within the 126 μs accuracy imposed on us by our processing system.

3. Results

All results were collected using Ubuntu 14.04 LTS, GNU Radio 3.7.10 with Universal Hardware Driver (UHD) 3.9.3. UHD is the driver that interfaces the processing system (laptop/desktop) to the USRP. The USRPs tested were also imaged with the latest firmware and FPGA source distributed with UHD 3.9.3. To reduce the dependence of our results on the OS scheduler, the GNU Radio real time scheduling option was used when running flowgraphs. This option increases the priority of threads related to GNU Radio flowgraphs ensuring trivial processes do not introduce a context switch at a bad time resulting in large unnecessary timer values between hops. In addition, the Unix *nice -n 20 python program_name* command is used to ensure that the top level Python script that controls and starts the flowgraph thread and starts and stops timers is also given priority above non-essential OS processes. With this said, there are still uncertainties related to the Linux scheduler which cannot be overcome without using an RTOS. Any large outliers may be attributable to this.

The maximum average hop rate, instantaneous hop rate and tune and lock times for various USRP combinations are listed in Table 2. The HackRF is an open source RF front end SDR unaffiliated with Ettus Research and the USRPs and is included in the table for comparison. The maximum average hop rate was determined by finding the largest frequency that supports 1000 hops without missing a single hop. The maximum instantaneous hop rate was determined by finding the largest frequency that supports 1000 hops with at least one hop being successful. The hop amounts used to create this table were randomly selected with a minimum of 3 MHz, and a maximum of 27 MHz. These values give a sense of the bounds of the hardware platform. Between these rates, the radio was able to complete some portion of the hops successfully. To visualize the hop rates supported across all platforms, see Fig. 3, which allows for visual interpolation of hop rates between the maximum average rate and the maximum instantaneous rate. See Table 3 for a list of missed hop percentages at a given rate for the USRP N210 WBX combination.

The retune times ($t_{tune} + t_{lock}$) and lock times (t_{lock}) varied from hop to hop, as can be seen in Fig. 4 and Fig. 5. The retune time average appears to be 500 μs, with 90% of the times contained between $350 - 650$ μs and the lock time average appears to be 300 μs, with 90% of the times contained between $200 - 450$ μs . Surprisingly, there are large periodic outliers as large as 1100 μs, repeating approximately every 75 hops. They can be seen in both the

Table 2: Hop rate measurements for various USRP SDR combinations and a HackRF. The times and maximum average hop rate are average values over 1000 hops. The final column is the maximum hop rate encountered for a single successful hop.

MB	DB	$t_{tune}+$ t_{lock} (μs)	t_{lock} (μs)	Max Avg (Hz)	Max Inst (Hz)
N210	WBX	510	290	500	2000
N210	SBX	483	289	500	2000
X310	UBX	610	418	500	2000
B100	WBX	782	573	500	1666
B100	SBX	772	546	333	1666
B200[1]	-[4]	3310	117	166	285
B210[1]	-[4]	3318	124	166	285
E310[2]	-[4]	31258	285	28	31
HackRF [3]	-[4]	215	0	1428	5000

[1] The B200/B210 uses an integrated RF front end which was not designed for fast hopping

[2] The E310 is an embedded device (self contained, no host personal computer) running its own version of Linux on a Xilinx System On Chip (SoC) Zynq processor.

[3] The HackRF does not have the ability to measure LO lock time therefore its hop rates are inflated and do not represent usable hop rates. The HackRF usable rates are likely similar to those of the USRPs.

[4] These SDRs do not use daughterboard (DB) cards

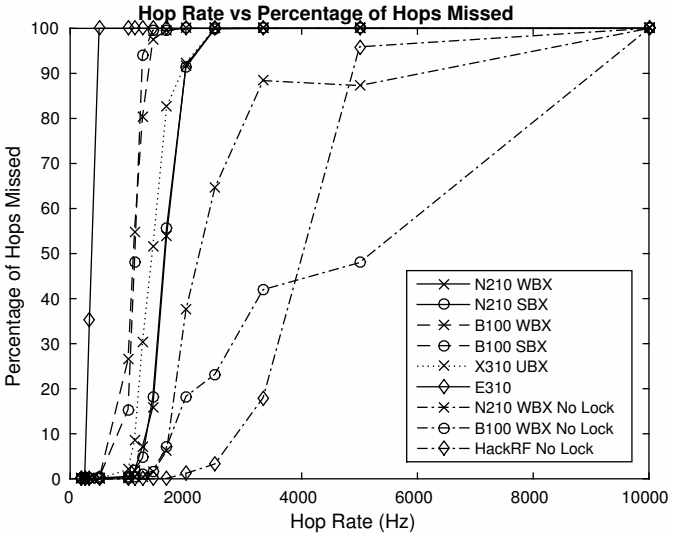


Figure 3: Hop rate verses percentage of hops missed. Note that the HackRF does not have a feature to measure time to lock built in, so the usable hop rate is likely similar to that of the USRPs. For comparison, plots of the N210 and B100 were added ignoring the time to lock.

tune times and lock times. It is mentioned in the UHD[2] manual that the lock time is not deterministic, which could be the source of these spikes. Because of the periodic nature of the spikes, we believe the source to be either completely caused by the processing system or completely caused by the USRP, because it would be very unlikely that both would line up so perfectly over time. The exact cause of these outliers is unknown at this time.

Table 3: This table shows the hop rate verses percentage of hops missed for the USRP N210 with WBX daughterboard combination.

Hop Rate (Hz)	Dwell Time (μs)	Hops Missed	Percentage Missed
200	5000	0	0
250	4000	0	0
333	3000	1	0.1
500	2000	2	0.2
1000	1000	5	0.5
1111	900	18	1.8
1250	800	72	7.2
1428	700	159	15.9
1666	600	540	54
2000	500	918	91.8
2500	400	1000	100

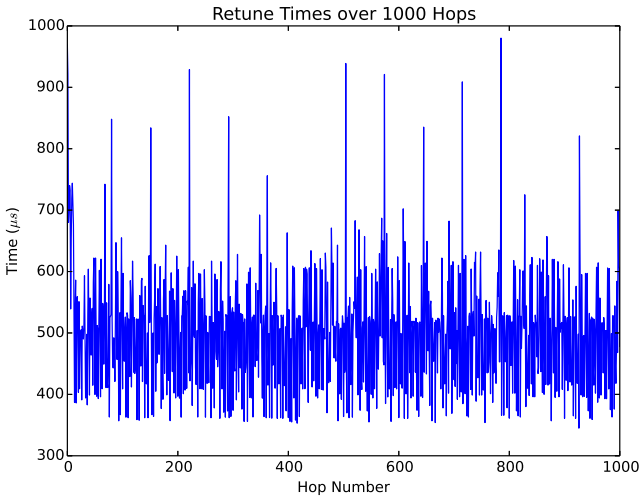


Figure 4: Retune times, $t_{tune} + t_{lock}$, for each of 1000 hops.

The Link 16 specification defines a total range of hopping frequencies that span nearly 300 MHz. For this reason, full RF retunes were characterized in the preceding discussion, because it would not be possible with current technology to rely on DSP retunes only to cover this range. Inevitably an RF retune would be needed and this would become the bottleneck. For completeness it is mentioned that hopping using DSP only retunes is possible if the hopping range is within the sampling rate of the ADC/DAC of your SDR. For example, the USRP N210 has an ADC/DAC sampling rate of 100 MHz. If your hopping frequencies stay within this band, you can rely on DSP only retunes and remove the LO locking time, because the RF frequency never changes.

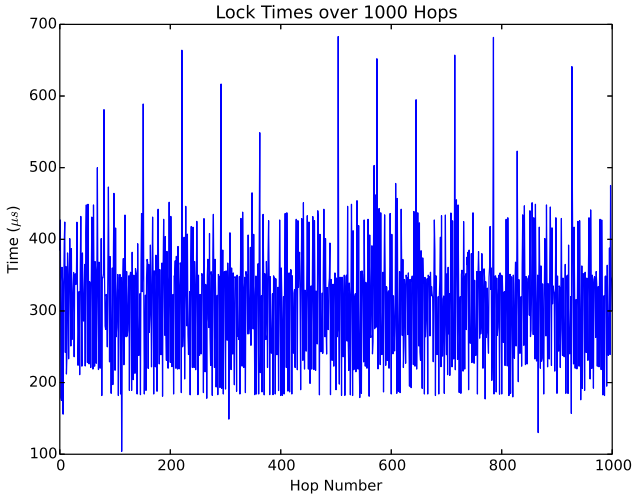


Figure 5: Lock times for each of 1000 hops.

Using DSP only retunes does allow for faster hop rates, showing approximately 60% improvement in the average tune plus lock time. However, there were very large anomalies in the tune plus lock time that occur using DSP only retunes that did not occur when using RF retunes. The cause of this is unknown. In addition, there were spurs present in the output spectrum of the DSP only retune that were not present when RF retuning was used.

4. Conclusion

It has been shown that a modern high end computer does not provide the time resolution required by the Link 16 waveform. The computer characterized in this paper was shown to have uncertainty in time resolution of up to 126 μs, far too large when deadlines on the order of 6 μs are required. It is believed that the results found using our computer are a good representative of the population of modern computers as a whole. The next step would be to evaluate an RTOS on a modern computer to see how much better resolution becomes. It is possible the physical oscillators and hardware used by commerical off-the-shelf (COTS) computers do not offer the resolution needed, regardless of the OS used, in which case a custom hardware

[2]See the "RF front-end settling time section", http://files.ettus.com/manual/page_general.html

solution will be required.

The USRP motherboard/daughterboard combinations tested in this paper show average tuning and locking times in the hundreds of microseconds range. This is too long to meet timing requirements of the Link 16 waveform. As shown in Eq. (4), the tune and lock times need to be sub 6 μs for the SDR to have any chance of meeting the required hop rate. The exact amount of time the tune and lock must fall under depends on the amount of time needed for baseband processing, given by t_{cpu}.

The method used in this paper to implement frequency hopping should be considered the "out of the box" method. No extensive modifications or developer level UHD knowledge is needed to implement the hopping. This is believed to be the way most developers would intend on using the devices and therefore it makes the most sense to characterize. While it may be possible to improve the results here by using custom low level C/C++ scripts, this would require more detailed knowledge of the UHD drivers. At this point, you begin losing your "low cost" solution to engineering development costs in writing the custom scripts. It is also the authors opinion that heavily optimized scripts would still be very far from hopping at the rates required by Link 16 using the USRPs.

It is the opinion of the author that a custom GPP and RF front end solution can be made to support the Link 16 waveform that does not depend on FPGAs. Though this custom solution would be more expensive than a COTS based solution, it would offer tremendous cost savings in maintenance and code revision for future updates/upgrades to the Link 16 waveform. It would also be very portable to COTS based RF front ends as they become available in the future when low cost technology catches up to the rates required by Link 16.

References

Adams, Leon. Choosing the right architecture for real-time signal processing designs. Technical Report SPRA879, Texas Instruments, Nov 2002. URL http://www.ti.com.cn/cn/lit/wp/spra879/spra879.pdf.

BladeRF. Bladerf, 2016. URL http://www.nuand.com/. accessed June 2016.

Dorf, R.C. and Bishop, R.H. *Modern Control Systems*. Pearson Prentice Hall, 2016. ISBN 9780132270281. URL https://books.google.com/books?id=V-FpzJP5bEIC.

Doyle, Linda. *The Essentials of Cognitive Radio*. Cambridge University Press, 2009. ISBN 9780521897709.

Ettus. Ettus research, 2016. URL accessedApril2016,https://www.ettus.com/. accessed April 2016.

Farrell, John Patrick. Digital Hardware Design Decisions and Trade-offs for Software Radio Systems. Master's thesis, Virginia Polytechnic Institute, Blacksburg, VA, 2009.

Forbes, Peader and Collins, Ian. Lock detect for the ADF4xxx family of PLL synthesizers. Technical Report AN-873, Analog Devices, 2006. URL http://www.analog.com/media/en/technical-documentation/application-notes/AN-873.pdf.

George, A. and Kiesler, M.H. Secret communication system, August 11 1942. URL http://www.google.com/patents/US2292387. US Patent 2,292,387.

GNURadio. Gnu radio, 2016. URL accessedApril2016,http://www.gnuradio.org. accessed April 2016.

Gruman, Northrop. Understanding voice and data link networking, 2014. URL http://www.northropgrumman.com/Capabilities/DataLinkProcessingAndManagement/Documents/Understanding_Voice+Data_Link_Networking.pdf. accessed June 2016.

HackRF. Hackrf, 2016. URL https://greatscottgadgets.com/hackrf/. accessed April 2016.

Stark, H. and Woods, J.W. *Probability and Random Processes with Applications to Signal Processing*. Prentice Hall, 2002. ISBN 9780130200716.