AFRL-RV-PS-
TR-2022-0088

SMALL SATELLITE POSITION, NAVIGATION, AND TIMING INNOVATIONS
VOLUME I - CONTACT CLOCK TESTBED

Christopher Flood and Penina Axelrad

University of Colorado Boulder
Aerospace Engineering Sciences, CCAR
3775 Discovery Drive
Boulder, CO 80303

30 August 2022

Final Report

AIR FORCE RESEARCH LABORATORY
Space Vehicles Directorate
3550 Aberdeen Ave SE
AIR FORCE MATERIEL COMMAND
KIRTLAND AIR FORCE BASE, NM 87117-5776

DTIC COPY

NOTICE AND SIGNATURE PAGE

AFRL-RV-PS-TR-2022-0088 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//SIGNED//	//SIGNED//
_____	_____
Dr. Spencer E. Olson	Mark E. Roverse, Chief
Program Manager/AFRL/RVB	AFRL Geospace Technologies Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
30-08-2022	Final Report	4 Dec 2018 – 30 Aug 2022

4. TITLE AND SUBTITLE
Small Satellite Position, Navigation, and Timing Innovations
Vol. I – CONTACT Clock Testbed

5a. CONTRACT NUMBER
FA9453-19-1-0076

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
63401F

6. AUTHOR(S)
Christopher Flood and Penina Axelrad

5d. PROJECT NUMBER
3682

5e. TASK NUMBER
EF134353

5f. WORK UNIT NUMBER
V1NE

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Colorado Boulder
Aerospace Engineering Sciences, CCAR
3775 Discovery Drive
Boulder, CO 80303

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Space Vehicles Directorate
3550 Aberdeen Avenue SE
Kirtland AFB, NM 87117-5776

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RVBYT

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-RV-PS-TR-2022-0088

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited (AFRL-2023-1731 dtd 12 Apr 2023).

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This final report documents the work completed by researchers and students in the Colorado Center for Astrodynamics Research (CCAR) and Smead Aerospace Engineering Sciences at the University of Colorado Boulder, to model and develop technologies and algorithms to advance small space platform positioning, navigation, and timing, with a primary emphasis on timing systems. The report is presented in three volumes. Volume 1 presents the CONTACT software defined radio (SDR) based testbed for measurement and ensembling of low size, weight, and power (SWaP) atomic clocks. Volume 2 describes the development of a CSAC flight experiment to be flown on the MAXWELL UNP-9 CubeSat, expected to be launched in 2023. Volume 3 focuses on modeling and analysis of distributed optical time and frequency transfer across small satellites in a large-scale low Earth orbit (LEO) constellation.

15. SUBJECT TERMS
CSAC performance, clock testbed

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Spencer E Olson
Unclassified	Unclassified	Unclassified	Unlimited	138	**19b. TELEPHONE NUMBER** *(include area code)*

This page is intentionally left blank.

TABLE OF CONTENTS

Section			Page
LIST OF FIGURES			iv
LIST OF TABLES			vii
1	SUMMARY		1
2	INTRODUCTION		2
	2.1	Project Motivation	2
	2.2	Project Description	2
3	METHODS, ASSUMPTIONS, AND PROCEDURES		4
	3.1	Testbed Architecture	4
	3.2	Clocks	5
	3.3	Software Defined Radios	10
		3.3.1 Red Pitaya	12
		3.3.2 N310	15
		3.3.3 N200	20
	3.4	Testbed Software Models	23
		3.4.1 Clock Profiles	23
		3.4.2 Clock Steering	24
		3.4.3 Clock State Estimation	25
	3.5	Programming in GNU Radio Companion	27
		3.5.1 Moving Data and Data Rates	28
		3.5.2 Embedded Python Blocks	29
		3.5.3 USRP Block Options	30
		3.5.4 Loops	30

TABLE OF CONTENTS (continued)

Section **Page**

3.6 Clock Phase Measurements 31
3.6.1 Effect of the Local Oscillator 33
3.6.2 External Reference Oscillator 33
3.6.3 Measurement Noise 38
3.6.4 Clock Phase Measurement Considerations 38
3.7 OCXO Frequency Characterization 39
3.8 Communication Protocols for Time Synchronization 41
3.8.1 NTP 42
3.8.2 PTP 43
3.8.3 White Rabbit 44
3.8.4 Limitations 45
4 RESULTS AND DISCUSSION 46
4.1 Testbed Software Models 46
4.1.1 Clock Profiles 46
4.1.2 Steering Simulation 47
4.1.3 IEM Steering 49
4.2 Experimental Clock Phase Measurements 50
4.2.1 Effect of the Local Oscillator 50
4.2.2 Measurement Noise 52
4.2.3 Measured Clock Stability 54
4.3 OCXO Frequency Characterization 56
4.3.1 Wenzel OCXO & Power Supply 56
4.3.2 NEL OCXO & DAC 57
4.4 SDR with External Reference Oscillator 59
4.4.1 SDR Transmit Test 59
4.4.2 SDR Receive Test 60
4.4.3 SDR Transmit & Receive Test 62

TABLE OF CONTENTS (continued)

Section **Page**

4.5 OCXO Steering to Known Reference 63

4.5.1 Wenzel OCXO & Power Supply 63

4.5.2 NEL OCXO & DAC 64

4.6 Clock State Estimation 65

4.7 Clock Ensemble Testbed Integration 68

5 CONCLUSIONS 71

REFERENCES 73

APPENDIX A - Red Pitaya Data Acquisition MATLAB Code 76

APPENDIX B - Testbed Simulation MATLAB Code 78

APPENDIX C - Clock Signal Generation MATLAB Code 82

APPENDIX D - Clock STM Generation MATLAB Code 86

APPENDIX E - Allan Deviation MATLAB Code 87

APPENDIX F - Simulate Testbed Function MATLAB Code 88

APPENDIX G - Phasor Block 91

APPENDIX H - Phase Unwrap Block 93

APPENDIX I - Ensemble Kalman Filter Block 95

APPENDIX J - OCXO Kalman Filter Block 101

APPENDIX K - DAC Message Block 106

APPENDIX L - Power Supply Message Block 109

APPENDIX M - DAC Arduino Code 112

LIST OF FIGURES

Figure		Page
1	Functional Block Diagram	4
2	Microsemi SA.45s CSAC [11]	5
3	SRS Rubidium Frequency Standard [12]	5
4	Wenzel Associates OCXO [13]	6
5	NEL OCXO [14]	6
6	Orolia Miniaturized Rb Oscillator [15]	6
7	Assembled Interface Board	7
8	Breakout Board Schematic	8
9	CONTACT Breakout Board Layout	9
10	CSAC/Breakout Board Serial Port Communication	10
11	10 MHz Output Comparison	10
12	Red Pitaya [17]	11
13	Ettus N200 [18]	11
14	Ettus N310 [19]	11
15	Red Pitaya [17]	13
16	10 MHz Sampled at 125 MHz - First 125 Samples	14
17	Ettus USRP N310 Front Panel	15
18	Ettus USRP N310 Rear Panel	16
19	Ettus USRP N310 Motherboard Block Diagram [19]	16
20	Ettus USRP N310 Transceiver Block Diagram [19]	17
21	Ettus USRP Tuning with Center Frequency and LO Offset	19
22	Time Series and Power Spectrum with (top) and without (bottom) LO Offset	20

LIST OF FIGURES (continued)

Figure **Page**

23 Ettus N200 [18] 21
24 Ettus N200 Motherboard [18] 22
25 Ettus N200 - BasicRX Daughterboard 23
26 System Response to Phase and Frequency Step Inputs 25
27 USRP Source Block Settings 30
28 USRP Source Block RF Options 30
29 Measurement System Block Diagram - Theory [7] 31
30 Measurement System Block Diagram - Implemented 32
31 Measurement System Implemented in GNU Radio Companion 32
32 SDR Transmit Test Configuration 34
33 SDR Receive Test Configuration 35
34 Simultaneous Transmit & Receive Test Configuration 37
35 N310 with Transceiver 0 Inputs (Blue) and Transceiver 1 Inputs (Red) 38
36 Benchtop Power Supply 40
37 Digital to Analog Converter 41
38 PTP Instancing Between Nodes 44
39 Simulated CSAC Time Series 47
40 ADEV of Simulated CSACs 47
41 Time Series of Steering OCXO to CSAC 01 - 1 hour 48
42 Steered OCXO Curves Minus CSAC 01 - 24 hours 48
43 Time Series of Steering OCXO to CSAC 01 - 24 hours 49
44 ADEV of Steered OCXOs, CSAC 01, and Unsteered OCXO 49
45 Time Series of Simulated CSAC, IEM, and Steered OCXO 50
46 ADEV of Simulated CSACs, IEM, and Steered OCXO 50
47 Measured Clock Phase with Common SDR Clock Contribution 51
48 Measured Clock Phase against a Rubidium Reference 51
49 Allan Deviation of CSACs and LO of Ettus N310 52

LIST OF FIGURES (continued)

Figure **Page**

50 N310 Different Daughterboard Noise....53

51 N200 and N310 Same Daughterboard Noise....53

52 Allan Deviation of SDR Noise....53

53 Stability of Clocks in CONTACT Project....55

54 Voltage / Frequency Relationship for Wenzel OCXO and Power Supply 57

55 Voltage / Frequency Relationship for NEL OCXO and DAC 58

56 CSAC & SDR Transmit Test Configuration ADEV....59

57 CSAC & SDR Receive Test Configuration ADEV - No Reference....60

58 CSAC & SDR Receive Test Configuration ADEV - With CSAC Reference . 61

59 Simultaneous Transmit & Receive Test Configuration ADEV....62

60 OCXO Steering Block Diagram. 63

61 Steering Wenzel OCXO to Chip with Power Supply....64

62 Steering NEL OCXO to Spacebuff with DAC 65

63 GNU Radio Companion Kalman Filter....65

64 Measured Clock Bias with respect to Rb Reference....66

65 Estimated Clock Bias Output from Kalman filter....66

66 Difference Between Measured (solid) and Estimated (dashed) Clock Biases . 67

67 Difference of Clock Bias Differences 67

68 ADEV of Measured and Estimated CSACs 68

69 Clock Ensemble Testbed Block Diagram 69

70 Clock Ensemble Testbed....69

71 ADEV of Steered OCXOs 71

LIST OF TABLES

Table **Page**

1 CONTACT Team Project Participants viii

2 Comparison Between Hardware Platforms 11

3 GNU Radio Blocks 28

4 GNU Radio Embedded Python Blocks Developed for CONTACT 29

5 Measured CSAC ADEVs 55

6 Measured Clock ADEVs 56

Acknowledgements

The authors gratefully acknowledge the specific contributions to the work described in Volume 1 by team members Alex Conrad, Prayag Desai, Daniel Dowd, Justin Pedersen, Rahul Ramaprasad, and William Watkins.

We acknowledge the valuable advice provided Dr. Joanna Hinks and her colleagues at AFRL who attended our presentations and gave helpful feedback for moving forward with the project; and by Dr. Robert Lutwak of Microchip regarding the performance and testing of the CSACs. Additionally, we thank Dr. Nicholas Rainville for providing project guidance; Harrison Bourne and Steve Taylor for SDR advice and network help; Dr. Franklin Ascarrunz of SpectraDynamics and NIST researchers - Dr. Nate Newbury, Dr. Jeff Sherman, and Dr. Stefania Romisch for sharing their expertise, loaning us equipment, and helping conduct early testing of our clocks.

Finally, we want to recognize the hard work of all CONTACT team members shown in Table 1 below, who contributed since this project's inception in Spring 2019.

Table 1. CONTACT Team Project Participants

Last Name	First Name	Position(s)	Dates
Colpaert	Cydnee	MS Graduate Project Team	08/21 - 05/22
Conrad	Alex	PhD Student Volunteer	01/20 - 05/20
Davies	Laura	PhD Student Volunteer	09/21 - 12/21
Desai	Prayag	Independent Study	08/20 - 12/20
Dixon	Caroline	Undergraduate Research Assistant	08/20 - 06/21
Dixon	Henry	MS Thesis, MS Research Assistant	01/19 - 02/21
Dobbin	Mikaela	MS Graduate Project Team, Research Assistant	08/21 - 08/22
Dowd	Daniel	MS Graduate Project Team, Independent Study	08/19 - 12/20
Flood	Christopher	PhD Student Volunteer	08/19 - 08/22
Khatri	Yashica	MS Graduate Project Team	08/19 - 06/20
Krebs	Christopher	Undergraduate Research Assistant	06/21 - 05/22
LaBarge	Quinn	MS Graduate Project Team	08/20 - 05/21
Mezich	Andrew	MS Graduate Project Team	01/19 - 05/19
Morris	Tyler	MS Graduate Project Team	08/19 - 05/20
Nichols	Alexander	MS Graduate Project Team	01/19 - 05/19
Pedersen	Justin	Undergraduate Research Assistant	06/22 - 08/22
Ramaprasad	Rahul	Independent Study	08/19 - 12/19
Reynolds	Zachary	MS Graduate Project Team	01/19 - 05/19
Rybak	Margaret	PhD Student Volunteer	01/19 - 05/19
Schement	Luciana	MS Graduate Project Team	08/20 - 05/21
Watkins	William	MS Graduate Project Team	08/21 - 05/22

1 SUMMARY

This final report documents the work completed by researchers and students in the Colorado Center for Astrodynamics Research (CCAR) and Smead Aerospace Engineering Sciences at the University of Colorado Boulder, to model and develop technologies and algorithms to advance small space platform positioning, navigation, and timing, with a primary emphasis on timing systems. The report is presented in three volumes. Volume 1 presents a software defined radio (SDR) based testbed for measurement and ensembling of low size, weight, and power (SWaP) atomic clocks. Volume 2 describes the development of a CSAC flight experiment to be flown on the MAXWELL UNP-9 CubeSat, expected to be launched in 2023. Volume 3 focuses on modeling and analysis of distributed optical time and frequency transfer across small satellites in a large-scale low Earth orbit (LEO) constellation.

The Colorado Nanosat Atomic Clock Testbed (CONTACT) was a three-year graduate project conducted at the University of Colorado Boulder (CU Boulder) to educate students and advance small satellite timing system technology. The CONTACT team designed, assembled, and operated a testbed to facilitate development of low size, weight, and power (SWaP) approaches to small satellite timing systems. The testbed is based around an SDR metrology architecture which enables clock measurements, clock characterization, clock ensembling, and steered signal generation. The testbed supports evaluation of concepts for timing systems that include the following functionality:

1. Low noise clock phase measurements for free-running clock characterization
2. Using relative phase measurements and Kalman filtering techniques to form a clock ensemble
3. Creating a steered realization of the ensemble time scale for on-board generation of communication or navigation signals

This report provides an overview of the CONTACT project, the hardware, and methods used to create a clock characterization and ensembling testbed using software defined radios. In the following sections we combine low-noise clock phase measurement systems, estimation algorithms, and signal steering techniques to produce an output signal with OCXO-like short term stability and long term stability of the CSAC clock ensemble IEM.

2 INTRODUCTION

2.1 Project Motivation

In August 2019, the DoD publicly released an unclassified version of its "Strategy for the Department of Defense Positioning, Navigation, and Timing (PNT) Enterprise: Ensuring a U.S. Military PNT Advantage." Emphasizing the foundational role of GPS PNT in Joint Force operations, the Strategy warns that "space-based PNT services provided by GPS will be targeted and will not always be available in contested military operating areas...complementary PNT capabilities must be applied." [1]

While GPS is best known as a navigation system, one of its critical functionalities in providing global PNT capability is disseminating precise time, time intervals, and frequency. Communications systems use both time and frequency to maintain accurate carrier frequencies and data-bit phase timing. Secure networks have especially stringent timing requirements for the synchronization of data encryption and decryption equipment [2]. Since the 1950s, the gold standard for timekeeping has been ground-based atomic clocks; which are generally not suitable for spaceflight or ground vehicles due to their size, weight, power requirements, and environmental sensitivity [3]. Space qualified atomic clocks are used in GPS to provide a consistent, accurate, and inexpensive external frequency source to both military and civil users around the globe.

As a part of the "complementary PNT capabilities" aimed at mitigating the inherent vulnerability of DoD dependence on GPS, the Air Force Research Lab (AFRL) is interested in developing accurate on-board timing technologies that are less reliant on frequent GPS time corrections. This report presents work completed by the Colorado Nanosat Atomic Clock Testbed aerospace engineering sciences graduate project team at the University of Colorado Boulder, sponsored by the AFRL Space Vehicles Directorate. The CONTACT project objective was to develop a testbed capable of ensembling three or more low-SWaP atomic clocks to produce an accurate and robust time signal for small satellite applications. Advanced PNT satellites could use this technology to achieve a highly accurate time scale when GPS is unavailable. Satellites in GPS-denied environments with strict position accuracy requirements could also benefit from an improved on-board time reference to support precise orbit determination [4].

2.2 Project Description

The continued development of accurate, robust on-board timekeeping technologies is foundational to the improvement of non-GPS PNT capabilities for a multiplicity of military and civil applications. Complementary PNT enabled by low-SWaP on-board timing systems has

potential to increase the resilience of communication and navigation systems in degraded or denied GPS environments. The limited performance inherent in low-SWaP clocks can be mitigated by ensembling several independent devices and generating a steered output frequency signal based on the weighted averages of the inputs. A testbed was used to evaluate the performance of various clock configurations and signal steering techniques [5] as well as to measure the effects of environmental factors and clock errors. In this report we describe the design and implementation of a clock ensemble testbed on a software defined radio (SDR) hardware platform [6]. Results from this work could be used in the development of low-SWaP on-board timing systems capable of meeting Precise Time and Frequency (PT&F) requirements in the absence of GPS.

The testbed has three chip scale atomic clocks (CSAC) as clock inputs to generate a steered output signal, via an OCXO, and a clock characterization report of the steered signal measured against a Rb frequency standard. The system is currently implemented on two SDRs and programmed with GNU Radio Companion, an open-source software development toolkit. The software radio makes it possible to process signals in the 10 MHz – 200 MHz range with the goal of clock characterization and ensemble formation. SDR oscillator metrology techniques[7] enable precise clock phase measurements while maintaining low on-board storage requirements. The measured phase data are processed to produce Allan Deviations (ADEVs) plots to evaluate the oscillator frequency stability.

In a clock ensemble, phase difference measurements are input to a Kalman filter to estimate the bias and drift of each clock under test [8]. The filter implements a standard two-state clock model and measured noise parameters from clock characterization experiments. The estimated states are the phase and frequency for each input clock - these estimates produce a composite clock timescale by way of a weighted-average ensembling algorithm [9]. The output signal is measured against a rubidium frequency standard to evaluate controller performance and the signal stability with different steering parameters.

Sections 3.1 - 3.3 below discuss the testbed architecture and the hardware used in the project. Theory is developed in sections 3.4 - 3.5 which describes the individual testbed components, how they were simulated in software, and an overview of GNU Radio Companion programming. Section 3.6 describes the process of making clock phase measurements.

Most theory and architecture is outlined in Section 3 with corresponding results in Section 4. All simulated data, clock phase measurements, and steering experiments results are contained in Section 4. System integration and testing is detailed in Section 4.7. The report conclusions are presented in Section 5 followed by references and a set of appendices including source code for key components of testbed software.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Testbed Architecture

The block diagram in Figure 1 shows the separate components of the standalone testbed system. The full, integrated testbed is currently implemented using the Ettus N310 SDR. The digital signal processing chain is implemented using GNU Radio which runs on a PC connected to the SDR. The testbed accepts clock signal inputs and generates a steered output signal along with characterization reports. Each subsystem block within the testbed is described below.

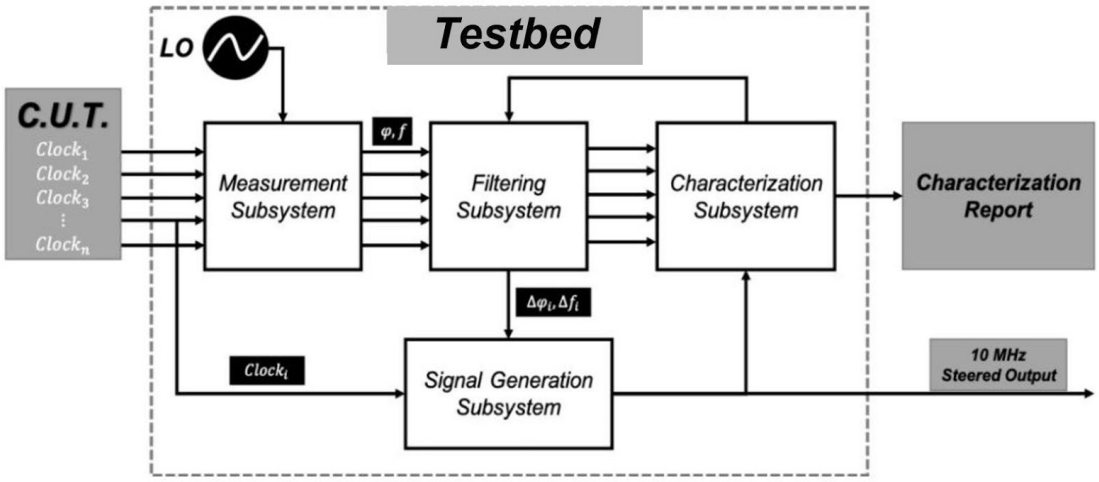


Figure 1. Functional Block Diagram

Measurement Subsystem

The measurement subsystem accepts input signals from multiple clocks and makes phase measurements for comparison between the selected clocks. The clocks under test are sampled and the measurement data is passed to the filtering subsystem.

Characterization Subsystem

The characterization subsystem processes clock time series data to produce Allan Deviation (ADEV) plots. The ADEV plots are used to determine the types of noise affecting the stability of the clock signals over different time intervals. The values are sent to the filter and ensemble subsystem to update the process noise and measurement noise covariance matrices. The ADEVs will also be compared to the frequency stability specifications provided by the manufacturer of the member clocks.

Filtering Subsystem

The filtering subsystem produces estimates for the bias and drift of each member clock. The filter uses a standard clock model and the variance values from the characterization subsystem for the clock state propagation. The differenced clock measurements (i.e. with respect to one of the member clocks) from the measurement subsystem are used as input to the filter. The filter produces clock state estimates relative to the implicit ensemble mean of the clock inputs.

A separate filter is used to estimate the frequency error between the steered output signal and the implicit ensemble mean, generating a frequency adjustment for the steered output signal.

Signal Generation Subsystem

The signal generation subsystem serves the role of realizing the implicit ensemble mean. This is accomplished by steering the frequency of an external OCXO towards the IEM [10]. In order to perform this required task, a programmable voltage sources are used to electronically steer an OCXO based on optimal frequency corrections computed by the filtering system.

3.2 Clocks

Four different types of clocks are used in this project: CSACs, OCXOs, a miniaturized rubidium oscillator (MRO), and a rubidium frequency standard. Photos of the clocks are shown in Figures 2-5. Three CSACs are used as the members of a small clock ensemble and two OCXOs are separately used to realize the implicit ensemble mean. An MRO could potentially serve as a fourth member of the clock ensemble or the IEM realization. The Rb serves as a reference against which other clock signals are measured.

Figure 2. Microsemi SA.45s CSAC [11]

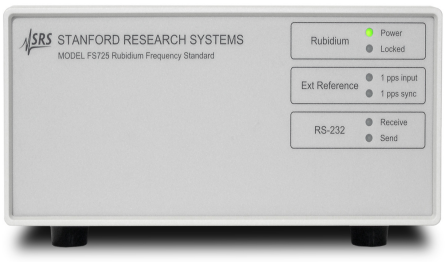


Figure 3. SRS Rubidium Frequency Standard [12]

Figure 4. Wenzel Associates OCXO [13]

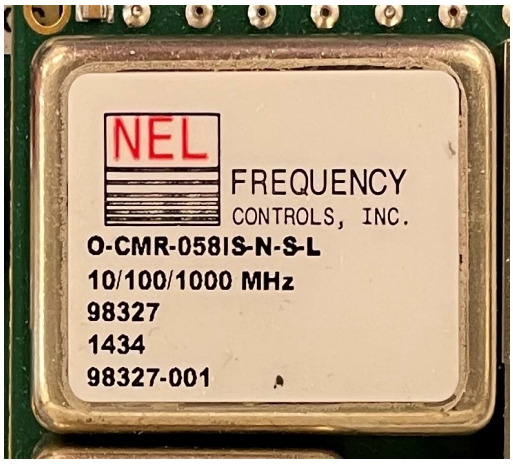


Figure 5. NEL OCXO [14]

Figure 6. Orolia Miniaturized Rb Oscillator [15]

CSAC Interface Board

The CONTACT team designed and built a functionally similar, lower cost alternative to the Microsemi SA.45s Chip Scale Atomic Clock (CSAC) Developer's Kit. Key functional requirements were a simple power interface, stable voltage, dual 10 MHz outputs supporting 50Ω loads, dual 1PPS outputs, and a socketed mount for flexibility in CSAC testing. The following subsections detail the steps involved in the breakout board design and testing.

Design Requirements The design of the breakout board is primarily based on the reference schematic provided by Microsemi for their Developer's kit [16]. All the core digital

Figure 7. Assembled Interface Board

logic circuitry, the power supply and filtering circuitry, and general layout guidelines were based on Microsemi's board design, and were copied over without any major modifications.

In addition to the core logic and power circuitry from Microsemi, some additional features were added to the design to facilitate CSAC testing. These additional features included a mini USB connector and a USB FTDI converter to power and enable communication with the CSAC over the same cable, an additional 10 MHz output derived from the CSAC output, and some ESD protection circuitry in the form of an ESD protection diode to protect the CSAC.

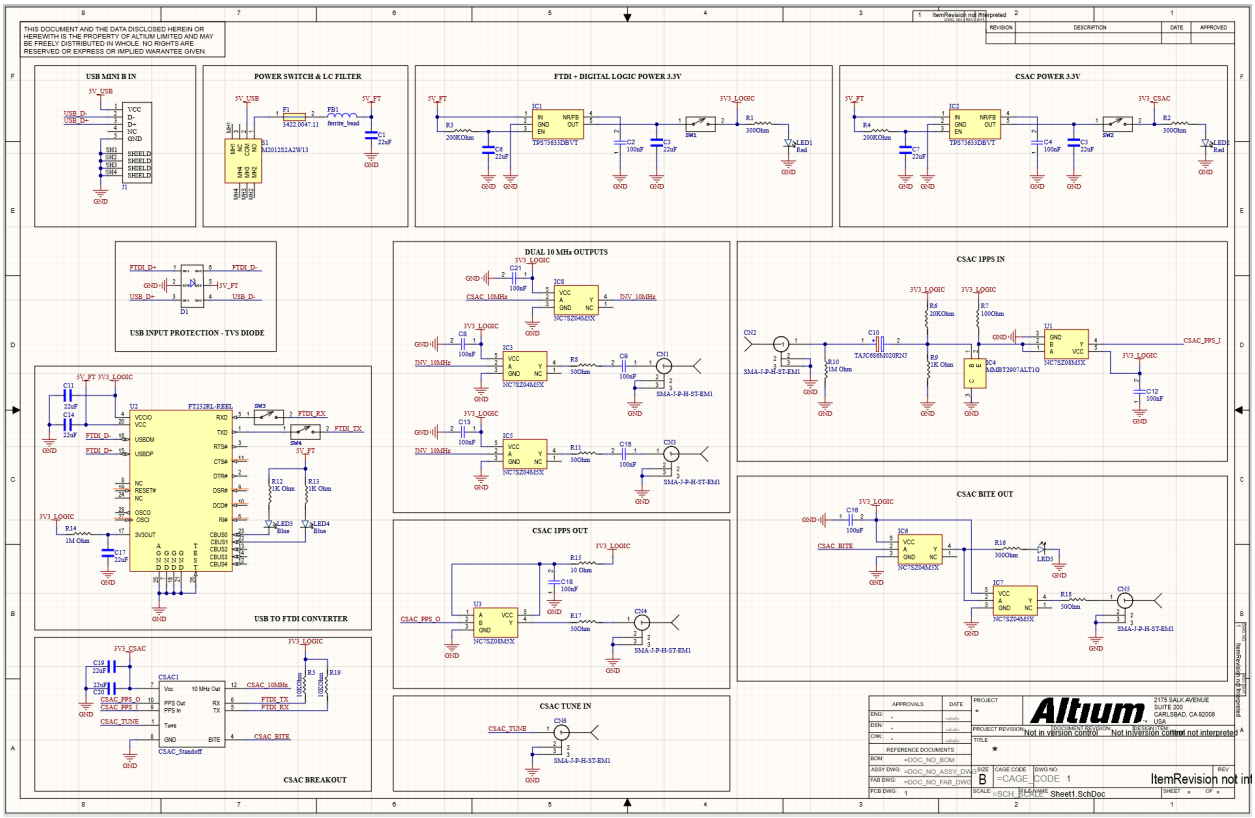


Figure 8. Breakout Board Schematic

Hardware Design Altium designer was used to develop the schematic of the breakout board. The design shown in Figure 8 is the final schematic of the breakout created after combining essential parts of Micosemi's reference design and the additional features added to the board.

To power the board, there are two low dropout 3.3 volt linear voltage regulators, each powered by +5V provided by the USB connector, which are then filtered with a ferrite bead. One regulator is used to power the CSAC, and the other is used to power all other digital logic. The FT232 USB to UART converter is used to communicate with the CSAC over USB. This eliminates the need for a separate +5V adapter to power the board and a separate RS232 cable connected to a USB to RS232 converter to enable communication with the CSAC. The TVS diode connected to the data lines of the USB connector should protect the board against accidental ESD discharges.

The layout, shown in Figure 9, was created using Altium Designer as well. To minimize the potential for signal interference with the clock, the USB inputs and the communication chip were placed away from the CSAC and the low-dropout regulators. The 10 MHz output traces were designed to have 50Ω traces and source series terminations to prevent reflection noise. The final layout was about 37% smaller than the Microsemi Developer's kit.

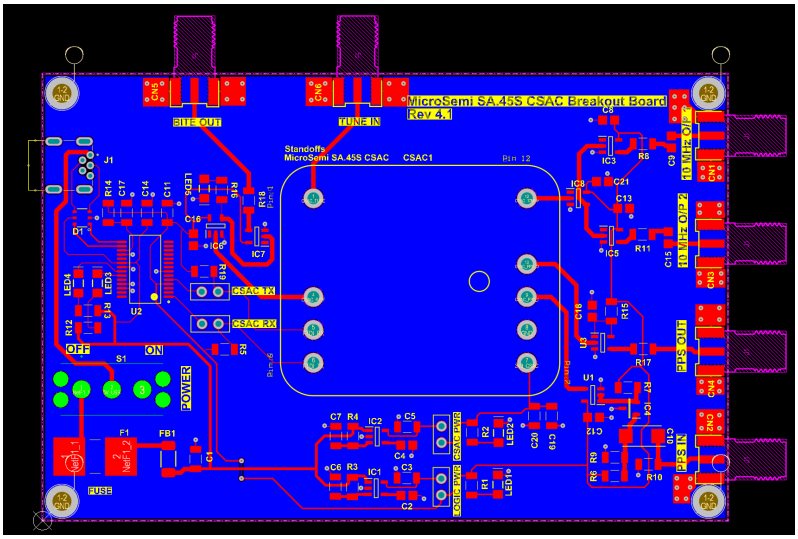


Figure 9. CONTACT Breakout Board Layout

Testing the Breakout Board Breakout board testing was performed in stages to prevent damaging the CSAC during initial power-up. First, after the assembly was completed, a simple continuity test between the pins of the CSAC and the power pads was done to detect any shorts in the board. Once that was done, and no direct shorts were detected, the board was powered on without the CSAC placed on the board. After powering the board for the first time, voltages were checked at the power pins of the CSAC to ensure that they are within operating specifications.

A function generator was used to generate reference 10 MHz and 1 PPS signals. These signals were fed into the digital logic used to condition the CSAC outputs, and the resultant waveform was tested with an oscilloscope. Once testing the digital logic was completed, the TX and RX pins of the USB-FTDI converter were shorted to echo the input, hence testing the USB to serial converter. After all the aforementioned tests were completed, the CSAC was inserted into the socket and all the tests were repeated.

The CSAC breakout board shows up as a COM port when plugged into a USB port, and a terminal emulator (e.g. puTTY) can be used to communicate with the board. On inserting the CSAC and connecting to it, a help menu shows up when the key 'H' is pressed, which is shown in Figure 10. As the CSAC responds to a key press, we confirmed that bidirectional communication with the CSAC is possible.

On comparing the 10 MHz output from the new breakout board (Figure 11a) and the Microsemi Developer's kit Figure (11b), we see that they are very similar. The breakout board has a slightly higher peak-to-peak voltage (40mV), but that might just be an effect of using a different power source to power the board. This slightly higher noise should not

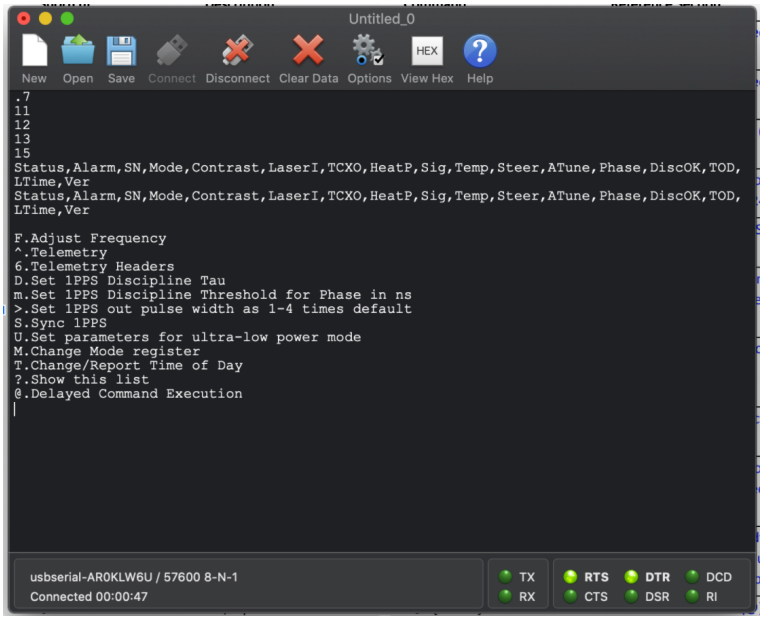
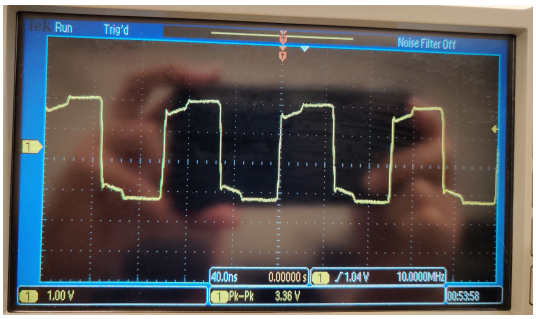
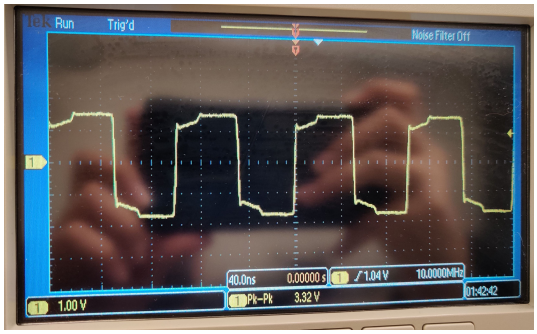


Figure 10. CSAC/Breakout Board Serial Port Communication



(a) Breakout Board 10 MHz output



(b) Microsemi Board 10 MHz output

Figure 11. 10 MHz Output Comparison

affect operation, as the frequency readout from the oscilloscope was a stable 10 MHz, and there is no noticeable noise or distortion around the zero crossings.

3.3 Software Defined Radios

Software defined radios were used to condition, digitize, and characterize 10 MHz timing signals. Three different devices were considered in this project: Red Pitaya 125-10, Ettus N200, and Ettus N310. The Red Pitaya is the lowest cost device, but has the least formal support and proved challenging to program. Both Ettus SDRs were much simpler to program with the GNU Radio software, but are more expensive than the Red Pitaya.

The Ettus N310 was selected for the clock ensemble testbed because it has four receive ports which suits the hardware requirements of the testbed. At least three channels are

required to measure and ensemble the CSACs under test, while the fourth receive channel is used for the steered output signal. The Ettus N200 is used to evaluate the stability of the steered signal against a rubidium frequency reference. This component is only available in the ground testbed; it would not be part of a onboard clock ensemble. Images of the SDRs are in Figures 12 - 14 below and a hardware comparison is shown in Table 2.

Figure 12. Red Pitaya [17]

Figure 13. Ettus N200 [18]

Figure 14. Ettus N310 [19]

Table 2. Comparison Between Hardware Platforms

Parameter	Red Pitaya	Ettus N200	Ettus N310
RX Channels	2	0 or 2	4
TX Channels	2	0 or 2	4
Frequency Range	0-50 MHz	1 - 250 MHz	10 MHz-6 GHz
Processor	ARM Cortex-A9	N/A	ARM Cortex-A9
FPGA	Xilinx Zynq 7010	Xilinx Spartan 3A-DSP	Xilinx Zynq 7100
ADC Resolution	14 bits	14 bits	16 bits
DAC Resolution	14 bits	16 bits	14 bits
Max Sampling Rate	125 MHz	100 MHz	153.6 MHz
Customer Support	No	Yes	Yes
Cost	$500	$2,500	$15,000

3.3.1 Red Pitaya

Red Pitaya devices are a family of low cost, compact, RF data acquisition and signal generation systems [17]. Each device has an ethernet port and unique IP address where users can interact with the device through the online application. There are a suite of default applications in the web interface - oscilloscope, spectrum analyzer, etc - that mirror the functionality of many common laboratory instruments. Alternative ways to interface with the device is over a network via standard commands for programmable instrumentation (SCPI) or through custom C scripts running on the device. The SCPI commands can be sent in MATLAB, LabVIEW, Scilab, or Python.

The Red Pitaya was purchased in the early stages of this project and was never fully incorporated to the testbed. Looking back, it appears the Red Pitaya has potential to be a useful component of a clock characterization system. As the Red Pitaya is a capable, low cost, open source device, there exists an active community who have used the products for many applications beyond the default functionality. Implementation of custom functionality usually requires FPGA modifications, a scary prospect for non-FPGA engineers. The details of these projects are scattered across user forums and GitHub accounts rather than in a central, company supported location - successful implementation of the the Red Pitaya as a clock measurement or signal generation device would require a good deal of time and research for those not familiar with FPGA programming.

The Red Pitaya documentation can be found here: https://redpitaya.readthedocs.io/

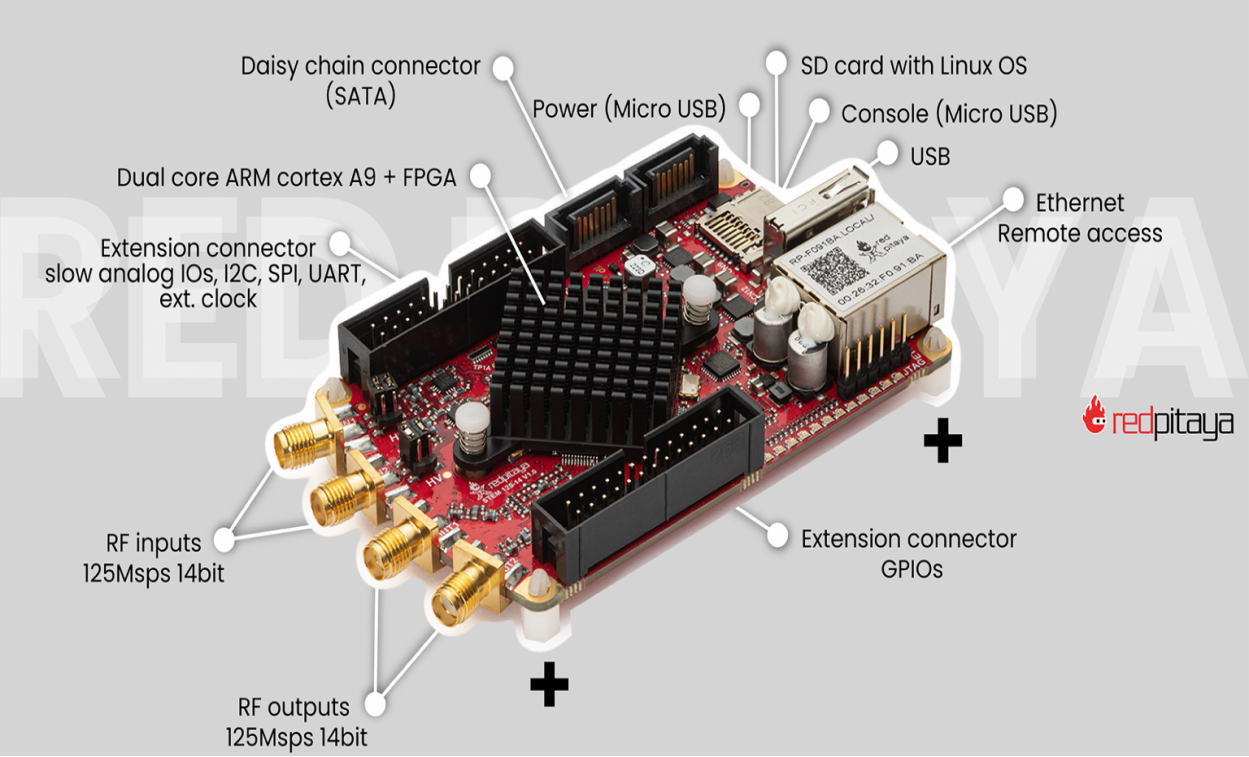


Figure 15. Red Pitaya [17]

Three programming methods were considered for system implementation on the Red Pitaya. Initial setup and testing was performed using the built-in SCPI server. LabVIEW does not support FPGA programming for devices not manufactured by National Instruments. For this reason, directly programming the FPGA directly was considered. The third implementation method was to host C scripts on the Red Pitaya operating system.

SCPI Programming The Red Pitaya documentation provides a set of SCPI commands that can be used to connect with the device in a variety of programming environments. Here we will discuss the experience with MATLAB and LabVIEW, two pieces of software selected due to familiarity of former team members.

The process for acquiring data in MATLAB is quite manual, requiring a series of commands to prepare for the data acquisition and read data from the buffer. The example script used for the test was copied from the documentation and is listed in Appendix 5. A 10 MHz signal was generated with an Agilent 33210A Signal Generator and connected to the Red Pitaya. In this example the decimation factor was set to 1, resulting in a sampling rate of 125 MS/s. Only one frame of buffer values is sent with each SCPI call, which introduces a risk of sample loss between calls.

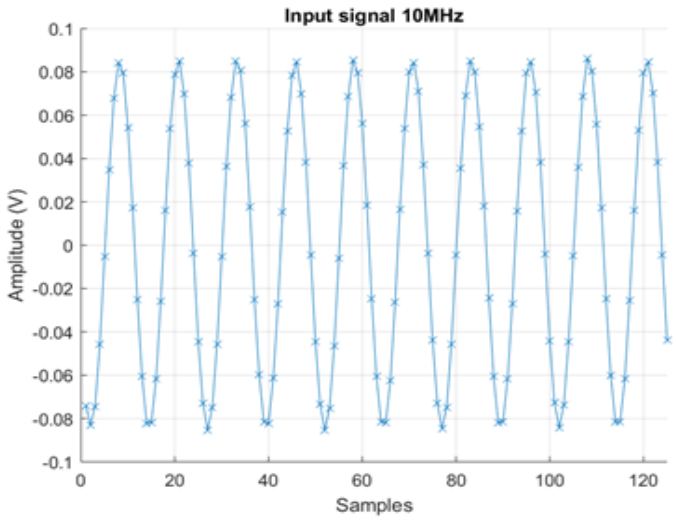


Figure 16. 10 MHz Sampled at 125 MHz - First 125 Samples

Figure 16 show 125 samples of the full 16,000 samples obtained from a single SCPI call. The graph shows approximately 10 cycles of the input signal, which agrees with what we expect given the input signal frequency and sample rate. Initially this method seems promising, but is limiting due to the nature of how data are transmitted between the Red Pitaya buffers and the MATLAB environment - there is always a risk that there will be gaps in the data between the SCPI calls.

Another potential method of interfacing with the Red Pitaya is through LabVIEW. LabVIEW is a graphical programming environment used to interface with National Instruments (NI) hardware platforms, including data acquisition devices and software defined radios. Initial development in LabVIEW seemed promising due to the capabilities demonstrated by similar products in the NI family. However, the team soon realized that since the Red Pitaya is not an NI device, the range of capabilities was much more limited. In fact, the drag and drop GUI blocks in LabVIEW were nothing more than a convenient method to build the same SCPI commands used in MATLAB.

Data Acquisition with C Scripts As mentioned previously, MATLAB scripts and the LabVIEW environment are not suitable for data acquisition because the SCPI calls provide only a single buffer of data, introducing a risk of data loss between calls. However, running C scripts on the Linux Red Pitaya OS allows for a lower level of control of the hardware through C API functions.

C scripts were developed to read input sample values stored in the ADC buffer. The ADC buffer provided by Red Pitaya can store up to 16,000 samples. In the default software, the buffer is split into two halves with boolean indicators showing when each half is full. In order

to ensure continuity of data acquisition, the first half of the buffer is transmitted to the PC once it is full. While the values in the first half of the buffer are transmitting, the ADC is storing information in the second half of the buffer. Using this method of switching between two buffers, data continuity is ensured. Web sockets were set up on both the PC and the Red Pitaya to stream the data. The sample values received by the PC are written to a local file using the File Stream library in C. This file can then be read by other programs such as MATLAB, Python, or C to perform further processing of the sample values.

3.3.2 N310

In 2020 the Ettus USRP N310 software defined radio was chosen as the hardware platform to ensemble clocks, primarily due to the number of channels on the device. It is a networked SDR with an AD9371 transceiver on each of two daughterboards (two Receive (RX) and Transmit (TX) channels each), the Xilinx Zynq 7100 FPGA SoC on the motherboard, a GPS disciplined oscillator (GPSDO), dual SFP+ port, and various other peripheral and synchronization features. The N310 front panel contains the SMA connectors to the TX/RX channels as well as local oscillator (LO) ports as seen in Figure 17. The rear panel contains a variety of host computer connections including ethernet, dual SFP + network, and USB as seen in Figure 18. There are also SMA inputs for a GPS antenna, a reference clock signal, PPS input trigger, and PPS output (trig out).

The sample rates are driven by the master clock which can be set to either 122.88 MHz, 125 MHz, or 153.6 MHz. Samples sent to the host computer through GNU Radio can be decimated by the FPGA to reduce the data rate. The decimation rates can only be set to an integer of the master clock frequency with a maximum decimation value of 1024.

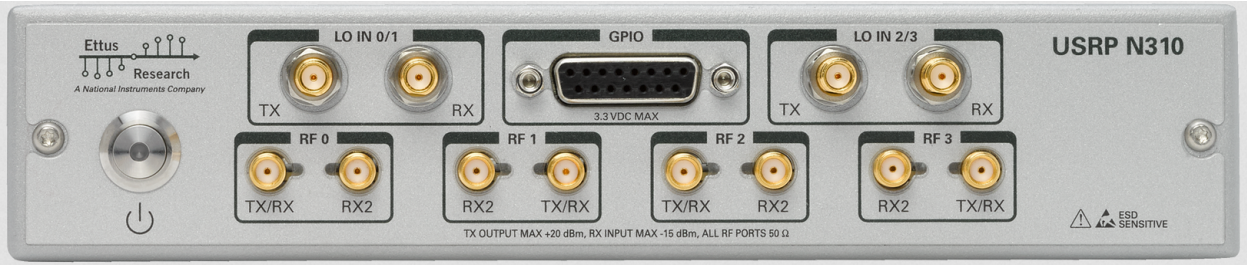


Figure 17. Ettus USRP N310 Front Panel

Figure 18. Ettus USRP N310 Rear Panel

Motherboard The N310 motherboard, as shown in Figure 19, contains the FPGA for digital signal processing as well as the connection interfaces used by the host computer to communicate with the N310. A clocking circuit is used to select a single oscillator input for all other clocks in the system. By default, the clocking circuit is driven by an internal 25 MHz oscillator. Other options include the 20 MHz GPS disciplined oscillator and an external reference clock. The 20 MHz GPS disciplined oscillator input can be used even without a GPS signal if desired - in this configuration the oscillator is an unsteered TCXO.

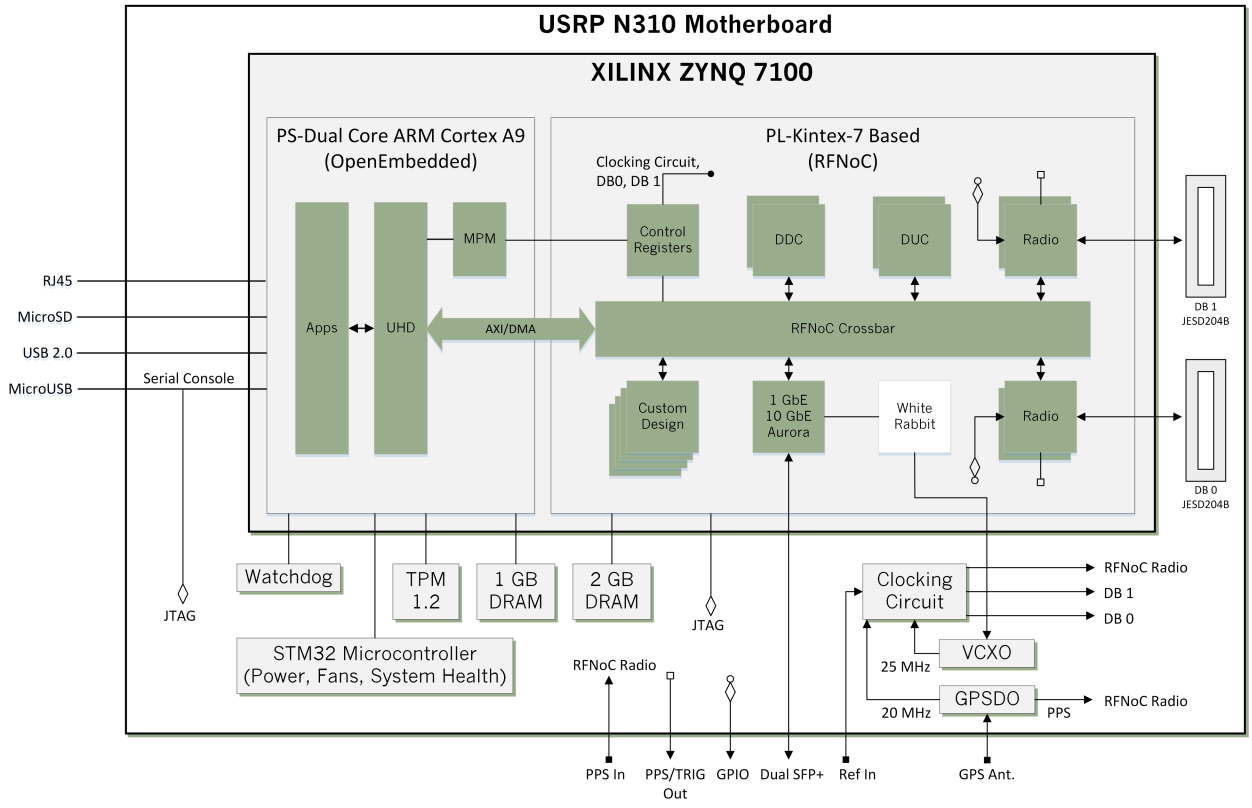


Figure 19. Ettus USRP N310 Motherboard Block Diagram [19]

Transceiver processing chain Each daughterboard supports two TX/RX ports and two RX2 ports. The block diagram for a single daughterboard is shown in Figure 20. Only one receive input between TX/RX and RX2 on a given RF port can be used at any given time for a total of four RX and TX ports, but all four RX and TX ports can be used simultaneously. Since the N310 contains the AD9371 transceiver, which only supports frequencies above 300 MHz, additional LO and mixer stages are needed to shift input signals below 300 MHz into the AD9371 transceiver range. All of the clock signals in our lab output at 10 MHz, meaning that these additional mixer stages will impact the processing . For received signals, the low band upconverter uses an IF of 2.4418 GHz, and for transmitted signals, the low band downconverter uses an IF of 1.95 GHz.

The motherboard contains the clock generation circuit that is connected to both the AD9371 transceiver and the upconverter/downconverter mixer stages. For the AD9371 transceiver, the reference clock signal is used to derive the local oscillators to mix the received signal down or generate the carrier for the transmitted signal. It should be noted that separate LO's are generated for the RX and TX signals, but both RX paths share the same LO and the same is true for both TX paths, so two received/transmitted signals cannot be tuned independently.

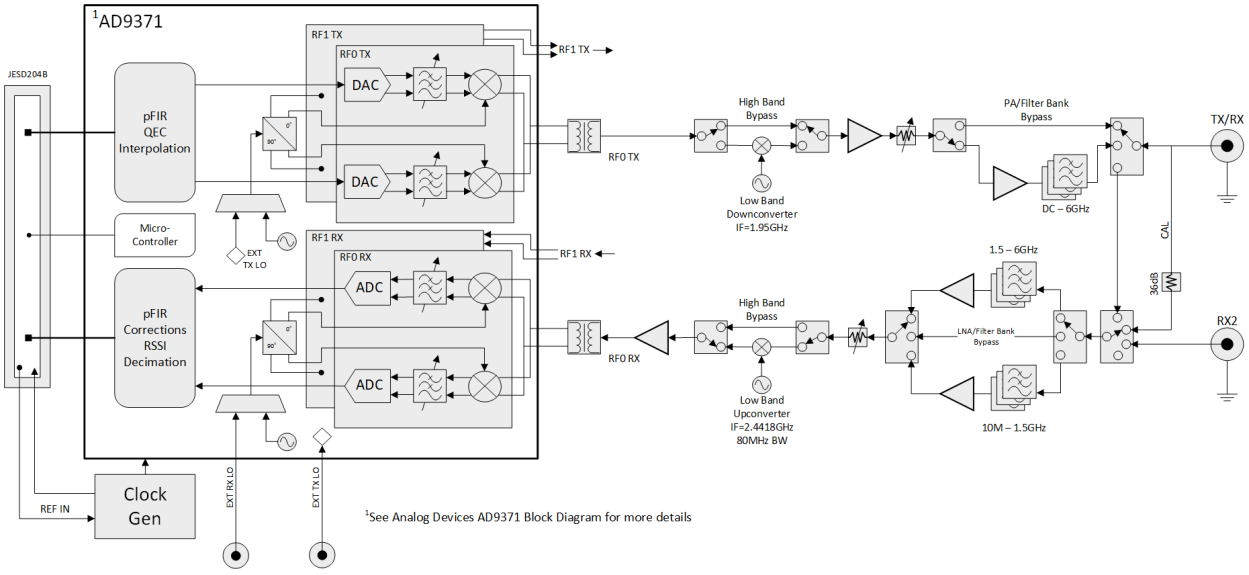


Figure 20. Ettus USRP N310 Transceiver Block Diagram [19]

Tuning The N310 is designed to shift received signals down to or near baseband, using both analog and digital signal processing methods. This shifting of frequencies reduces the required sampling rates and computational overhead in GNU Radio, but introduces a significant a mount o f a nalog s ignal p rocessing (filtering, up conversion, downconversion) before analog to digital conversion (ADC) for received signals, and after digital to analog

conversion (DAC) for transmitted signals.

The host computer communicates with the N310 through the USRP Hardware Driver (UHD). This allows the user to send tune requests through GNU Radio to bring a received signal to baseband. A tune request allows the user to specify the expected frequency of received signal. From this, the N310 sets the analog filters and LO's. There are multiple ways of commanding the N310 to tune the hardware. The simplest is to specify a center frequency, and the N310 will shift the received signal to baseband by mixing down the received signal by the center frequency. Another way is to use a UHD Tune Request. This allows for specifying the center frequency, which still controls the final baseband signal, but also allows for specifying an offset to the physical LO relative to the center frequency. In both cases above, the N310 final tuning is done through two parameters, the RF/LO frequency and a DSP frequency.

Terminology:

- Center frequency: the input frequency that is to be brought to baseband
- RF/LO frequency: the frequency of the physical LO used to mix down the received analog signal
- LO offset: a specified offset to the RF/LO frequency
- DSP frequency: the frequency shift that is done digitally after sampling

When commanded to perform a UHD tune request, the N310 sets the RF/LO to the user specified center frequency plus the LO offset. Figure 21 provides an overview of the tune request. The received signal is first mixed down by the RF/LO frequency. This signal is then digitally sampled, and the DSP frequency is used to shift the signal by the LO offset to bring it fully to baseband. The reason for using an LO offset is to shift the DC component of the LO out of the sample bandwidth. Using the tune request allows more control over the tuning process. The resulting samples for an incoming 10 MHz signal and a specified center frequency of 9,999,990 Hz with and without an LO offset can be seen in Figure 22. Using a center frequency of 9,999,990 Hz should result in a baseband 10 Hz signal. The N310 tuning result using these parameters is much cleaner with the LO offset and for that reason, this is the recommended tuning method.

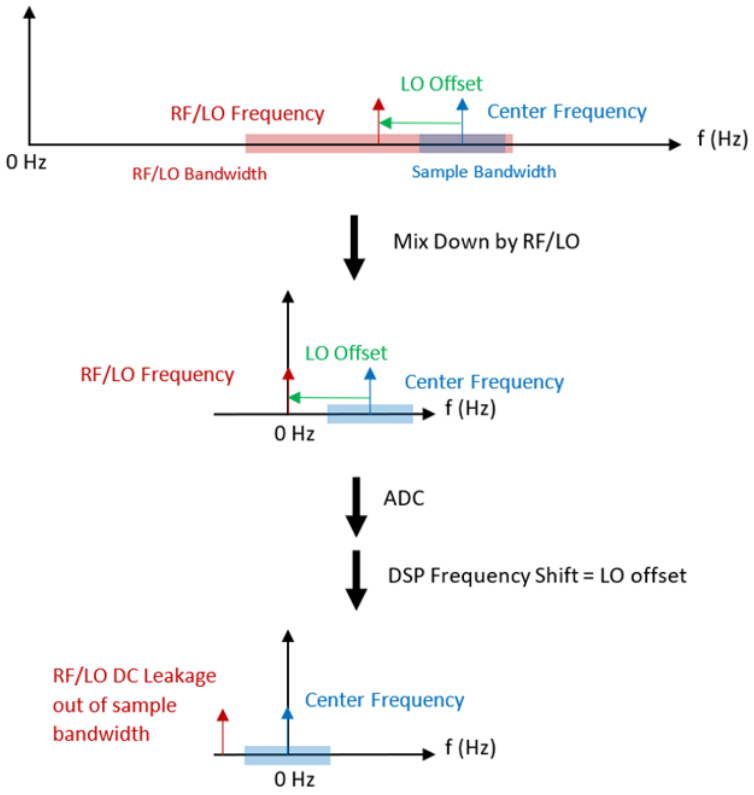


Figure 21. Ettus USRP Tuning with Center Frequency and LO Offset

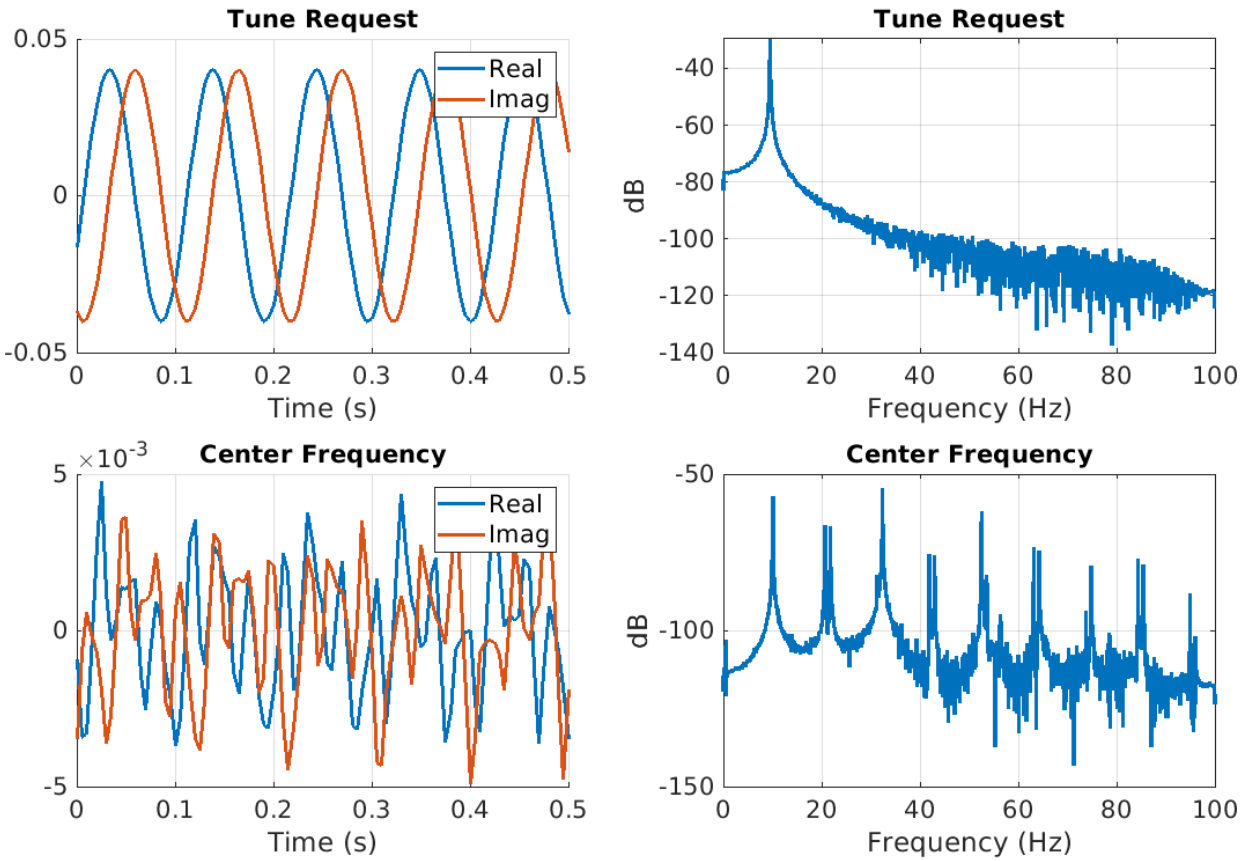


Figure 22. Time Series and Power Spectrum with (top) and without (bottom) LO Offset

3.3.3 N200

The Ettus USRP N200 is a two channel, networked software defined r adio. T he daugtherboards in this device are interchangeable with the capability of being either a transmit or a receive device. The SDR has two RF signal input ports, an external reference input, a PPS input, and ethernet port.

Figure 23. Ettus N200 [18]

The N200 motherboard is shown in Figure 24 which contains the FPGA, the standardized connection for the daughterboards, and the ethernet interface used by the host computer to communicate with the N310. A clocking circuit drives all the clocks on the connected daughterboard as well as the ADC and DAC. By default, the clocking circuit is driven by an internal TCXO oscillator. Other options include connecting an external reference clock or adding a separate GPSDO hardware component.

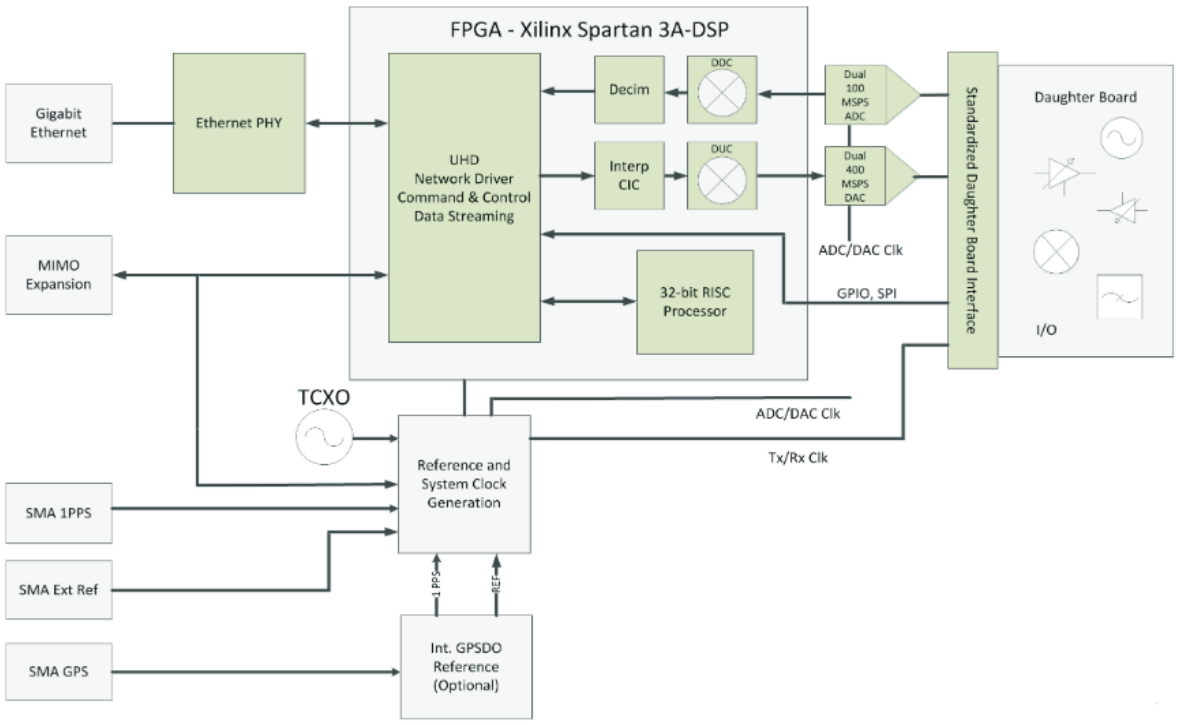


Figure 24. Ettus N200 Motherboard [18]

The N200 documentation can be found here: https://www.ettus.com/n200/.

BasicRX Daughterboard The N200 is different from the N310 in that the functionality of the device depends on the interchangeable daughterboard inside of it. The daughterboard that we used in this project is the BasicRX board which turns the N200 into a receive device with a frequency input range of 1 - 250 MHz.

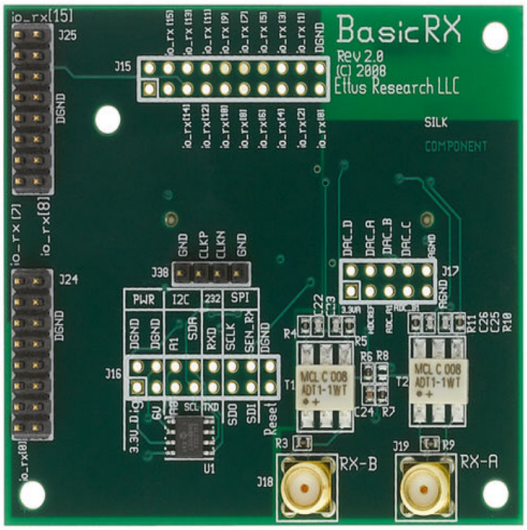


Figure 25. Ettus N200 - BasicRX Daughterboard

The BasicRX documentation can be found here: https://www.ettus.com/basicrx/.

Tuning There is no local oscillator on either the BasicRX or BasicTX boards, meaning that all tuning is handled by the FPGA in the form of digital downconversion. Aliasing is not a concern as the frequency of the clock signals are small with respect to the sampling rates of the DAC and ADC. The error of the local motherboard clock still contributes to the sampled signal as the local oscillator is clocking the FPGA.

3.4 Testbed Software Models

Early in the project a strong emphasis was placed on the development of code to simulate various aspects of the testbed. Writing capabilities in software enabled the team to gain an understanding of the concepts prior to hardware acquisition and enabled rapid prototyping capabilities. The following sections describe the code used to simulate clock profiles, clock steering, and the formation of a clock ensemble. All functions and scripts are included in the appendicies for reference.

3.4.1 Clock Profiles

Clock profiles are the phase and frequency time series for a simulated oscillator. We use a simple two state dynamic clock model, with oscillator specific stability parameters determining the process noise covariance matrix. The discrete time model used in the project is shown in the equations below[20].

$$x_{k+1} = \phi(\tau)x_k + w_k \tag{1}$$

Where the state transition matrix is defined by:

$$\phi(\tau) = \begin{bmatrix} 1 & \tau \\ 0 & 1 \end{bmatrix} \tag{2}$$

And process noise vector:

$$w_k \sim \mathcal{N}(0, Q(\tau)) \tag{3}$$

The process noise is assumed to be Gaussian and zero mean with covariance $Q(\tau)$ described by white and random walk frequency noise parameters, q_1 and q_2. These noise parameters are specific to each clock; realizations of this noise are represented by the noise vector - Equation 3 - and are incorporated into the propagation according to Equation 1. Simulated clock profiles are shown in Section 4.1.1.

$$Q(\tau) = \begin{bmatrix} q_1\tau + \frac{q_2\tau^3}{3} & \frac{q_2\tau^2}{2} \\ \frac{q_2\tau^2}{2} & q_2\tau \end{bmatrix} \tag{4}$$

3.4.2 Clock Steering

The repeated adjustment of oscillator frequency is known as clock steering. Clock steering techniques are often used in the realization of timescales in order to take advantage of the best clock stability across different averaging intervals. When steering in simulation, the phase and frequency offset between the OCXO and the target state is known perfectly. The offset is treated as a perturbation from which a frequency adjustment is computed and applied.

The OCXO frequency is controlled by changing the applied voltage on a frequency adjustment pin. We want to minimize the state error - the phase and frequency difference between the OCXO and the CSAC - by tuning the clock frequency. The discrete-time oscillator dynamics are presented in Equation 5.

$$\mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{u}(k), \qquad \mathbf{u}(k) = -K\mathbf{x}(k), \qquad \mathbf{x}(k+1) = (A - BK)\mathbf{x}(k) \tag{5}$$

The eigenvalues of the matrix A, where $A = \Phi(\tau)$ from Equation 2, do not produce a desirable system response to perturbations. Through the use of a control input, B, and gain matrix, K, the eigenvalues can be chosen such that the perturbations are driven to zero over time. The gain matrix computation is based on the pole placement method [21] where the

characteristic equation of $A - BK$ is used to solve for K matrix values that yield the desired poles.

Examples of a generic clock system response to phase and frequency step inputs are shown in Figure 26. Smaller eigenvalues will result in faster settling times.

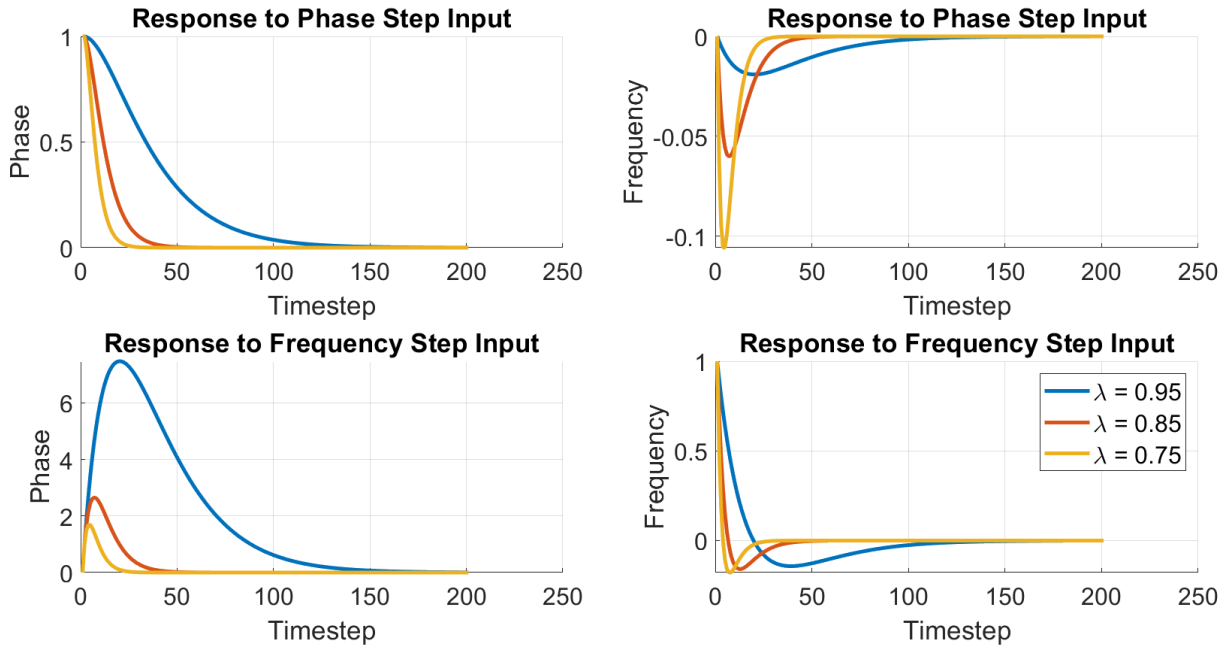


Figure 26. System Response to Phase and Frequency Step Inputs

Clock steering simulations are presented in Section 4.1.2.

3.4.3 Clock State Estimation

The phase and frequency of one clock can only be measured relative to another reference. In our laboratory, rather than measuring a clock directly against a highly stable reference, the phase of the clock under test and the phase of a reference are measured with respect to a less stable clock, as detailed in [7] and Section 3.6.1. The difference in phase measurements removes the effect of the local oscillator and serves as the basis for characterizing the clock under test. In a small system like the CONTACT testbed, we employ this approach, where the sampling clock of the SDR serves as the common clock source, and differences between two of the clocks under test, or a lab reference are used to characterize. The clock ensemble uses relative phase measurements between clocks of similar stability as input to a Kalman filter. The filter uses these phase difference measurements to estimate the member clock states following Brown [9] and also as outlined in [8].

Clock Model The clock model and state transition matrix for the Kalman filter is shown below. There are three clocks each with two states - phase and frequency - represented by b and f, respectively.

$$x_k = \begin{bmatrix} b_{1,k} \\ f_{1,k} \\ b_{2,k} \\ f_{2,k} \\ b_{3,k} \\ f_{3,k} \end{bmatrix}, \qquad \phi(\tau) = \begin{bmatrix} 1 & \tau & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \tau & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \tau \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

Measurement Model The inputs to the system are phase difference measurements between the ensemble member clocks. The phase difference measurements are used to estimate the phase and frequency of each of the member clocks.

$$z_k = Hx_k + v_k \tag{7}$$

$$H = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{8}$$

A system composed of N clocks and $N-1$ relative clock measurements will always be unobservable when estimating N clock states, since an error common to all of the clocks will not manifest in the relative measurements [9]. An assessment of the observability matrix will show that the system is unobservable for 3 clock states.

$$rank(O) = rank\left(\begin{bmatrix} H \\ H\phi \\ H\phi^2 \\ \vdots \\ H\phi^{m-1} \end{bmatrix} \right) = 4 \ \forall \ m \geq 2 \tag{9}$$

Since there are 6 elements of the system state vector, the covariance matrix of the state estimate in the filter will continue to grow without bound. The covariance reduction method detailed in [9] separates the observable and unobservable components of P and uses the observable component to prevent unbounded uncertainty growth.

$$H^* = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{10}$$

$$P_r = P - H^*(H^{*^T} P^{-1} H^*)^{-1} H^{*^T} \tag{11}$$

IEM Realization A steerable OCXO serves as the basis of the IEM realization. The clock state estimates and the phase of the OCXO are used to compute the phase offset of the OCXO with respect to the IEM. Following the theory in [8], the clock estimates ($\hat{x}_i(t_k)$) and measurements ($z_k(t_k)$) can be written as follows:

$$\hat{x}_i(t_k) = x_i(t_k) - x_0(t_k) - e_i(t_k) \tag{12}$$

$$z_N(t_k) = x_{N+1}(t_k) - x_1(t_k) + v(t_k) \tag{13}$$

The phase estimate of the first clock is added to the phase difference between the OCXO and the first clock. The output of this operation is the offset of the OCXO with respect to the IEM, along with measurement and estimation error. This phase value is input to a filter to estimate phase and frequency. The estimated phase and frequency are then used to compute a frequency adjustment command for the OCXO.

$$\hat{x}_1(t_k) + z_3(t_k) = x_4(t_k) - x_0(t_k) - e_1(t_k) + v(t_k) \tag{14}$$

3.5 Programming in GNU Radio Companion

GNU Radio Companion (GRC) is an open source programming environment designed to provide users with methods to simulate or interface with software radios [22]. The graphical user interface is drag-and-drop with built in blocks performing a variety of signal processing functions. Data are represented as signal streams passing between various blocks that operate on the data. In GNU Radio Companion the USRP devices can be data sources (receivers) or data sinks (transmitters). Once the clock signals are inside the GRC program, the data streams are operated on by custom signal processing chains using both the built in signal processing blocks as well as custom blocks. Some of the built-in blocks are listed in Table 3.

Table 3. GNU Radio Blocks

Block Type	Description
UHD: USRP Source/Sink	Enables the user to receive/transmit signals from/using the SDR (includes the UHD tune request)
File Source/Sink	Read or write data streams from/to a binary file to enable pre/post-processing
Complex to Arg/ Arg to Complex	Convert between I & Q data and signal phase
Add/Subtract/ Multiply/Divide	Perform standard element-wise operations on a stream input data
Rational Resampler	Resample a signal through interpolation/decimation of an input data stream
Signal Source	Generates a digital signal during program execution
QT GUI Sink	Displays the frequency spectrum and time-domain of signals during program execution

3.5.1 Moving Data and Data Rates

Each GNU radio program we have developed starts with a digitized, complex signal stream for each clock input to the SDR. Keeping track of the sampling rate at the different stages in the program is important; the block functionality often depends on the sample rate and a rate mismatch can cause errors in how the data are processed. We use variables for the sample rate and decimation blocks to monitor the true sample rate at various points in the program.

The sampling rate at most points in the GNU Radio program is slow enough that it is not causing processing choke points and fast enough that there are not buffer filling issues, as described next. In this nominal scenario the data rate from a block output is equal to the input data rate. The relevant edge case where this breaks down is at low sample rates.

Very low frequency processes can be tricky to implement due to the way GNU Radio moves data around. The dynamic GNU Radio scheduler attempts to optimize the program performance by breaking the streams of data into large chunks. The overhead associated with moving data is significant. The program runs more efficiently (i. e. with higher throughput) when each block in the flowgraph operates on large chunks of data at a time, minimizing data movement. So, GNU Radio sets up the program such that each block includes an input buffer that fills up with the chunks of data prior to program execution.

However, this method of data handling presents a challenge for real-time systems[23] with components that are low frequency - such as a < 1 Hz. A block will only operate on the input data once the input buffer is filled - as a result, a block may be waiting for the input

buffer to fill up and will not process data at the desired rate. The minimum buffer size in GNU Radio is 8 samples. If the input rate to a block is 1 sample per second, the block will operate on all samples every 8 seconds rather than match the output rate to the input rate. In order to realize an output data rate of 1 sample per second, the input data rate should be set higher and decimation should be handled in the Python code to allow time for the buffer to fill. In this method the buffer fills faster than the desired output rate and the output rate is controlled programatically.

3.5.2 Embedded Python Blocks

Embedded Python blocks allow users to develop functionality not provided by built in signal processing blocks. When an embedded block is added to the GNU Radio program a boilerplate Python script is generated that will operate on the input data stream. This underlying Python script can be significantly edited to achieve the required functionality.

Inside the Python code there is an initialization function and a work function. The initialization function controls the number of inputs and outputs of the block, the corresponding data types, and any default parameters that should be passed into the block through the GUI or constants that should be initialized. The work function operates on the input data streams and assigns the processed data to the output ports of the block. The example provided by GRC is a simple scaling operation - however, reasonably complex functionality can be implemented in this code such as phase unwrapping, phasor subtraction, Kalman filters, and communication between the instruments in our laboratory. The Python functions we have developed are attached in the Appendix. A list of the custom python blocks that the team has developed is shown in Table 4 below.

Table 4. GNU Radio Embedded Python Blocks Developed for CONTACT

Block Type	**Description**
Phasor Block	Remove a nominal 73 Hz signal from the I and Q data
Phase Unwrap Block	Adjust signal phase by $N * 2\pi$
Clock Estimate KF	Estimate phase and frequency of N clocks from $N-1$ relative phase measurements
OCXO KF	Estimate phase and frequency of OCXO offset from IEM from OCXO phase measurements
Power Supply Message	Convert a frequency adjustment to a voltage and send command to power supply

3.5.3 USRP Block Options

Figure 27 shows the general options for setting up the USRP source block in GNU Radio. The device arguments field can be used to set a variety of other options available to the N310, such as tracking calibrations built into the AD9371 transceiver, master clock rate, and others. For a full list of device arguments please see the documentation here: https://files.ettus.com/device-args/.

The tuning parameters are assigned in the RF options tab of the USRP Source block, shown in Figure 28. The center frequency of each RF channel is filled with a UHD tune request command, specifying the center frequency and LO offset values of the tuning chain. The tuning process is described in detail in Section 3.3.2.

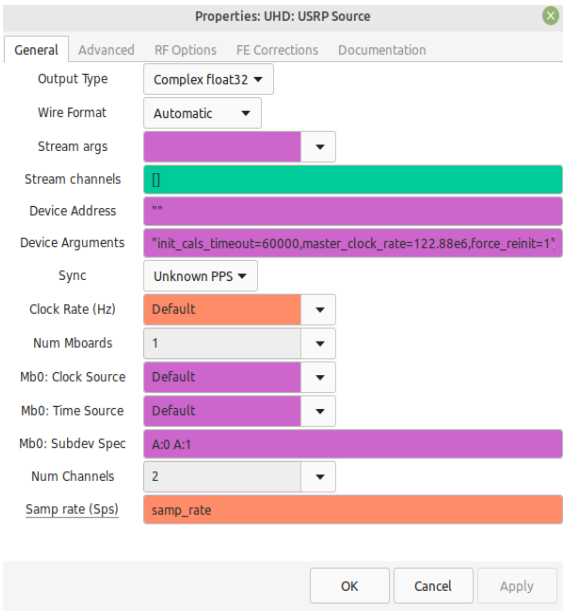


Figure 27. USRP Source Block Settings

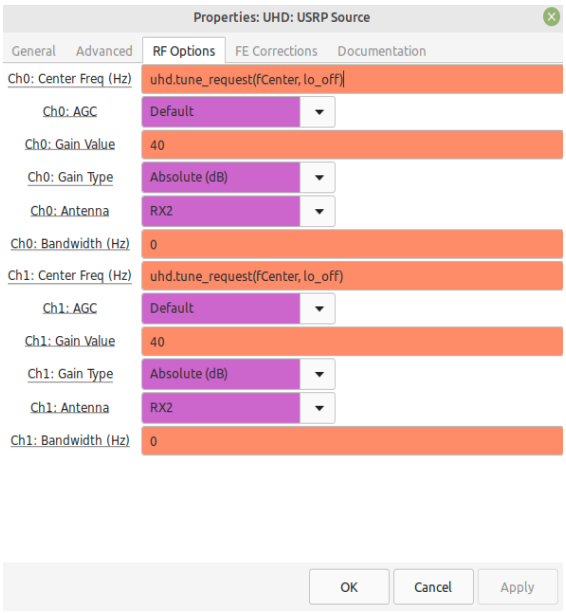


Figure 28. USRP Source Block RF Options

3.5.4 Loops

Despite many RF and signal processing applications requiring them, GNU Radio does not explicitly support loops. An error will be thrown if a loop is created when the output of a block is connected to the input of another block. The reason for this is related to how data streams are moved in the program - detailed in Section 3.5.1 - and described further here: https://wiki.gnuradio.org/.

A loop could be created in GNU Radio by connecting the output of frequency adjustment code to a signal source block or USRP signal sink, both methods of signal steering where

the signal source originates in GNU Radio. Theoretically, this setup would satisfy the requirements of the testbed - a frequency adjustment is continuously computed based on the input clock signals and the frequency is adjusted accordingly. However, this program would not run due to the reasons mentioned above.

There are a few ways to circumvent the loop limitations - they can be constructed via an asynchronous message passing interface or an embedded Python block. The asynchronous blocks are built into GNU Radio but provide no guarantee on execution timeliness, as per the name. Embedded Python blocks are the best option for implementing loop structures within GNU Radio. As a block of data is passed to the function, a loop can be written to iterate over and operate on the incoming data. In terms of closing the frequency steering loop, the signal source is kept external to GNU Radio and is controlled via Python commands. Different Python packages are used to control the voltage of a power supply or DAC connected to an OCXO, closing the loop by sending commands from GNU Radio to the respective device.

3.6 Clock Phase Measurements

The software defined radios and GNU Radio environment together enable us to make phase measurements of the clocks. Measurement campaigns which store the time series of the clock phase allow us to compute the frequency stability of the oscillators under test.

A journal paper from NIST [7] provides the theory upon which the measurement system is based. Both the theory and our implementation make phase measurements by recording phase information of a low frequency ($f << 10$ MHz) beat signal. The processing diagram shown in Figure 29 differs slightly from our implementation in Figure 30 - the beat frequency of the signal is removed prior to computing the argument of the complex signal.

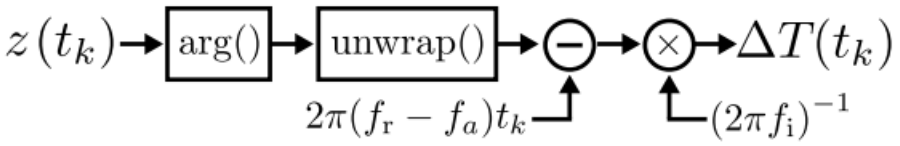


Figure 29. Measurement System Block Diagram - Theory [7]

In the figure above, $z(t_k)$ represents the complex signal data that is passed into the argument function. The wrapped phase is then unwrapped, the ideal linear phase growth is removed, and the remaining data is used to calculate the time offset based on the measured phase difference. The measurement system implements the above logic via custom python blocks in GNU Radio Companion.

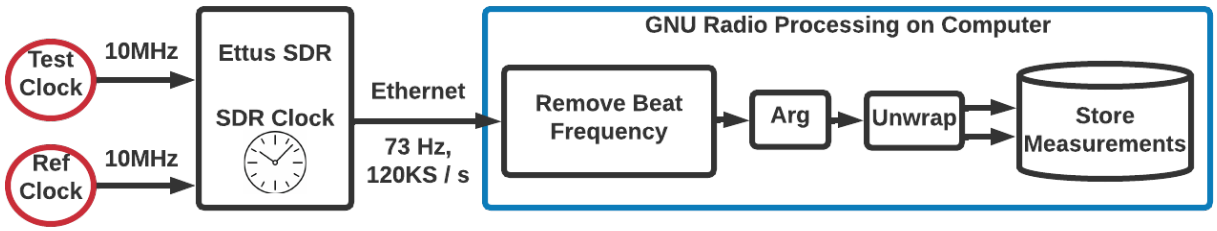


Figure 30. Measurement System Block Diagram - Implemented

Two clocks shown on the left side of Figure 30 produce analog 10 MHz signals. After the SDR performs the signal conditioning, downconversion, and sampling, the stream of samples sent to the host PC is near baseband, offset by 73 Hz, and sampled at 120 kHz. During the frequency mixing process the clock onboard the Ettus N310 contributes a common error that is present in both signals. The argument of the complex signal is computed and results in a phase from $[-\pi, \pi]$. This wrapped phase is unwrapped, producing phase information dominated by the local clock stability of the SDR oscillator. The primary trend in the measured phase deviation for each clock over time will be due to the clock onboard the SDR, as shown by the results in Figure 47.

We found that it is important to remove a 73 Hz signal from the beat frequency prior to computing the argument and unwrapping the data. The unwrapped phase for each 73 Hz beat signal, while not a particularly high frequency, will grow without bound at the rate $\phi = 2\pi f_{beat} t$. The unwrapped phase of both signals are subtracted from each other, providing a phase difference between the test and reference clocks. The problem with this approach is that the magnitude of the frequency error in the test clock is much smaller than the beat signal frequency. As the unwrapped phases grow without bound, there is a loss in decimal precision of the numerical phase difference due to limits in the finite number of bits used to represent the unwrapped phase values. Any loss in decimal precision will degrade accuracy with which the test clock phase can be measured, since the clock errors of interest are very small and change quite slowly.

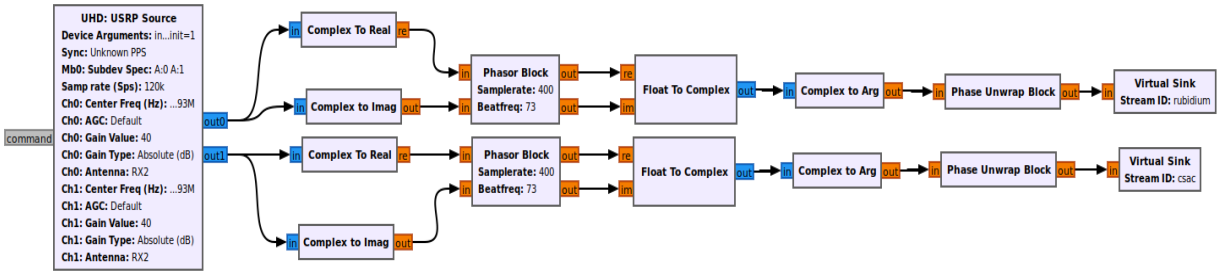


Figure 31. Measurement System Implemented in GNU Radio Companion

Figure 31 shows the GNU Radio block diagram used to make phase measurements of one clock with respect to another. The two custom blocks in this flowgraph are the Phasor Block, which removes the 73 Hz signal, and the Phase Unwrap Block, which unwraps the signal phase. The code for the Phasor Block is in Appendix 5 and the code for the Phase Unwrap Block is in Appendix 5.

3.6.1 Effect of the Local Oscillator

Any signals that are measured using an SDR will initially be measured against the local clock on the device. The internal clock of the measurement system has relatively poor stability compared to atomic clocks and will impact the measured phase behavior of any input signals. This approach is quite similar to clock characterization systems at NIST where a multi channel measurement system (MCMS) is used to compare up to 16 oscillators on one device[24]. The local oscillator of the MCMS has worse stability than the input signals, but comparing two signals on the MCMS will cause the local oscillator effect to drop out. In our setup we have four input clock signals: a rubidium frequency standard and three CSACs. The clock phases were computed using the process shown in Figure 30. Effects of the local oscillator on the input measurements are shown in Section 4.2.1.

3.6.2 External Reference Oscillator

Each SDR has an external reference input where a timing signal can be supplied to functionally replace the local oscillator. The ability of the N200, USRP2, and N310 to lock to an external reference was evaluated in three different configurations: transmit mode, receive mode, and simultaneous transmit and receive mode. The overlapping Allan deviation was used to characterize signal locking in the SDR. A reference lock is verified by examining the frequency stability of the measured SDR signal with and without an external clock input. If the stability of the SDR signal matches the stability of the reference clock then we say a successful lock has been achieved. Each test was conducted for 45 minutes. GNU Radio Companion was used to interface with the SDRs and store complex signal measurements for post-processing in MATLAB. All external reference oscillator test results are presented in Section 4.4.

Each test used the stored signals from GNU Radio Companion and converted them from IQ data to phase measurements. Equation 15 shows the phase measurement model, which consists of the beat frequency f_b and phase variation $\Delta\phi$ due to oscillator noise properties.

$$\phi = 2\pi t f_b + \Delta\phi \tag{15}$$

The phase contribution due to the beat frequency is subtracted from the measured phase to isolate the phase variations, $\Delta\phi$. The $\Delta\phi$ term contains the aggregate contributions due to frequency instability in the test oscillator and noise from signal processing.

SDR Transmit Test The first test configuration analyzes the transmitted signal stability from an SDR locked to an external reference. A separate Ettus device receives the transmitted signal from the test SDR, A(t), and the rubidium frequency reference (Rb), B(t). The test configuration is shown in Figure 32.

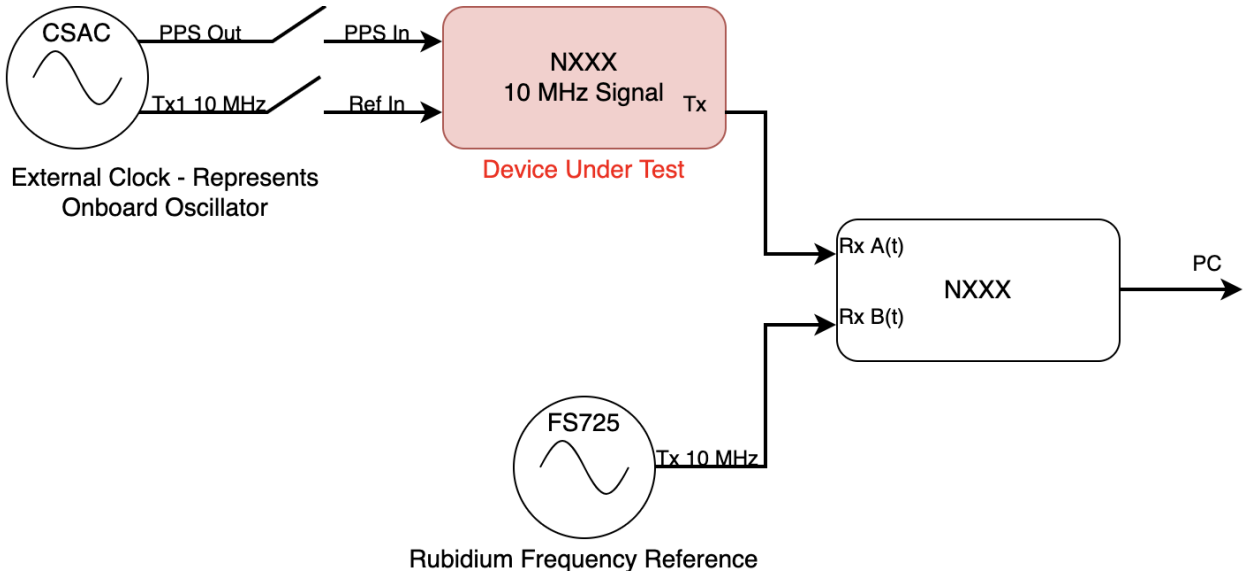


Figure 32. SDR Transmit Test Configuration

Both input signals are initially measured with respect to the local oscillator on the receive SDR, $\Delta\phi_{Rx}$. The measured phase variations, $\Delta\phi_A$ and $\Delta\phi_B$, will contain a significant phase contribution due to frequency instabilities in the SDR local oscillator (LO). This error is common to the received signals, and the difference o f t he p hase v ariations r emoves the common error, $\Delta\phi_{Rx}$, as shown in Equation 16 and 17. The phase difference y ields the frequency stability of the SDR transmitted signal as measured against the Rb.

$$\Delta\phi_A = \Delta\phi_{Tx} - \Delta\phi_{Rx}, \qquad \Delta\phi_B = \Delta\phi_{Rb} - \Delta\phi_{Rx} \tag{16}$$

$$\Delta\phi_A - \Delta\phi_B = \Delta\phi_{Tx} - \Delta\phi_{Rb} \tag{17}$$

With no external reference supplied to the transmitting device, the $\Delta\phi_{Tx}$ term represents SDR LO instability. When a CSAC external reference is connected to the SDR the device

realizes the new clock and the $\Delta\phi_{Tx}$ term now represents the stability of the CSAC, $\Delta\phi_{CSAC}$, and a noise contribution from the phase-locked loop (PLL), $\Delta\phi_{PLL}$, as shown in Equation 18.

$$\Delta\phi_A - \Delta\phi_B = (\Delta\phi_{CSAC} + \Delta\phi_{PLL}) - \Delta\phi_{Rb} \tag{18}$$

The oscillators on all SDRs have relatively poor stability; therefore $\Delta\phi_{Tx}$ will be the dominate trend in measured phase and the measured signal should represent LO stability. When a CSAC is used as an external reference clock $\Delta\phi_{Tx}$ is significantly reduced and $\Delta\phi_{CSAC}$ is the largest contributor to instability. This will result in the transmitted signal from the SDR exhibiting CSAC like stability, as seen in the ADEV results in Section 4.4.1.

SDR Receive Test The second test configuration verifies SDRs' ability to lock to an external reference when operating as a receiver. For this configuration a single SDR is setup to receive two clock signals, one from the rubidium frequency reference and the other from the CSAC. The same CSAC will provide a reference clock to the SDR. Figure 33 illustrates this configuration.

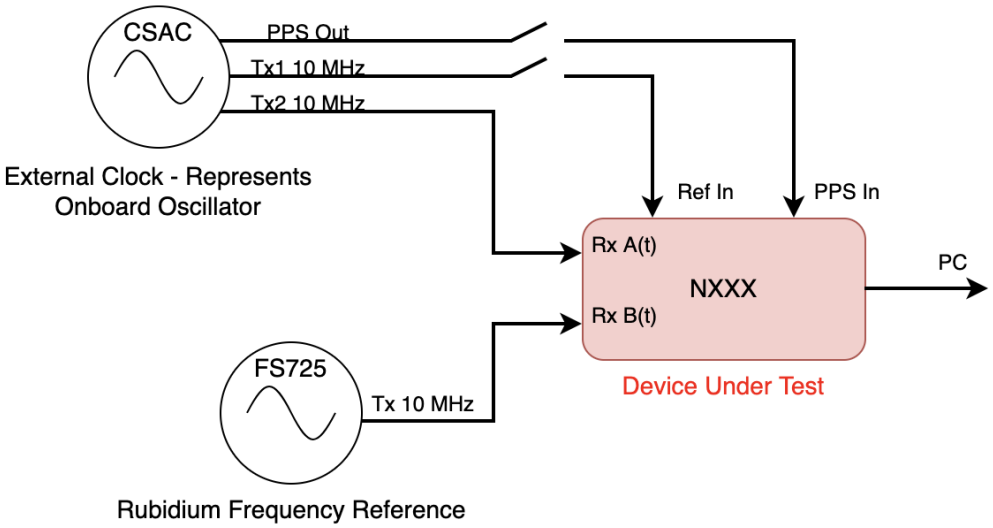


Figure 33. SDR Receive Test Configuration

Both received signals will be measured against the LO of the SDR under test. Each signal was examined independently to investigate the effect of the LO on the frequency stability, with and without the CSAC as a reference clock. The phase variations for each receive channel, $\Delta\phi_A$ and $\Delta\phi_B$, include the contribution of each source clock and the SDR, as shown in Equations 19 and 20.

$$\Delta\phi_A = \Delta\phi_{CSAC} - \Delta\phi_{Rx} \tag{19}$$

$$\Delta\phi_B = \Delta\phi_{Rb} - \Delta\phi_{Rx} \tag{20}$$

Without a reference clock the frequency instability will be predominately from the LO of the SDR, $\Delta\phi_{Rx}$. Both received signals result in overlapping Allan deviation curves that match the stability of the SDR LO. With the CSAC connected as an external reference clock the $\Delta\phi_{Rx}$ term now represents the frequency variation in the CSAC, $\Delta\phi_{CSAC}$, and the SDR PLL, $\Delta\phi_{PLL}$. This eliminates the CSAC frequency instability contributions to the A receive chain with only the PLL noise component remaining, as shown in Equation 21.

$$\Delta\phi_A = \Delta\phi_{CSAC} - (\Delta\phi_{CSAC} + \Delta\phi_{PLL}) = -\Delta\phi_{PLL} \tag{21}$$

The B receive is now the Rb signal measured against the SDR LO realization of the CSAC. As shown in Equation 21, the contributors to the frequency variation will be the Rb, CSAC, and PLL. In this case the $\Delta\phi_{CSAC}$ term becomes the primary contributor to instability in the received signal and the resultant ADEV curve exhibits CSAC-like stability. These results are presented in Section 4.4.2.

$$\Delta\phi_B = \Delta\phi_{Rb} - (\Delta\phi_{CSAC} + \Delta\phi_{PLL}) \tag{22}$$

SDR Transceiver Test The last test configuration involves operating the USRP2 and N200 as both receivers and transmitters simultaneously, as shown in Figure 34. These devices use interchangeable daughterboards and had previously only been operated in a transmit or receive mode. In this test both a receive and transmit board were installed in the devices with one RF cable connected to each board, making the SDRs transceivers. This test has a twofold objective: to determine if the USRP2 and N200 can operate as a transceiver and to ascertain how well the devices can lock to an external reference clock while in a transceiver mode. A CSAC is the USRP2 external reference clock and the rubidium is an external reference clock to the N200. For comparison the test was also conducted with no reference clocks provided to either SDR.

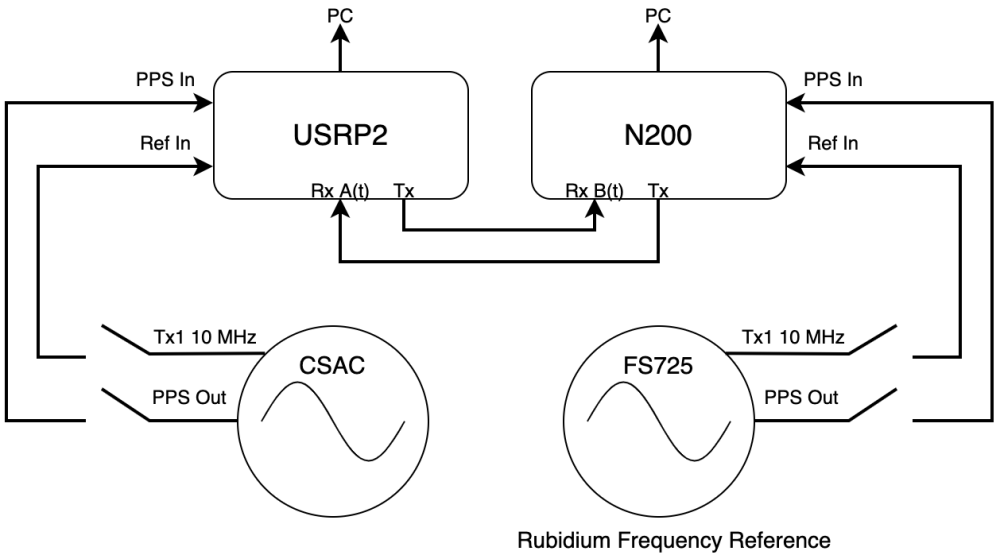


Figure 34. Simultaneous Transmit & Receive Test Configuration

The USRP2 will receive a signal that includes the frequency variations from the N200, $\Delta\phi_{N200}$, and will be measured against the USRP2 LO, $\Delta\phi_{USRP2}$. Received signals from each SDR will express frequency variation similar to the LO with the largest instability. This stability limiting oscillator will depend on the local oscillator on each SDR, as shown in Equations 23 and 24.

$$\Delta\phi_A = \Delta\phi_{N200} - \Delta\phi_{USRP2} \tag{23}$$

$$\Delta\phi_B = \Delta\phi_{USRP2} - \Delta\phi_{N200} \tag{24}$$

With the addition of reference clocks the received signals will include the frequency variation from the PLLs and both reference clocks. The received signals in the USRP2 and N200 will now be limited by the reference clock with the poorest stability, in this case the CSAC. Equation 25 and 26 show the measured phase variation from each SDR. The $\Delta\phi_{CSAC}$ will be the dominant term, resulting in the received signals exhibiting CSAC like stability. The corresponding ADEV curves shown in Section 4.4.3 support this theory.

$$\Delta\phi_A = (\Delta\phi_{Rb} + \Delta\phi_{PLL-N200}) - (\Delta\phi_{CSAC} + \Delta\phi_{PLL-USRP2}) \tag{25}$$

$$\Delta\phi_B = (\Delta\phi_{CSAC} + \Delta\phi_{PLL-USRP2}) - (\Delta\phi_{Rb} + \Delta\phi_{PLL-N200}) \tag{26}$$

3.6.3 Measurement Noise

Whether characterizing one clock with respect to another or making relative phase measurements for use in a Kalman filter, SDR phase measurements are usually made by subtracting phase values. The SDR noise contribution to these measurements is computed using phase measurements of a common clock signal. There are two transceivers in the N310 - shown in Figure 35 - which each have separate noise cancellation properties. All of the RX inputs of the N310 and N200 will be occupied for the fully integrated clock ensemble tests; as such, it is important to characterize the noise contribution from each device to the measurements. With two transceivers on the N310, there are three unique measurement systems of interest: the N200, the same transceiver of the N310, and the different transceivers of the N310. The same clock signal was sent into each of these measurement systems, the phase was computed, and then differenced. The resulting data represent the combined measurement noise of the system under test, shown in Figure 51. The ADEV values are computed on this data to show the magnitude of the measurement noise as compared to the clock stabilities of interest. If the measurement noise contribution is larger than the clock ADEV values, data gathered from the system will obscure the true stability of the clocks at certain averaging intervals. Results for the measurement noise tests are in Section 4.2.2.

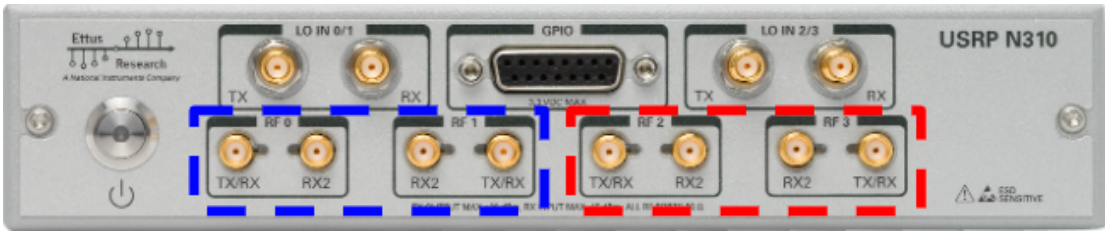


Figure 35. N310 with Transceiver 0 Inputs (Blue) and Transceiver 1 Inputs (Red)

3.6.4 Clock Phase Measurement Considerations

One feature to note when using an SDR for clock characterization is the low input power limit for both SDRs, -15dBm. Therefore, when connecting the clock signals directly to the SDR they must be attenuated to levels below the maximum allowable input power. In addition, the Ettus N310 is designed to support signals ranging in frequency from 10 MHz to 6 GHz. The ensemble member clocks generate signals at 10 MHz. Using oscillators that are at the low end of the N310's RF capabilities requires an additional upconversion stage within the SDR that contributes noise to the clock measurements. This upconversion is driven by the AD9371 transceivers, which support frequencies above 300 MHz. The N310 upconverts the 10 MHz clock signals into the operational range of the transceivers and then downconverts the signal prior to being sampled by the ADC. The daughterboards in both the N200 and

USRP2 directly digitize the input signals, so the effect of the analog pre-processing is not relevant.

The low frequency signals from the analog RF processing are digitized by the ADC, sent to the PC, and finally through the GNU Radio signal processing chain. It is important to note that the frequency shifted signals significantly reduce the data storage requirements associated with the digital signal processing by decreasing the required sampling rate for the input signals. The downconversion process is commanded by a UHD tune request in GNU Radio, where the user specifies a center frequency (in Hz) and a local oscillator (LO) offset (in Hz). The center frequency represents the difference between the input clock frequency (i.e. 10 MHz) and the desired near-baseband frequency. For example, a center frequency of 9,999,927 Hz is used to mix a 10 MHz clock input down to 73 Hz. Shifting the input signal down by the center frequency is achieved in two stages. First, the input signal is mixed with the LO, which is offset from the center frequency by the user-specified frequency. Again, the analog mixing process does not fully shift the input signal to the desired near-baseband frequency. Assuming the same 10 MHz clock signal, a 9,999,927 Hz center frequency, and an LO offset of 250 kHz, the analog mixer will shift the input signal down to 250,073 Hz (i.e. 73 Hz + 250 kHz). The digital down-converter then shifts the signal by the LO offset frequency, producing a signal at the desired near-baseband frequency. In the previous example, the digital down-converter shifts the output of the analog mixer by 250 kHz to produce the desired 73 Hz signal.

3.7 OCXO Frequency Characterization

An OCXO is often used to realize timescales due to the high quality short term stability of the oscillator [24]. The IEM of our clock ensemble is realized by applying small frequency adjustments to an OCXO, a process hereafter referred to as clock steering. Each OCXO has an electrical tuning input which changes the frequency of the oscillator as a function of applied voltage. The frequency response of each OCXO to a predefined voltage profile was measured experimentally to determine the slope of the voltage / frequency response curve. This curve is used in the steering model to compute voltage adjustments based on measured phase differences between the OCXO and the target state.

Adjusting the OCXO Frequency

The OCXO tuning voltages must be applied to the tuning pin without human interaction in order to implement a closed-loop system. Two options for voltage adjustments were considered: a programmable benchtop power supply and a digital to analog converter (DAC).

Benchtop Power Supply The CONTACT team purchased a GW Instek GPP4323 power supply for this purpose, as the GPP series of power supplies supports multiple I/O ports and uses a standardized command syntax for remote control. Custom python scripts were developed to programmatically control the bench-top power supply. The pyvisa python library was used as it was designed for interfacing with and controlling lab instruments using SCPI command syntax. This python script accomplishes three main functions: turns on the power supply and waits for the OCXO to warm up; once the OCXO is warmed up, it steps through a user-defined voltage range at the chosen step size; it then sets all outputs to 0 before shutting off the output channels. The user can then shut off GNU Radio Companion which ends the data logging. While the script is running the measured supply voltage, tuning voltage, and timestamps are written to a csv file.

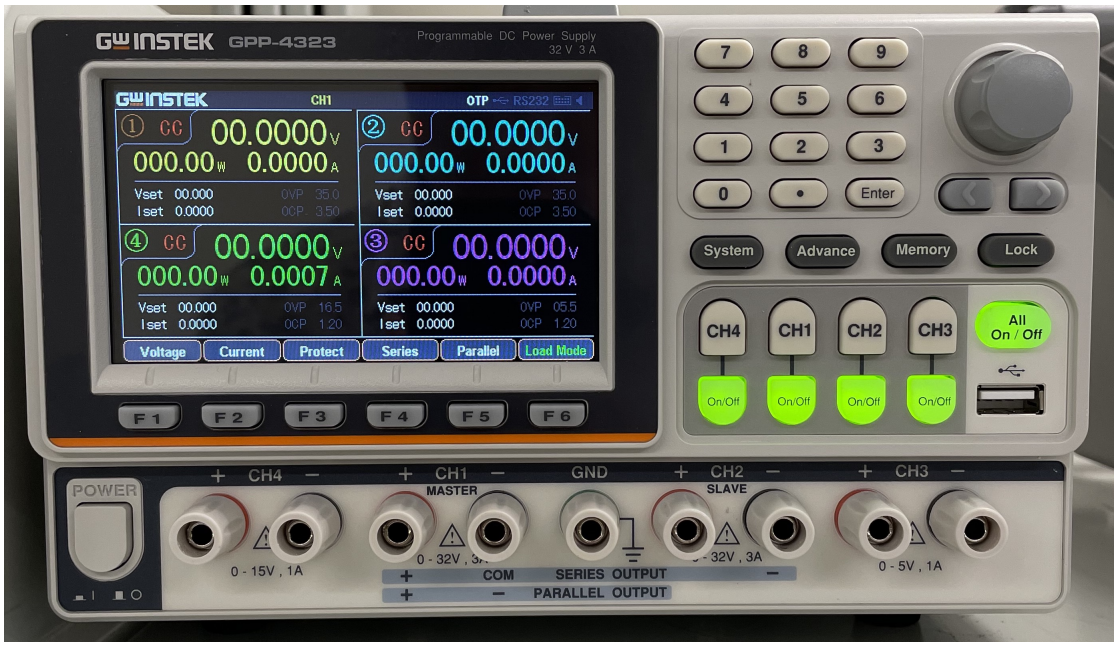


Figure 36. Benchtop Power Supply

18 bit DAC A low noise 18 bit DAC was acquired as an alternative to the bench-top power supply for frequency adjustment voltage commands. A development board from Analog Devices known as a "Linduino" allows users to interface with the DAC as if it were an Arduino device, enabling quick script development. Scripts were written in this Arduino environment to sweep across voltages with a user defined range and step size.

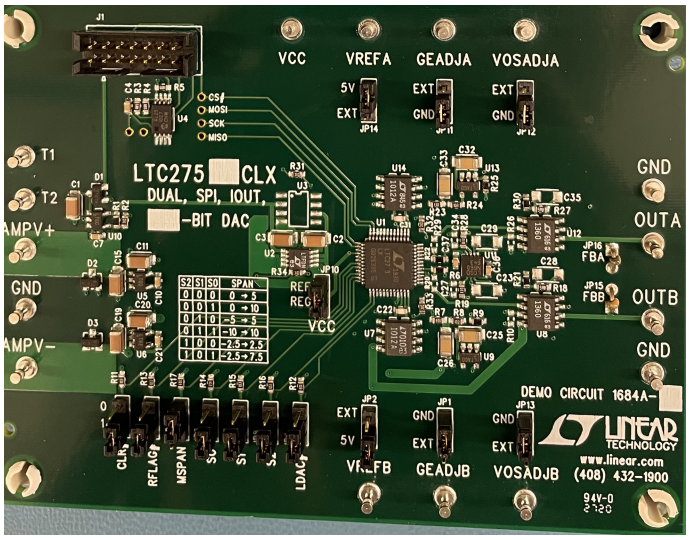


Figure 37. Digital to Analog Converter
https://www.overleaf.com/project/62a7822b46ccc367285809ac

3.8 Communication Protocols for Time Synchronization

Applications such as financial transactions, emergency services, and control systems require the communication of accurate timing. Network protocols provide the communication structure for time synchronization between machines by disseminating a reference clock through a hierarchy of network levels, or Stratum. The top of the hierarchy is referred to as Stratum 0 and consists of the reference clock or time scale, usually a device that can realize UTC. Each subsequent Stratum will synchronize clocks to the previous Stratum until Stratum 16 is reached, at which point the device is considered un-synchronized. A jump in a Stratum will result in a corresponding decrease in the communicated time precision, however more robust protocols are designed to mitigate this loss.

In theory, any set of devices that transfer data with modern communication protocols should be able to also implement network-based time synchronization. Proposed clusters and constellations of spacecraft have planned inter-satellite communication capabilities, suggesting the potential for time transfer via network protocols. The following protocols were considered to examine if communication protocols for time synchronization is viable for modeling multi-platform time transfer, as in a satellite cluster or constellation: network time protocol (NTP), precision time protocol (PTP), and White Rabbit (WR). Each version improves time synchronization precision by approximately three orders of magnitude, with WR operating at nanosecond precision.

Communication protocols were determined to have limited immediate applicability to our laboratory environment. The general method of clock synchronization is in a hierarchical

architecture which does not accurately represent a mesh communication topology of multiple identical clocks transferring timing information to each other. The only protocol that provides synchronization at relevant PNT levels is White Rabbit; however, this requires significant hardware upgrades in our lab environment to utilize and has limited commercial availability. We include a summary of our study here for future reference.

3.8.1 NTP

Due to the advent of the Internet, billions of devices are interconnected and communicating data. Accurate time synchronization across machines ensures this system operates properly. Professor David L. Mills from the University of Delaware addressed this problem with the creation NTP. This protocol is ubiquitous and exists as a pre-installed daemon on almost all computer systems. NTP was designed to provide synchronization using a messaging system between a client and a server [25]. By default, a computer will synchronize its time with a NTP server, which is stratum 1 or 2 clock. The synchronization process is initiated by the client sending a sync request message to the server. A request is time stamped using the client clock and is labeled as t_0. Once the sync request message is received by the server a return message is sent to the client. A server time stamp is generated for the received message and the outbound message, t_1 and t_2 respectively, and communicated back to the client. The client timestamps the server reply with respect to its local clock, t_3. Simply stated, the client asks the server what time it is and the client adjusts its clock to match the reply. However, due to variable distances and processing delays in the synchronization process, the reply is limited in accuracy. The four time stamps, t_0, t_1, t_2 and t_3, are used to determine the time offset and request message delay to compensate for this uncertainty. Equations 27 and 28 below are used to calculate the time offset α and round-trip delay β.

$$\alpha = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \tag{27}$$

$$\beta = (t_1 - t_0) + (t_2 - t_3) \tag{28}$$

The client clock is slewed over time to match the server clock using the offset and round-trip delay. Additional NTP requests will be generated to fine tune the server time to millisecond synchronization. Frequency drift for clocks at a lower Stratum will require continual NTP requests to retain clock accuracy. Depending on what type of network is used the accuracy can be improved. Time transfer accuracy in a network is sensitive to packet delay variation (PDV). Packets can be transmitted from host to host on unpredictable network paths and subjected to varying processing delays. The less nodes between a client and host for time transfer the less PDV will impact the accuracy of time synchronization. Short distance

requests to stratum 0 time sources can yield sub millisecond time accuracy; however, NTP is primarily designed for long distance internet time transfer and is therefore relegated to time accuracy in the tens of milliseconds. Although millisecond time synchronization is acceptable for computer networks, a navigation system with this level of timing accuracy would result in range errors on the order of 300 kilometers, rendering it ineffective.

3.8.2 PTP

IEEE 1588 PTP uses an enhanced messaging system and specialized hardware to meet the demanding timing needs of industrial applications, such as power management [26]. Contrary to NTP, a PTP synchronization is initiated on the server side. The messaging system generates a server time-stamped synchronization message, t_0, that is communicated to the client. An additional follow up server time-stamped message is generated by the server and sent to the client a few seconds later, t_1. The client clock will adjust its initial time, t_{ci}, to an intermediary time, t_{cm}, that closely resembles the server time using t_0 and t_1, as show in Equation 29.

$$t_{cm} = t_{ci} + (t_1 - t_0) \tag{29}$$

The new client intermediary time does not account for delay in the network. A client-side time-stamped delay request message is generated and sent to the server, t_2. After the server receives the message a server-side time-stamped delay response message is generated and sent to the client, t_3. The difference between the time stamps is calculated, divided by two, and then added to the intermediary server time. This will yield the final server-client synchronized time, t_{cf}, as show in Equation 30.

$$t_{cf} = t_{cm} + \frac{t_3 - t_2}{2} \tag{30}$$

PTP synchronization involves twice as many messages as NTP to better account for system delays and enables better accuracy in the transferred time. An additional reason for improved time precision of PTP is a result of boundary clocks. NTP does not account for systematic delays in a dynamic network with multiple nodes and varying link distances. PTP measures these systematic delays by using boundary clocks at each node to preserve the synchronization signal. In a IEEE 1588 network each node or switch contains a boundary clock that operates a PTP synchronization instance. This synchronization instance allows the boundary clock to calibrate itself by recovering and regenerating timing from the previous clock, thus significantly reducing delay propagation through a network. Figure 38 illustrates the difference in NTP and PTP time transfer, with a PTP instance occurring between each

switch.

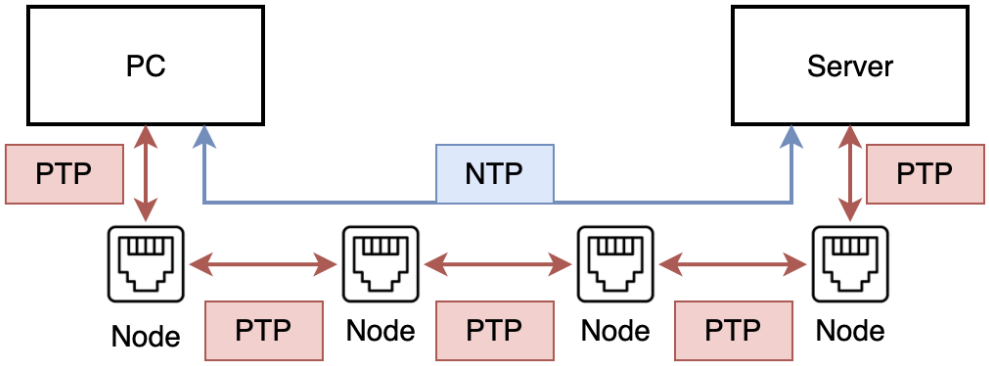


Figure 38. PTP Instancing Between Nodes

Another component of PTP that improves synchronization accuracy is hardware time stamping. Instead of time stamping the message at the software level like NTP, PTP works by injecting the time stamp at the physical layer [27]. IEEE 1588 standardized hardware uses a Time Stamping Unit (TSU) located between the Ethernet MAC and Ethernet Phy that marks the exact time a synchronization message is sent. This hardware-assisted time stamping reduces the operating system processing latency in the message exchange, thus decreasing PDV in the synchronization process. Microsecond time synchronization is achievable in PTP but requires the exclusive use of IEEE 1588 hardware. All switches, nodes, and machines in the network are required to be PTP certified to operate boundary clocks and hardware time stamping. IEEE 1588 PTP can provide time synchronization accuracy three orders of magnitude greater than NTP, however in a navigation system this still yields unacceptable levels of range error.

3.8.3 White Rabbit

The European Organization for Nuclear Research (CERN) operates the worlds largest particle collider, the Large Hadron Collider (LHC), consisting of a underground tunnel 27 kilometers in circumference. Linear accelerator injectors and particle detectors spaced hundreds of meters apart in the LHC require nanosecond accurate timing. Previously, CERN relied on distinctive clock synchronization systems for different systems. Managing multiple unique timing systems was resource intensive, prone to error, and difficult to scale. White Rabbit was designed to address these issue by providing reliable, nanosecond clock synchronization and picosecond stability over CERNs long distances via flexible, Ethernet based hardware [28]. Time transfer laboratories such as NIST, the National Institute for Standards and Technologies, investigated utilizing WR to replace their campus-wide Coordinated Universal Time (UTC) distribution system. Their report concluded that WR meets the accuracy

and stability requirements for providing UTC(NIST) to systems across NIST campus that contribute to their UTC realization [29].

Similar to PTP, timing is disseminated via a hierarchical architecture with synchronization initiated by servers to clients. The following improvements are employed by WR to achieve nanosecond accurate timing: enhanced server-client messaging with frequency transfer over the physical layer and time stamping via phase measurements. The WR messaging system is comprised of the WR link initialization and the standard PTP synchronization message exchange. Syntonization of the client clock to server clock is achieved during the link setup using synchronous Ethernet (SyncE), a communications standard that allows for clock signals to be transferred over the physical layer. The initial WR synchronization process then uses PTP to measure a coarse round-trip delay between the client and server. Round-trip delay precision is then improved using Digital Dual Mixer Time Difference (DDMTD). The clock signal received by the client is transmitted back to the server and fed into DDMTD along with the original clock signal to calculate the phase difference. With the phase difference known, PTP timestamp precision is extended and the round-trip delay can be re-calculated. Next, the link asymmetry, precise one-way delay, and server-to-client clock offsets are calculated to fix any offset in the client. Finally, a feedback loop continually measures the server-to-client offset to compensate for any delay changes. WR can only achieve sub-nanosecond synchronization using gigabit optical links, but can operate in any Ethernet based network that uses SyncE. A core element of the network is the WR switch, which contains specialized hardware and software to perform the messaging, clock signal transmission and receive, and calculate offsets for compensation [30]. If a non-WR switch is connected to the network then the link will be recognized as a connection to a standard router and the WR synchronization process will fail. These features of WR allow for nanosecond time synchronization, which is equivalent to 30 centimeters of range error. Time synchronization at this level makes WR a potential candidate for time transfer protocols between satellites providing navigation services.

3.8.4 Limitations

Network based clock synchronization protocols are designed to distribute timing from a single, highly stable clock treated as the reference. The hierarchical relationship between synchronizing/syntonizing oscillators with a single truth clock is not an accurate representation of the inter-satellite architecture we are developing in the lab. For a satellite cluster, several clocks with near-identical stability will be transferring timing signals to each other. In this scenario there is no absolute reference clock as implied by network based time protocols. There may be other applications where this type of hierarchical time transfer system can be used. An example would be a constellation with one platform operating with a substantially

better clock acting as a reference clock to other platforms in the system. Another limitation to network protocols is their inflexibility to alterations in design. If the hierarchical time distribution mechanism could be adjusted to better model a mesh topology, then it may be applicable in multi-platform time-transfer with similar clocks. However, due to the network based clock synchronization protocols being built on time transfer for computer networks, which are inherently hierarchical, the ability to re-work the protocol is limited. Further research would be required into mutual synchronization methods such as those employed by telecommunications systems. Such systems utilize components of the network based time protocols, like SyncE, to synchronize clocks in non-hierarchical network. This technology has the possibility to be used in a lab environment to model multi-platform time synchronization but requires further investigation. White Rabbit is best suited for precise, multi-platform time transfer due to its nanosecond time-transfer precision, but the protocol is still maturing and has limited presence in the commercial market. Expensive upgrades, such as the addition of WR switches and optical Ethernet cables, to the COMPASS lab would be required. Due to these limitations we determined that software defined radios with radio frequency signals was a better solution for modeling multi-platform time transfer.

4 RESULTS AND DISCUSSION

4.1 Testbed Software Models

4.1.1 Clock Profiles

The simulated behavior of three CSACs initialized with 0 phase and frequency error is shown in Figure 39. After this 24 hour simulation all of the clocks are within ± 3 μs of true time. The corresponding Allan deviation curves are shown in Figure 40, along with the CSAC stability specifications. In initial simulations, the CSAC profiles were generated using the exact specifications from the manufacturer. Once the clock characterization system was operational, multiple tests were performed to assess the true stability parameters for each clock. Each of the curves is designed to be slightly more stable than the specification, reflecting the measured stability of the CSACs in our timing lab. The measured stability specifications should be used in the clock model code such that the simulated system behavior closely represents the truth.

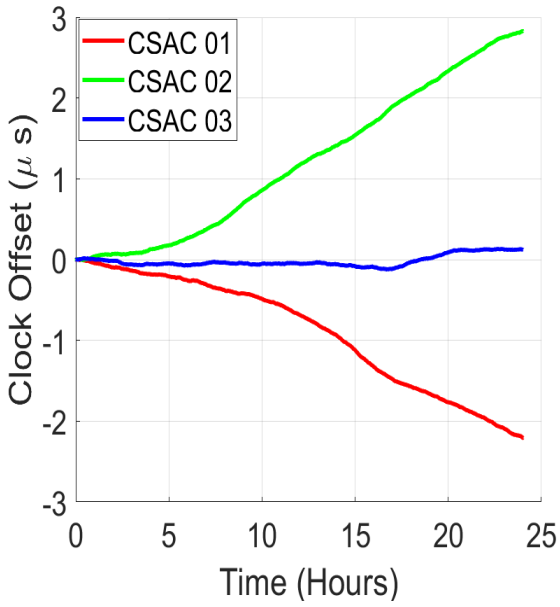


Figure 39. Simulated CSAC Time Series

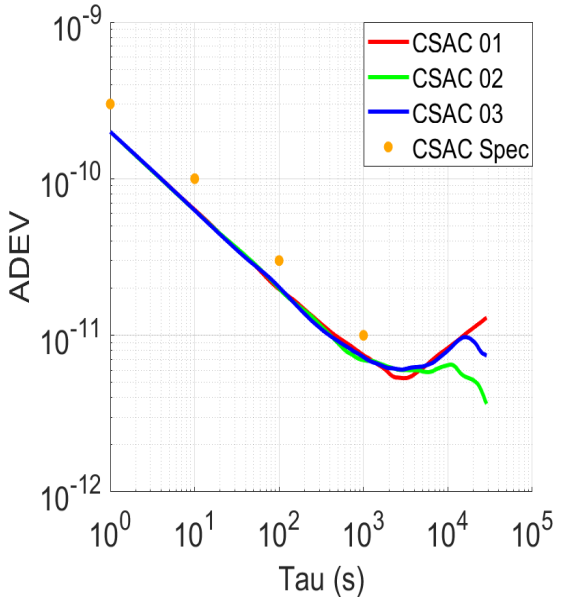


Figure 40. ADEV of Simulated CSACs

4.1.2 Steering Simulation

The two design parameters that affect the steered system response are the control rate for frequency adjustments and the stiffness of the response to the perturbations. In simulation, there are not constraints on the control rate; however, limitations in the hardware will affect both the minimum control rate and the quantization of the applied voltage on the frequency control pin. The optimal control rate will depend on latency of the feedback in the power system, the settling time of the voltage source, and the stability of the OCXO with respect to the target oscillator. The stiffness of the response will determine the gain parameters used to compute the voltage adjustment. The system performance was first modeled in simulation to predict the effect of varying control rates and gain values on the steered signal response.

Figures 41 and 43 show the result of five OCXO steering simulations with control rates of one second and various gain matrices. The red curve in both figures is CSAC 01 - the target state - and the other curves are the time series of the steered OCXO. The black curve in Figure 41 corresponds to the gain matrix that produces closed loop eigenvalues $\Lambda \approx 1$, an underdamped response. The difference between the steered OCXOs and the target state are shown in Figure 42. All of the steered signals are zero mean with various oscillation magnitudes about zero.

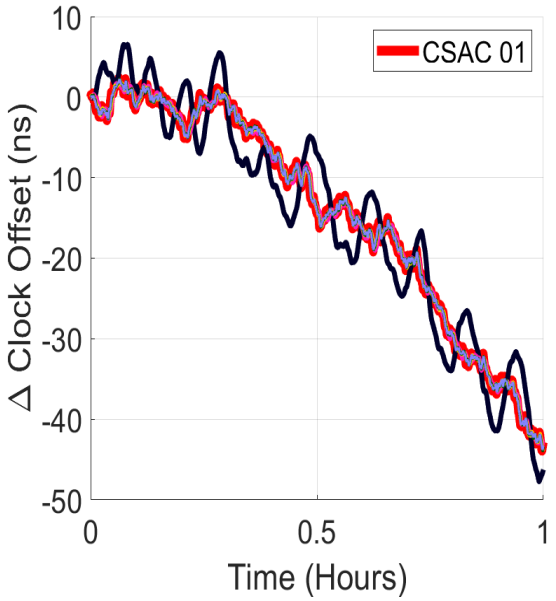


Figure 41. Time Series of Steering OCXO to CSAC 01 - 1 hour

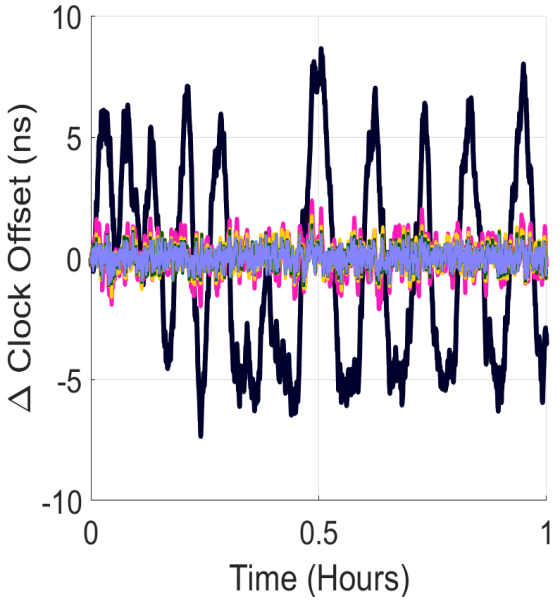


Figure 42. Steered OCXO Curves Minus CSAC 01 - 24 hours

The best response is challenging to select from the time series. The Allan deviation is a clearer metric to use when comparing the effects of different control parameters. The ADEV curves for each of the control simulations is shown in Figure 44. As the eigenvalues of the system response become stiffer, the ADEV of the steered response retains less of the short term stability of the OCXO but reaches the target state faster. With less stiff eigenvalues the short term stability of the OCXO is preserved, but oscillates about the target state at longer averaging intervals. The best system response corresponds to some intermediate value between these two extremes. Once a desirable simulated response is created, the control parameters can be incorporated into the hardware for additional testing.

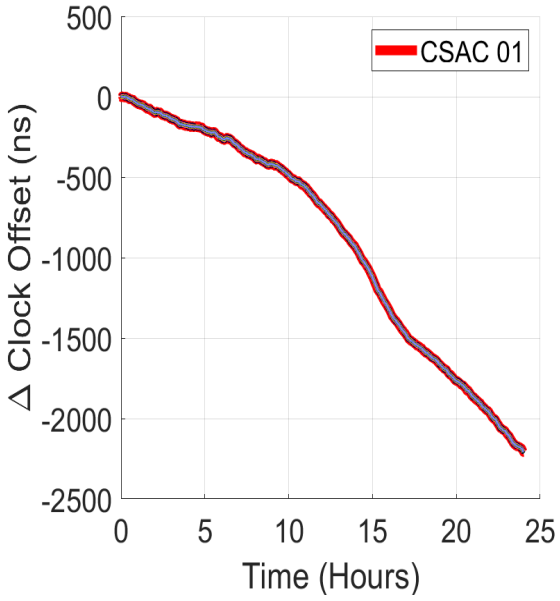


Figure 43. Time Series of Steering OCXO to CSAC 01 - 24 hours

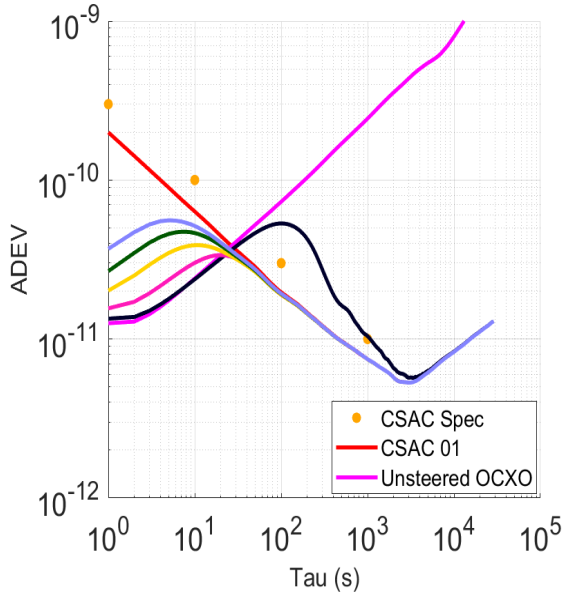


Figure 44. ADEV of Steered OCXOs, CSAC 01, and Unsteered OCXO

4.1.3 IEM Steering

The phase and frequency estimates of the OCXO are used to compute a control command that will adjust the OCXO frequency. The computation of the control command is based on the pole placement method [21]. Two of the parameters that a control designer can use when designing the response of a system are: 1) the eigenvalues of the state dynamics and 2) the control rate. The simulation was modeled after the clock ensemble testbed, using three CSACs as the ensemble member clocks and an OCXO as the steered signal source. The stability characteristics of the clocks were chosen based on the measured process noise parameters from previous clock characterization experiments. As shown in Figure 46 the simulated OCXO is more stable than the simulated CSACs up until 10 seconds. These particular stability parameters for the basis of a steered signal are desirable since with the designed control response the steered signal will have the short term stability of the OCXO and the long term stability of the IEM.

The simulated steered signal response below has a control update interval of 1 second and a quantized frequency step size based on the specified precision of the DAC. Slower control update rates resulted in system responses that did not approach the IEM in a reasonable time, if at all. Faster control update rates were not realizable due to the hardware limitations of the testbed, and thus were not simulated.

Figure 45 shows the time series behavior of each simulated CSAC, the IEM, and the steered

OCXO. The steered OCXO matches the IEM quite closely over the duration of the test. The Allan deviation of each curve is shown in Figure 46. At short averaging intervals, the steered OCXO has similar stability behavior as the unsteered OCXO. Over longer averaging intervals, as the clock is steered, the ADEV of the steered OCXO approaches the IEM.

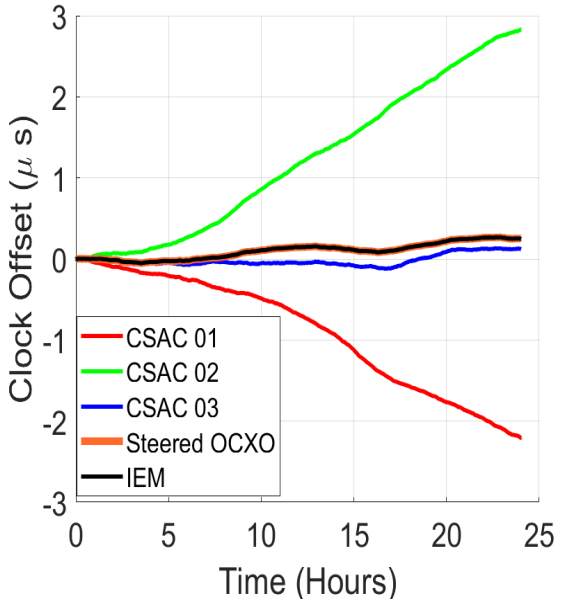


Figure 45. Time Series of Simulated CSAC, IEM, and Steered OCXO

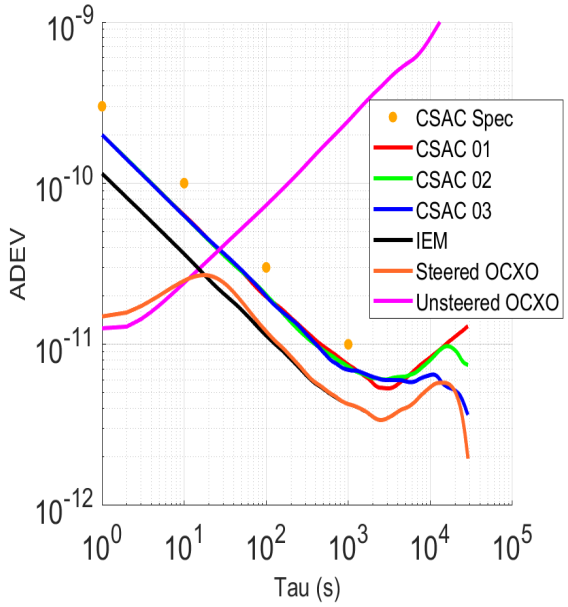


Figure 46. ADEV of Simulated CSACs, IEM, and Steered OCXO

4.2 Experimental Clock Phase Measurements

4.2.1 Effect of the Local Oscillator

All four clock signals have a common trend that represents the behavior of the local SDR clock. If Allan deviation values were computed on these phase curves, the resulting stability values would not match the Rb or the CSACs, but rather that of the local SDR clock. The most stable clock in Figure 47, the Rb, is subtracted from all other CSACs and the results are shown in Figure 48. The difference of clock signals causes the local clock contribution to drop out and shows the frequency drift of each CSAC with respect to the Rb oscillator.

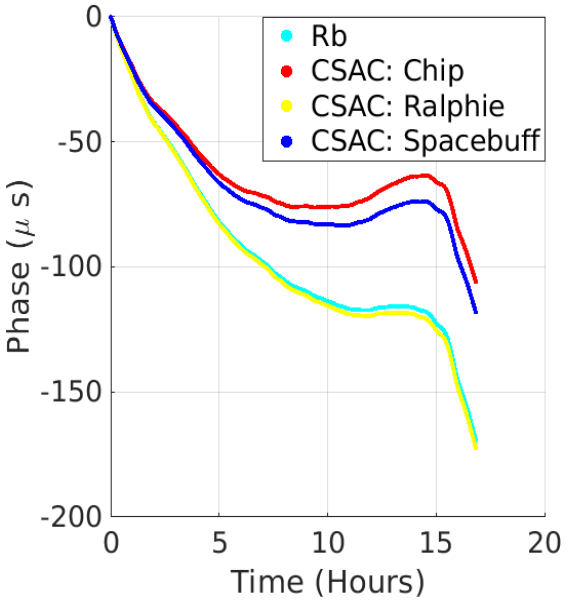


Figure 47. Measured Clock Phase with Common SDR Clock Contribution

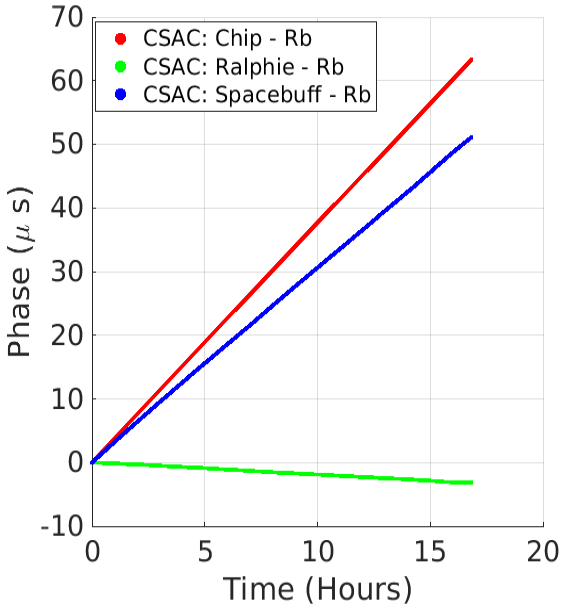


Figure 48. Measured Clock Phase against a Rubidium Reference

The Allan deviations of all curves in Figure 48 are shown in Figure 49. Despite having the largest slope in Figure 48, the CSAC Chip is seen to be the most stable and has the smallest ADEV of the three clocks. Spacebuff and Ralphie have similar frequency stability characteristics. Also shown on this graph is the frequency stability of the local oscillator on the N310 - the result of computing the ADEV of one time series curve in Figure 47. The black curve shows that the common effect of the SDR clock would obfuscate the true behavior of the CSACs at intervals greater than 10 seconds; differencing the clock measurements in Figure 47 causes the SDR clock contribution to drop out, as previously discussed.

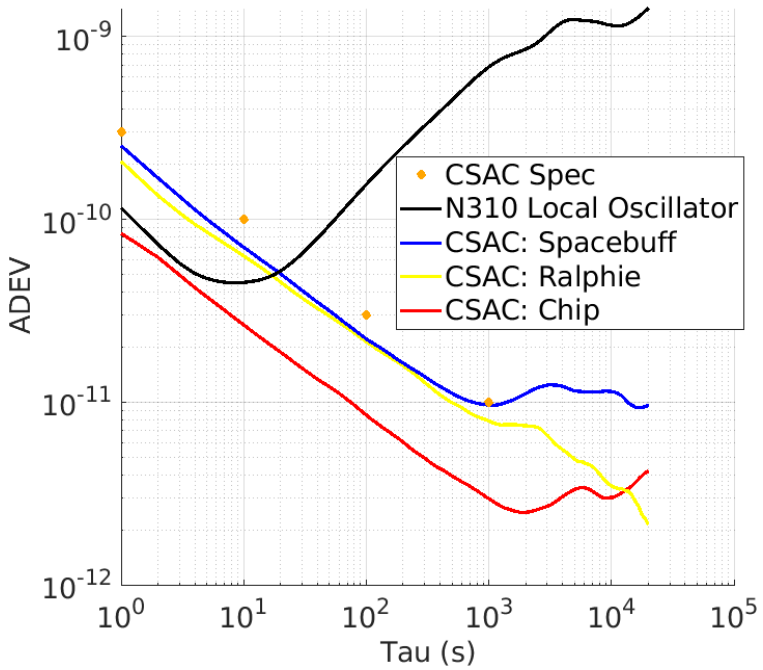


Figure 49. Allan Deviation of CSACs and LO of Ettus N310

4.2.2 Measurement Noise

The results from the measurement noise characterization tests are shown in Figure 52. The measurement configuration with the largest amount of noise is the cross transceiver configuration on the N310. Measurements on the same transceiver of the N310 have the smallest noise. Measurements made on the N200 are in the middle of these two values. At intervals longer than 10 seconds, there is no risk of any measurement configuration concealing the clock behavior. The only configuration that may obscure the clock behavior is the cross transceiver configuration at short time intervals.

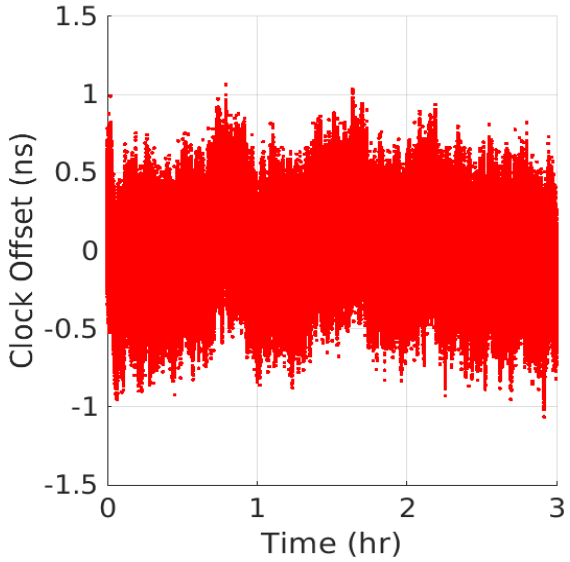


Figure 50. N310 Different Daughterboard Noise

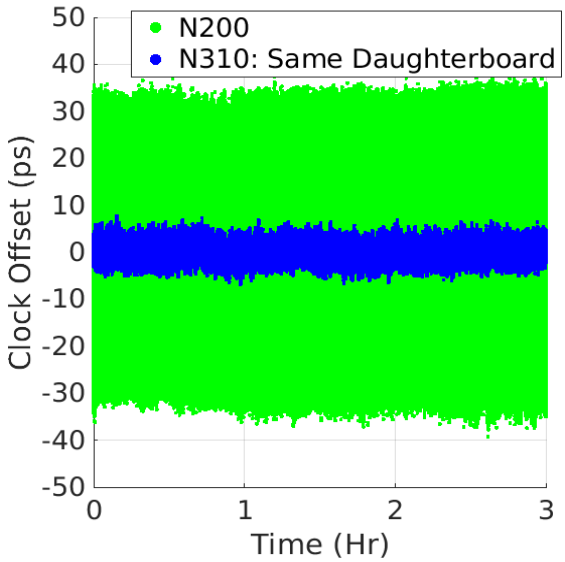


Figure 51. N200 and N310 Same Daughterboard Noise

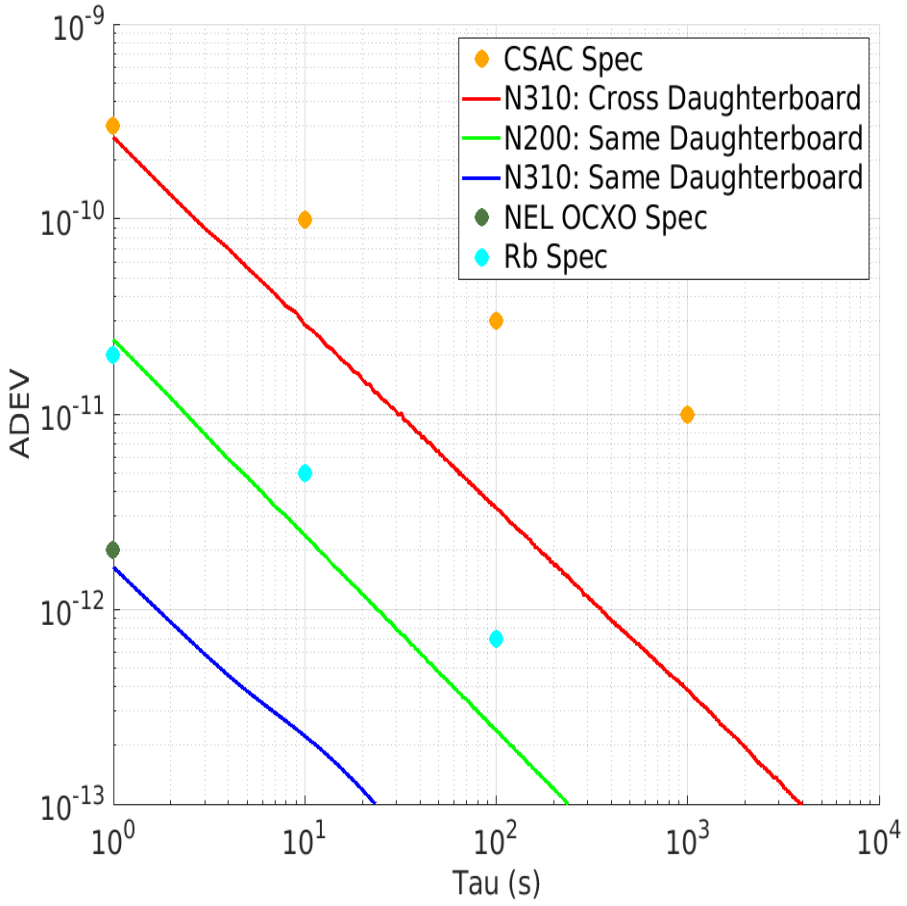


Figure 52. Allan Deviation of SDR Noise

4.2.3 Measured Clock Stability

Three categories of clocks are used in this project: CSACs, OCXOs, and a rubidium frequency standard. Three Microsemi SA.45s CSACs are used as the members of a small clock ensemble; two OCXOs are evaluated as potential candidates for the realization of the IEM - a Wenzel Associates 501-04609 and NEL Frequency Controls, Inc. 0-CMR-058IS-N-S-L; and a rubidium frequency standard, Stanford Research Systems FS-725, is used as a reference oscillator. The measured ADEV values and the specifications for the CSACs, OCXOs, and rubidium frequency standard are shown together in Figure 53. The data for the Rb clock is from a characterization campaign at NIST in 2019 where it was measured against a hydrogen maser. All other clocks on this figure are evaluated using the methods described in Section 3.6.

The stability values shown in Figure 53 are computed with respect to the Rb on the same transceiver of the N310 to use the system with the smallest noise contribution. The noise contribution is much smaller than the noise in the clocks for all measurement intervals, indicating that this measurement configuration does not obscure the clock behavior. The stability of the three CSACs is larger than the Rb at all measurement intervals, meaning the CSAC clock measurements against the Rb truly represent CSAC behavior. Interestingly, the measured short term ($\tau < 4$s) stability of both OCXOs is limited by the short term instability of the Rb. The true behavior of the OCXOs in the short term cannot be measured using our existing setup, but we assume the actual stability is better than what is shown.

The manufacturer stability specification for each clock, when available, is shown on Figure 53. All three CSACs are performing within the specification with Chip being slightly more stable than Spacebuff and Ralphie. The short term behavior of both OCXOs is obscured due to the short term limitations in the rubidium frequency standard. For averaging intervals longer than approximately 5 seconds, the free-running OCXOs both rapidly degrade and eventually become worse than the CSACs.

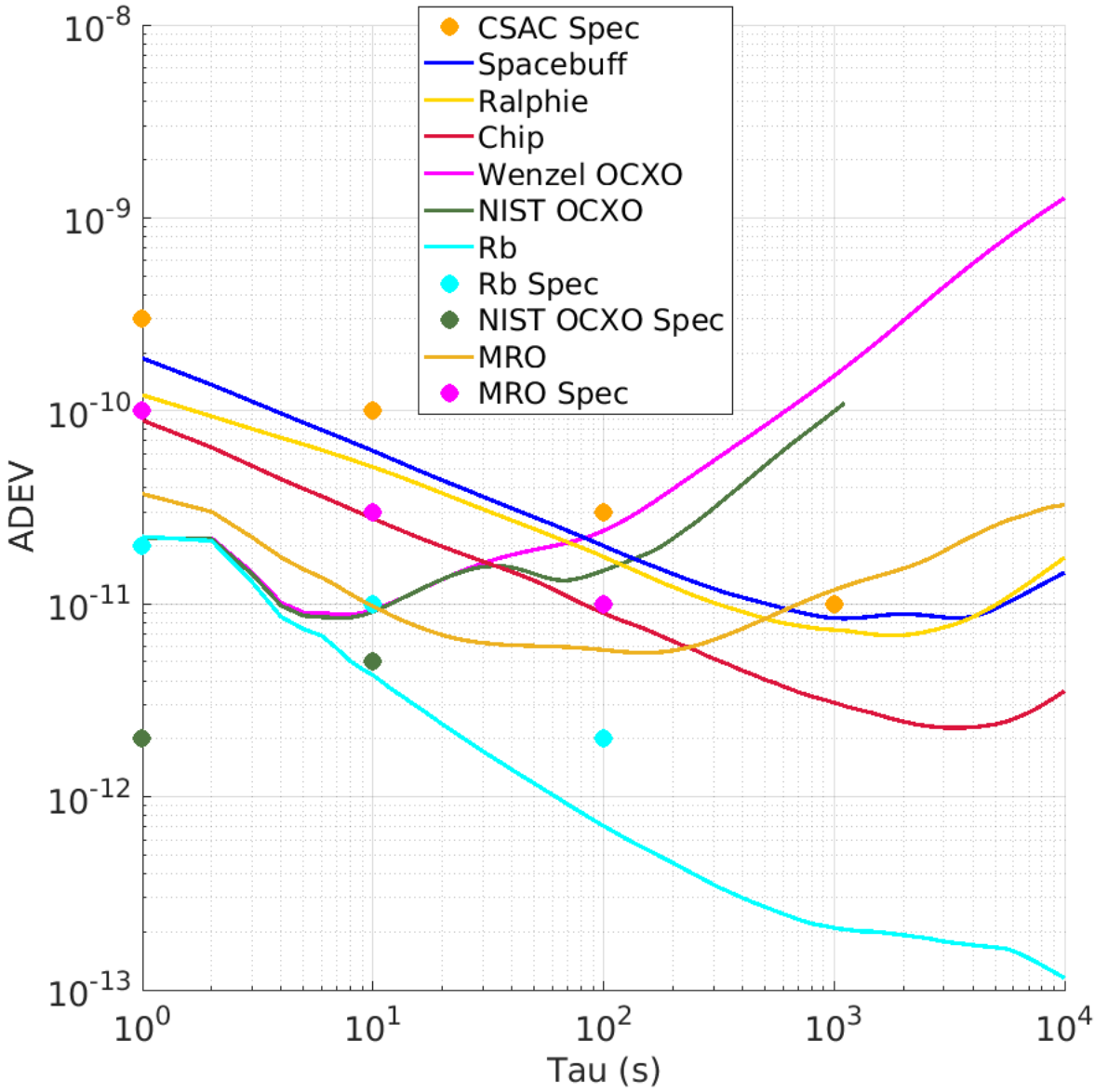


Figure 53. Stability of Clocks in CONTACT Project

Table 5. Measured CSAC ADEVs

Tau	CSAC: Chip	CSAC: Ralphie	CSAC: Spacebuff
1	$9 \cdot 10^{-11}$	$1 \cdot 10^{-10}$	$2 \cdot 10^{-10}$
10	$2 \cdot 10^{-11}$	$5 \cdot 10^{-11}$	$6 \cdot 10^{-11}$
100	$9 \cdot 10^{-12}$	$2 \cdot 10^{-11}$	$2 \cdot 10^{-11}$
1000	$2.5 \cdot 10^{-12}$	$7 \cdot 10^{-12}$	$9 \cdot 10^{-12}$

Table 6. Measured Clock ADEVs

Tau	MRO	Rb	Wenzel OCXO	NEL OCXO
1	$4 \cdot 10^{-11}$	$2 \cdot 10^{-11}$	$2 \cdot 10^{-11}$	$2 \cdot 10^{-11}$
10	$1 \cdot 10^{-11}$	$4 \cdot 10^{-12}$	$1 \cdot 10^{-11}$	$1 \cdot 10^{-11}$
100	$6 \cdot 10^{-12}$	$7 \cdot 10^{-13}$	$2.5 \cdot 10^{-11}$	$1.5 \cdot 10^{-11}$
1000	$1 \cdot 10^{-11}$	$2 \cdot 10^{-13}$	$1.5 \cdot 10^{-10}$	$1 \cdot 10^{-10}$

4.3 OCXO Frequency Characterization

The OCXO response to various voltage profiles was measured to compute an expected response for the steering model. Voltage profiles were generated with both the power supply and the DAC at step sizes ranging from 19 microvolts to 1 volt. The curve slopes are used in software models to compute a voltage adjustment in both the steering tests and later in the clock ensemble. The results for these tests are presented in Section 4.3.

4.3.1 Wenzel OCXO & Power Supply

The specification sheet for the Wenzel OCXO provides a fractional frequency response range of $\pm 2 \cdot 10^{-7}$ for applied voltages $\pm 5V$. The specification sheet states a negative slope, meaning +5V applied to the tuning pin of the OCXO would theoretically result in a FFO of $-2 \cdot 10^{-7}$. By dividing this result by 5V, we obtain an expected slope of $-4 \cdot 10^{-8}$ per volt.

Various tests were run on the Wenzel OCXO using different voltage step sizes: 1V, 100mV, 10 mV, and 1mV. This was done to ensure that the electrical tuning worked and then to determine if the OCXO had a minimum voltage step size. Results for the 100mV step size test are shown in Figure 54 below - the frequency response slope is negative and has a measured slope of $-5.17 \cdot 10^{-8}$, slightly different than the specification.

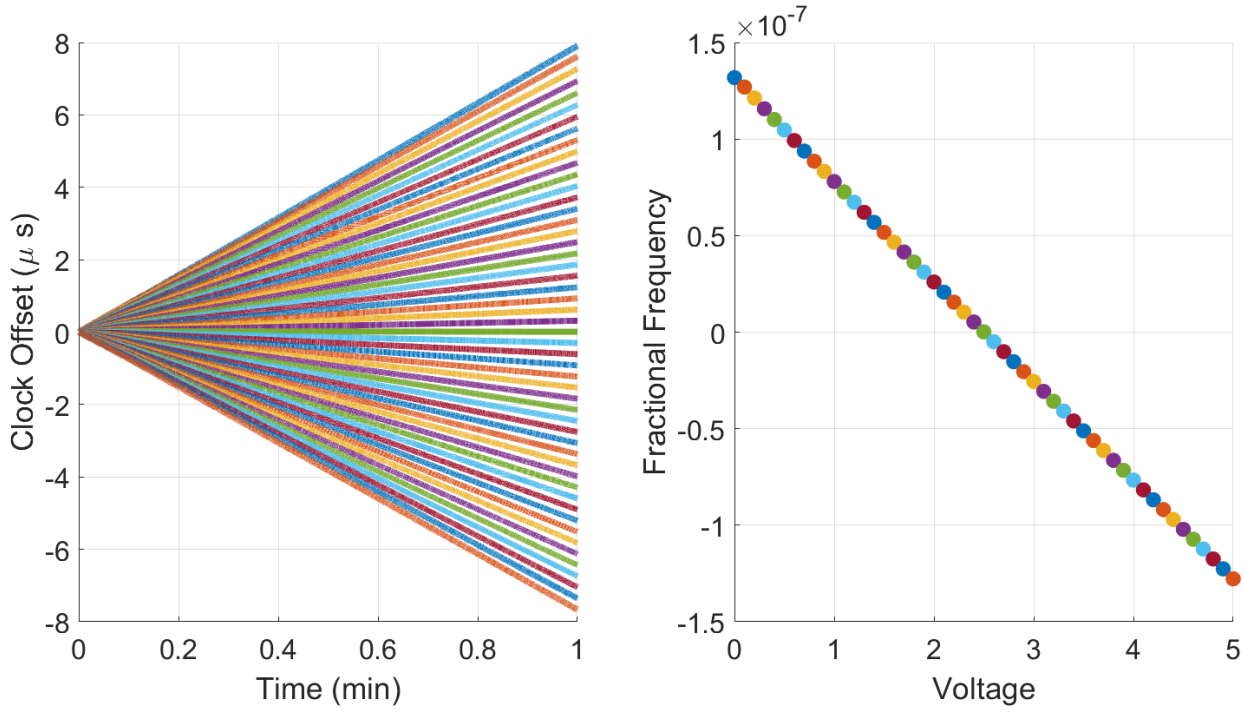


Figure 54. Voltage / Frequency Relationship for Wenzel OCXO and Power Supply

4.3.2 NEL OCXO & DAC

The frequency response of the NEL OCXO was characterized with the 18 bit DAC. The smallest voltage range for the DAC is ±2.5V which yields a voltage resolution of 19 microvolts. A voltage profile from -2.5 to 2.5V at a 0.5V step size was generated and held at each level for 5 minutes - the leading and trailing 30 seconds were ignored to eliminate the transient effects of changing voltage levels. The time series of the clock offset is shown on the left side of Figure 55 and is made up of 11 different lines, each corresponding to a different voltage level. As the voltage increases the frequency increases, creating a symmetric fan that is approximately centered on 0V. The corresponding fractional frequency offset for each voltage level is computed and shown in matching colored dots. The slope of the voltage / frequency curve is $2.19 \cdot 10^{-7}$.

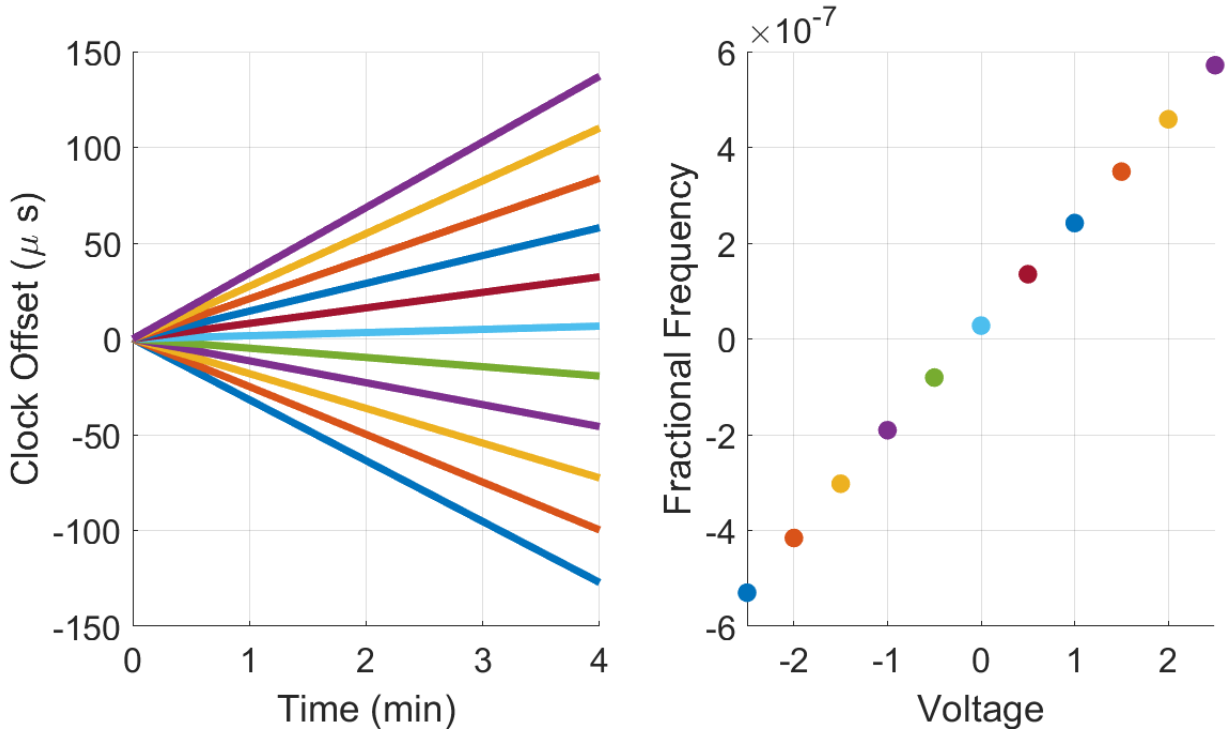


Figure 55. Voltage / Frequency Relationship for NEL OCXO and DAC

4.4 SDR with External Reference Oscillator

4.4.1 SDR Transmit Test

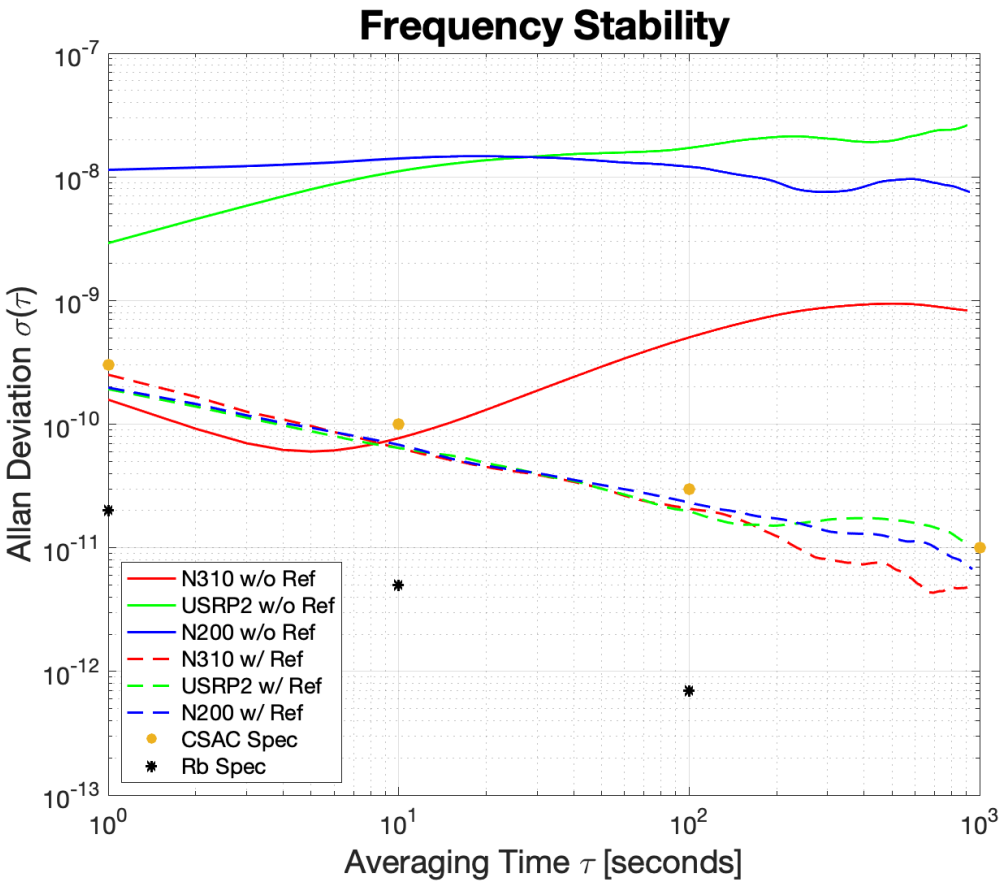


Figure 56. CSAC & SDR Transmit Test Configuration ADEV

Figure 56 shows the ADEV results from the SDR in the transmit test configuration. The solid lines indicate the frequency stability of the measured signal from the SDR that was being tested without an external reference clock. In all three cases the average stability is worse than the stability defined by the CSAC data sheet specifications. The N310 does exhibit slightly better stability at low averaging times compared to the CSAC specifications, but resolves to poor stability after the averaging time exceeds 10 seconds. Both the N200 and USRP2 display relatively poor stability and average about 2 magnitudes of order worse than the CSAC specifications. The dashed lines represent the stability of the measured signal from the SDR that was being tested with an external CSAC reference clock. All three dashed lines lie just under the CSAC specification markings, indicating that they are exhibiting CSAC like behavior. Due to the transmitted signals from the SDR under test exhibiting CSAC stability it can be concluded that the SDRs are successfully locking to an external

reference clock. The N310 displaying short term improved stability compared to the CSAC is indicative of the N310 operating with a relatively good onboard clock. However, this clock is unable to retain the stability that the CSACs can operate with for longer averaging times.

4.4.2 SDR Receive Test

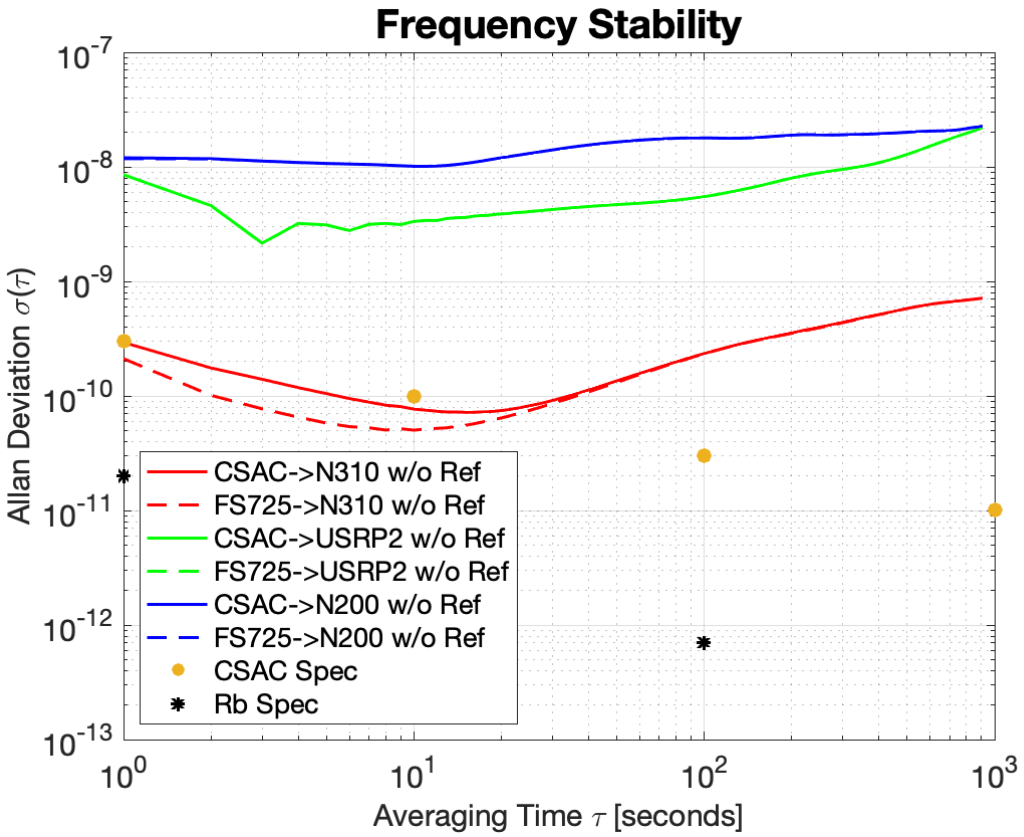


Figure 57. CSAC & SDR Receive Test Configuration ADEV - No Reference

The results for the CSAC and SDR receive test configuration examine the stability of each received signal on the SDR with or without an external reference clock. The solid lines on each plot represent the received CSAC signal and the dashed lines represent the received Rb signal. With no external reference clock the receive signals are limited by the stability of the local oscillator in the SDR. The CSAC and Rb stability curves overlap as they are both limited by the instability of the LO in the N200 and USRP2, as shown in Figure 57. In the N310s case, the LO is characterized by the Rb measurement and found to have short term stability that exceeds the CSAC specifications. This relatively good, short term stability of the LO allows the stability of the CSAC signal to be temporarily exposed. Eventually, as the averaging time increases, the CSAC and Rb curves converge and both are limited by the instability from the N310 LO.

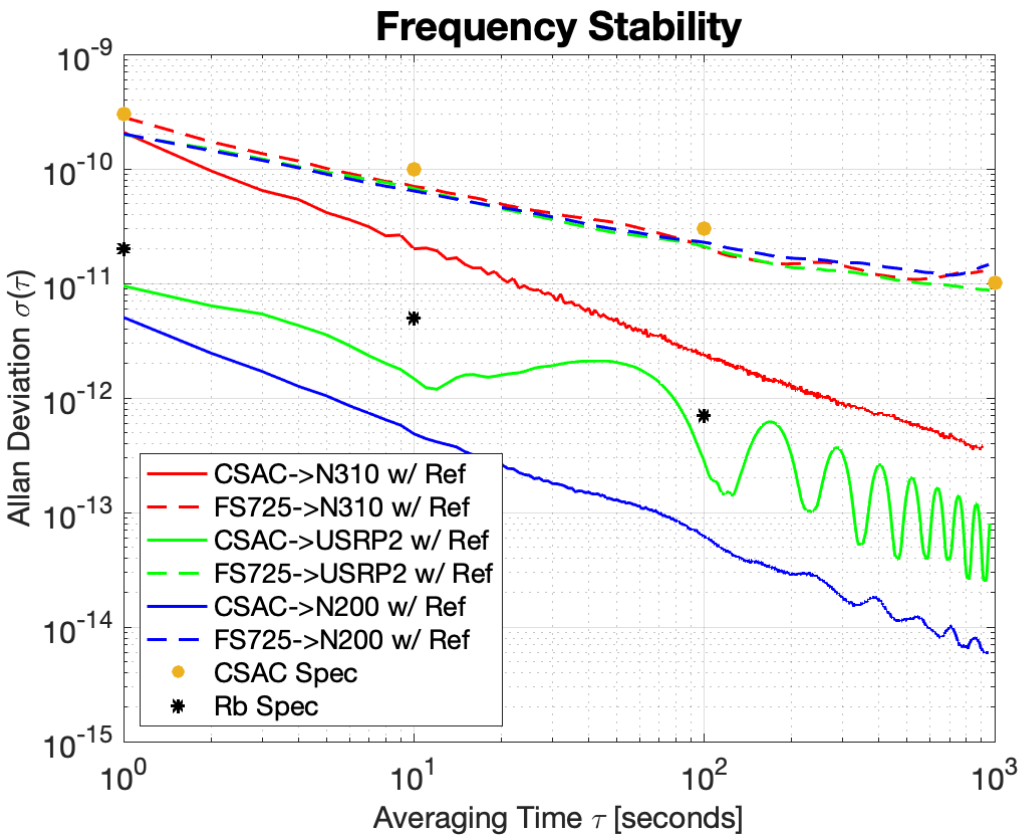


Figure 58. CSAC & SDR Receive Test Configuration ADEV - With CSAC Reference

With the CSAC added as a reference clock to the SDR both signals are now measured against the SDR LO tuned to the CSAC. The received CSAC signal is now being measured against itself resulting in the CSAC frequency variation being eliminated. This is visible in the three solid lines. Both the N310 and N200 CSAC phase measurements show linear ADEV curves with a slope of -1. This slope is representative of white noise in the receive channel chain. All three dashed lines exhibit CSAC like stability. This confirms the receied Rb signal is being limited by the stability of the SDR as it locks to the CSAC. There were two anomalies in the results, the first being the wavy feature of the USRP2 noise measurements and the second being the high noise level of the N310. The wavy feature of the USRP2 noise curve is thought to be sourced from signal interference in the receive chain. The high noise level shown by the N310 is possibly sourced from the on-board GPS disciplined oscillator (GPSDO). Ettus documentation on the N310 states that the GPSDO should be powered down to reduce phase noise. Despite the anomalies, the results confirm that all three lab Ettus SDRs are capable of locking to an external reference clock while operating as a receiver only.

4.4.3 SDR Transmit & Receive Test

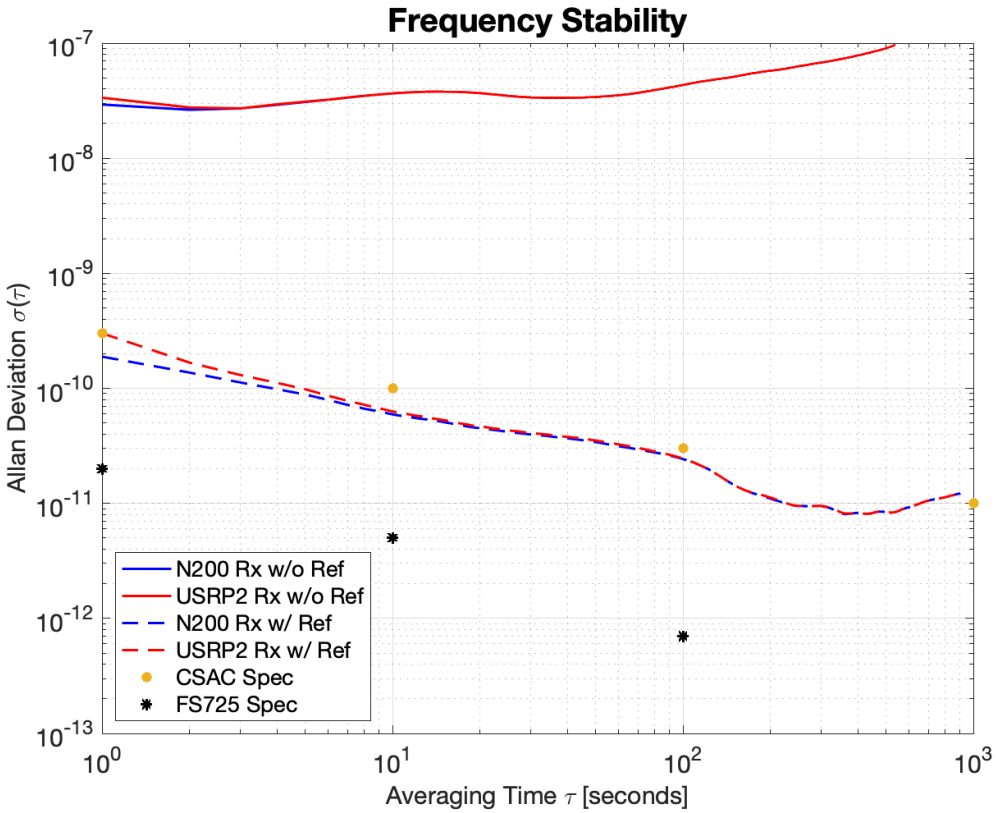


Figure 59. Simultaneous Transmit & Receive Test Configuration ADEV

The final test configuration involved the N200 and USRP2 simultaneously transmitting and receiving signals to each other. Figure 59 displays the results, with the solid lines indicating the SDRs tested without an external reference clock and the dashed lines with. Without an external reference clock the N200 and USRP2 are relegated to measuring signals with stability limited by their own local oscillators. Thus, the resulting stability measurements show very poor stability. This stability is similar to what was seen in the received signals for the N200 and USRP2 in Figure 57. With the USRP2 locked to a CSAC and the N200 locked to the FS725 the measured signals from each SDR match CSAC stability. The results prove that the USRP2 and N200 are capable of transmitting and receiving signal simultaneously, and both devices can successfully lock to an external reference clock while in a transceiver mode.

4.5 OCXO Steering to Known Reference

The closed loop steering functionality was initially demonstrated by steering an OCXO to a CSAC. The block diagram of the configuration used to steer the OCXO to a single CSAC is shown in Figure 60.

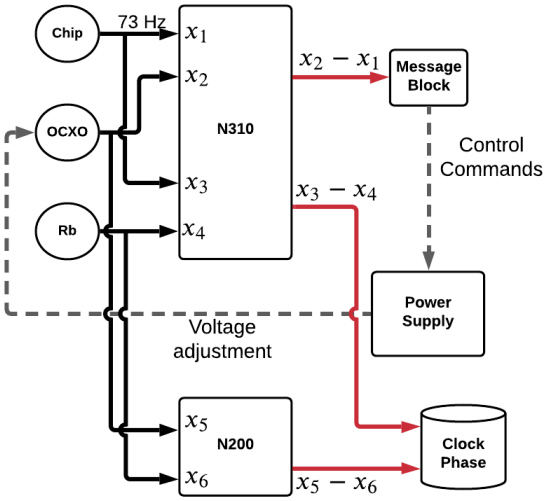


Figure 60. OCXO Steering Block Diagram

In this block diagram, the CSAC, OCXO, and Rubidium clocks are all connected to the Ettus N310. Each input signal to the N310 and N200 goes through the phase measurement processing chain as described in Section 3.6. The CSAC and OCXO are connected to the RF1 and RF2 ports on the N310 - the phase difference between these two oscillators is the error signal that we want to drive to zero. The second output of the CSAC and the Rubidium clock are connected to the remaining N310 ports to measure the true behavior of the CSAC. Finally, the OCXO and the rubidium were connected to the RF1 and RF2 ports on the N200 in order to measure the OCXO performance.

4.5.1 Wenzel OCXO & Power Supply

The custom power supply message block shown was used to steer the Wenzel OCXO. This block uses the pyvisa library and the measured frequency response of the OCXO to continuously adjust the output voltage on the GPP4323 power supply; this channel is connected to the electrical tuning pin on the OCXO, producing the requested change in frequency.

The results from the steering test with the Wenzel OCXO and power supply are shown in Figure 61. Both Chip and the OCXO are measured against the rubidium frequency reference. The effect of signal steering is clear as the steered OCXO approaches Chip at longer averaging intervals. For short averaging intervals there is a small region where the

OCXO is better than Chip, followed by a transient region where the OCXO is turning away from it's free running stability towards Chip. The behavior in this transient region is undesirable as the stability overshoots Chip by a significant amount. The source of this "servo bump" is primarily due to the imperfections in the voltage source. If the power supply had infinite precision, instantaneous voltage change, and zero settling noise, the bump size would be greatly reduced.

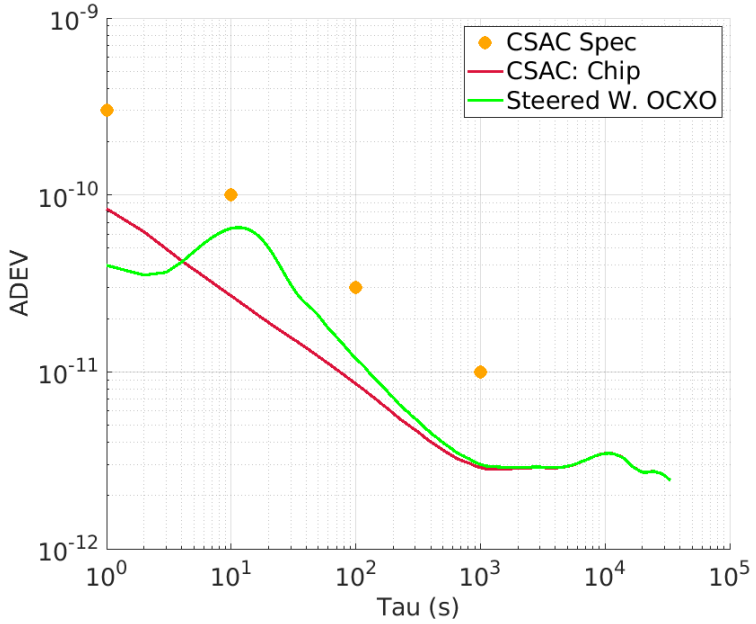


Figure 61. Steering Wenzel OCXO to Chip with Power Supply

4.5.2 NEL OCXO & DAC

Another steering test was conducted with the NEL OCXO and DAC. An Arduino script provided by Linear Technologies was modified to enable continuous, programmatic voltage changes in the DAC output. The results from the steering test are shown in Figure 62.

The control rate for this test was 8 Hz as opposed to 0.5 Hz in the previous experiment. Smaller gain values were used along with the faster steering rate; as a result, the steered signal in Figure 62 follows the unsteered behavior of the OCXO for longer and has a smaller servo bump.

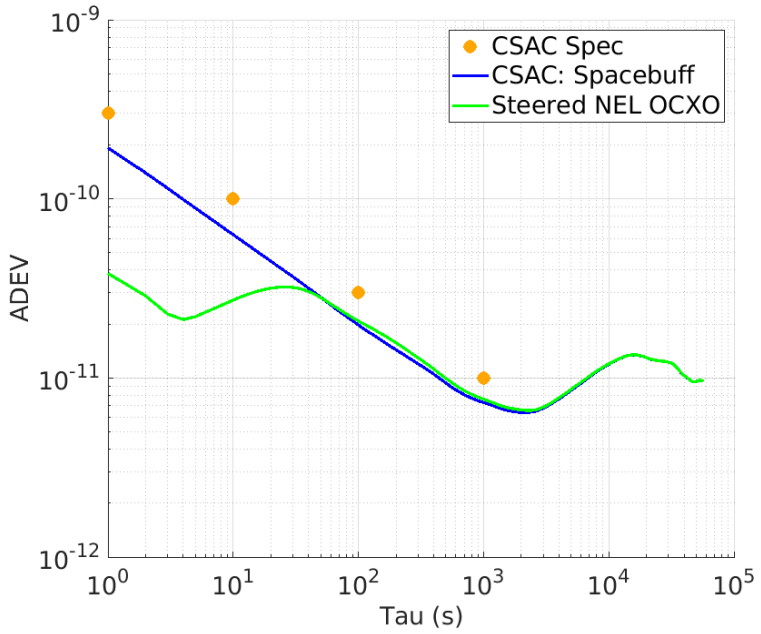


Figure 62. Steering NEL OCXO to Spacebuff with DAC

4.6 Clock State Estimation

The Kalman filter described in Section 3.4.3 was ported to Python and adapted for processing streams of data in real time. The block diagram of the Kalman filter is shown in Figure 63. The relative CSAC measurement inputs are shown on the left and are converted to units of time prior to entering the Kalman filter b lock. The six outputs are the phase and frequency estimates of each of the three CSACs. The code for the Kalman Filter Block is provided in Appendix 5.

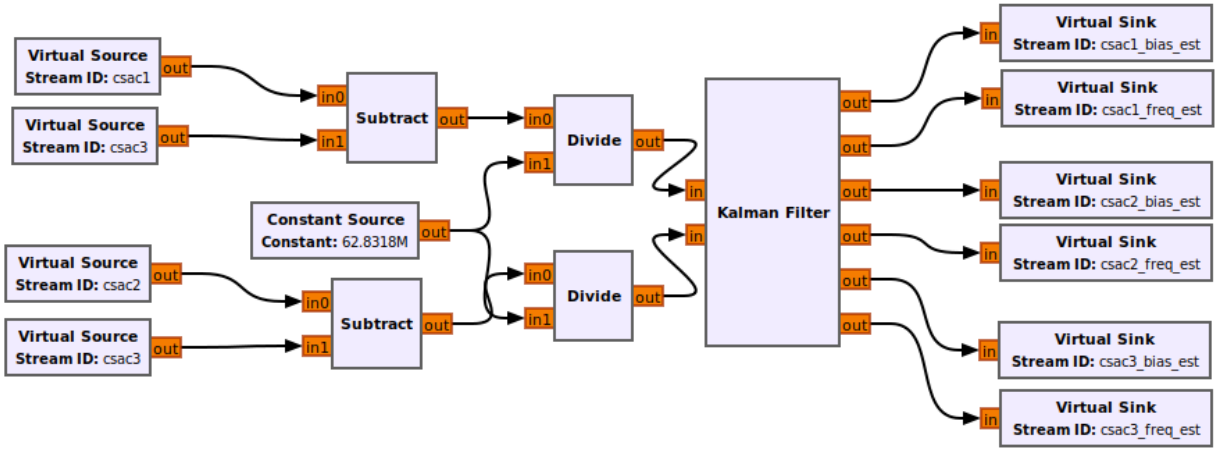


Figure 63. GNU Radio Companion Kalman Filter

An experiment was conducted with three CSACs and a rubidium frequency reference connected to the Ettus N310. Relative phase measurements were made between the CSACs and were filtered to produce estimates of the phase and frequency of each input clock. The Rb oscillator was used to measure the behavior of each CSAC independent of the filtering system; these measured results are compared to the estimated CSAC behavior to assess the filter performance.

Figure 64 show the time series of each CSAC as measured against the Rb and Figure 65 shows the time series of the CSAC phase estimates from the Kalman filter. The two plots do not match exactly, but the relative ordering of the three CSAC curves is the same in both plots. Additionally, the difference between each CSAC curve in the measured and estimated case seems approximately the same. This result makes sense, as the filter is using relative clock phase measurements to estimate the absolute clock states, which is an unobservable system. The absolute orientation of the estimated oscillator states cannot be known in the absence of another more stable reference.

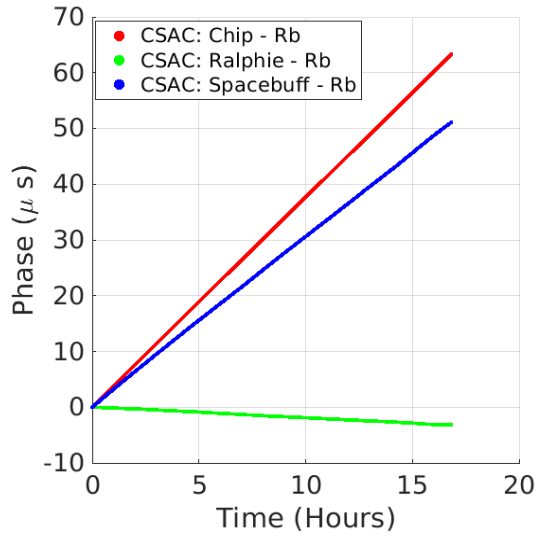


Figure 64. Measured Clock Bias with respect to Rb Reference

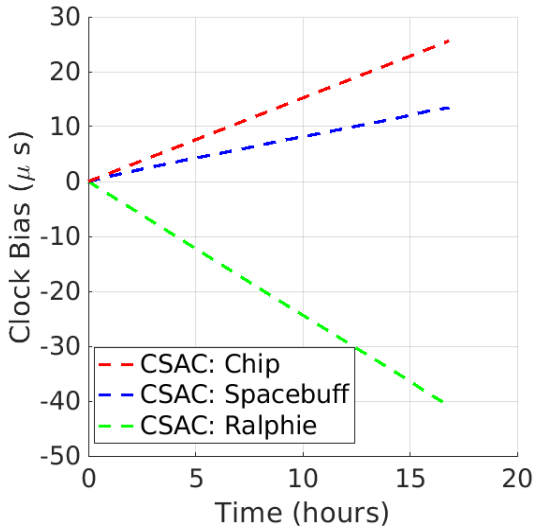


Figure 65. Estimated Clock Bias Output from Kalman filter

The difference between all curves in Figure 64 provides a time series of the relative behavior between the CSACs, which is a proper superset of the clock measurements input to the Kalman filter. Figure 66 shows the result of this difference operation with the measured values in solid, colored lines and the estimated values in black dashed lines. The agreement is quite close - these curves were again differenced to see how closely they match, shown in Figure 67. The result of this seems to be white Gaussian noise with sigma values close to the known measurement noise contribution from the SDR. In this test Spacebuff and Ralphie were on the same daughterboard and Chip was on a separate daughterboard - this is reflected

in Figure 67 where the same daughterboard measurements have a much smaller distribution than the two other cross daughterboard measurements.

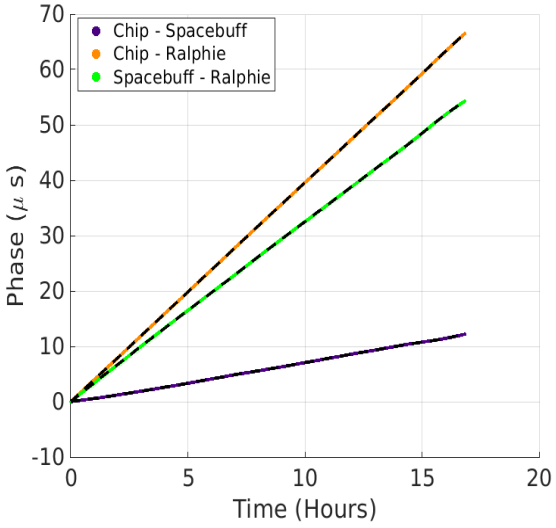


Figure 66. Difference Between Measured (solid) and Estimated (dashed) Clock Biases

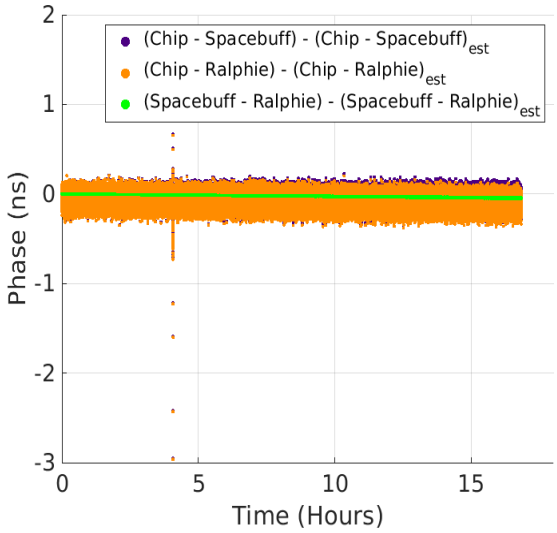


Figure 67. Difference of Clock Bias Differences

The Allan Deviation is computed on both the measured CSAC values in Figure 64 and the estimated CSAC values in Figure 65. The ADEV curves for the estimated clock states are similar to the measured values, perhaps a bit optimistic. The ADEVs for the estimated Chip values are significantly smaller than the measured value - this may be due to the presence of Chip in both measurements.

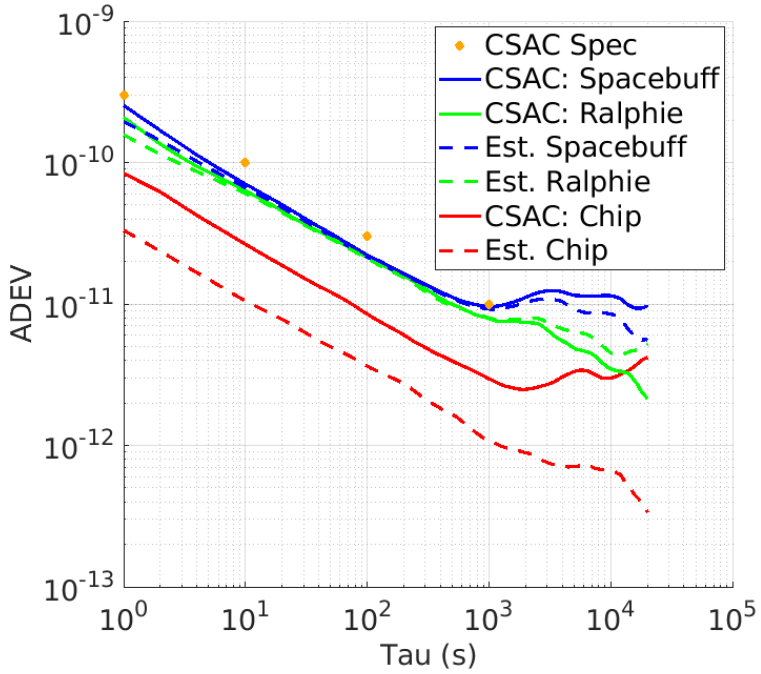


Figure 68. ADEV of Measured and Estimated CSACs

4.7 Clock Ensemble Testbed Integration

The individual components of the testbed described in the previous sections are combined to yield the fully integrated, closed loop clock ensemble testbed. A block diagram for the testbed is shown in Figure 69. The members of the clock ensemble are three CSACs and the OCXO is used as the realization of the IEM. The Ettus N310 is used to make clock phase measurements, a separate computer is used to run the custom signal processing, and a N200 is used to truth the OCXO signal with respect to a rubidium reference. The hardware setup in the timing laboratory is shown in Figure 70.

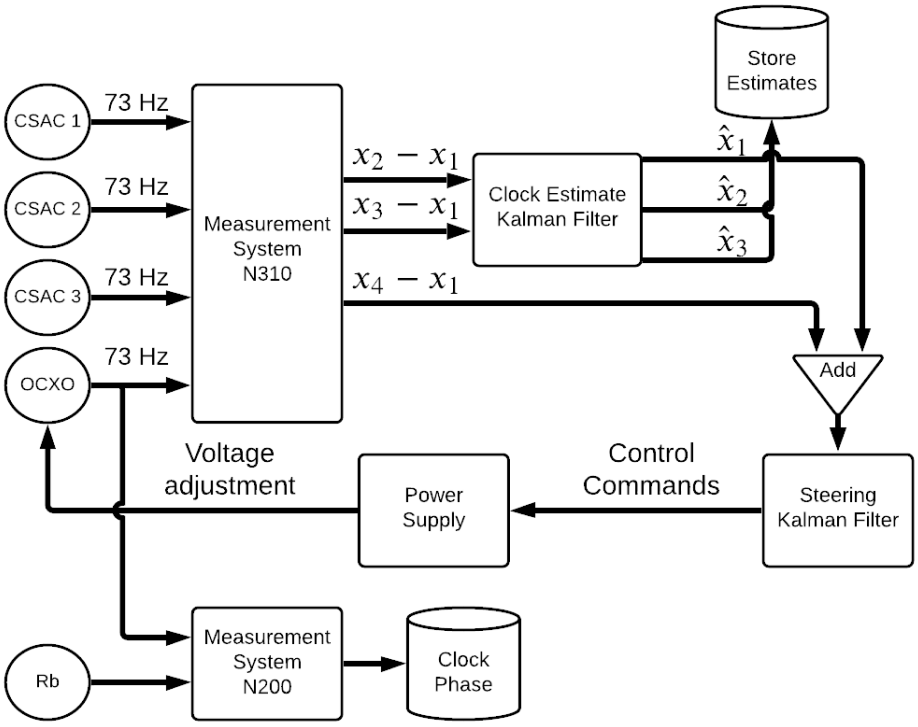


Figure 69. Clock Ensemble Testbed Block Diagram

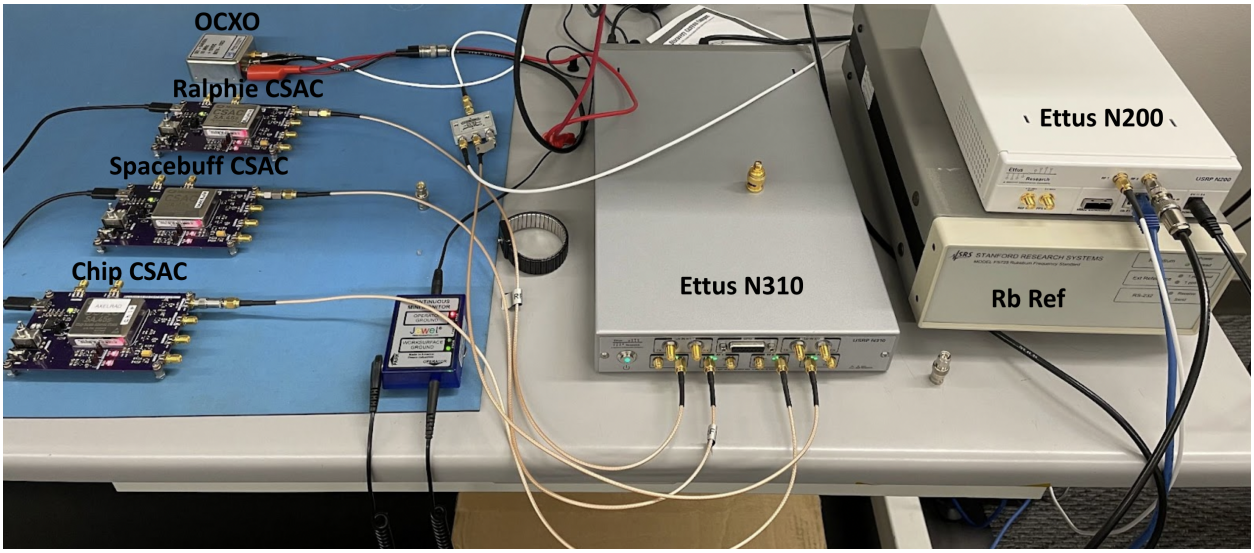


Figure 70. Clock Ensemble Testbed

Integrated Testbed Results

Multiple tests were run with different OCXOs, various methods of voltage adjustements, and varying control rates to assess the optimal combination. The time series of the OCXO phase was measured against the Rb on the N200 and the Allan deviation was computed to assess the signal stability. The best results for each OCXO and the stability of the other clocks are shown in Figure 71.

The effect of OCXO steering is clear at averaging intervals longer than 100 seconds as the signal stability is much smaller than the corresponding free-running curve. The steered Wenzel OCXO (green) is the result of using the benchtop power supply for the closed loop control. In this configuration the programmable voltage resolution was 1mV and the minimum control rate was 2 seconds due to processing latency and noisiness in the power supply. As a result, the short term behavior of this steered signal is not great - for short averaging intervals ($\tau < 4$s) the steered signal retains some of the unsteered behavior. However, the continuous application of a frequency adjustement through the power supply results in a servo bump that overshoots the stability of all the CSACs. Over time the Wenzel OCXO approaches the IEM, but the transient behavior at $4s < \tau < 100s$ is poor.

The steered NEL OCXO (blue) has better free-running short term stability values than the Wenzel OCXO and uses the DAC for programmatic frequency adjustment. The voltage resolution of the DAC is 19 microvolts, a resolution improvement of two orders of magnitude over the power supply. Additionally, the latency of the system and settling time of the DAC is small enough that a 1 second control rate can be used. These improvements result in an improved steered signal behavior as seen in Figure 71. The steered NEL OCXO curve more closely follows the free-running NEL OCXO behavior and the effect of steering with the DAC yields a much smaller servo bump than with the power supply. At longer intervals the stability of the steered signal approaches the IEM.

Both OCXOs are steered towards the IEM of the clock ensemble - thus the long term stability of both steered curves should have a better stability than any of the individual member clocks. As seen in Figure 71, the steered clocks have a better stability than the two nominally performing clocks, but almost the same stability as the best CSAC. This effect agrees with what was shown in simulation: in the case of a small clock ensemble where one member clock is significantly better performing than the others, the IEM of the ensemble will have approximately the same frequency stability as the best clock.

The difference between the stability of the two steered OCXOs in the long term is insignificant. If a certain mission application only requires long term stability, then the methods are approximately equal. However, if short term signal stability is important, then the method using a stable OCXO and high resolution DAC is clearly superior.

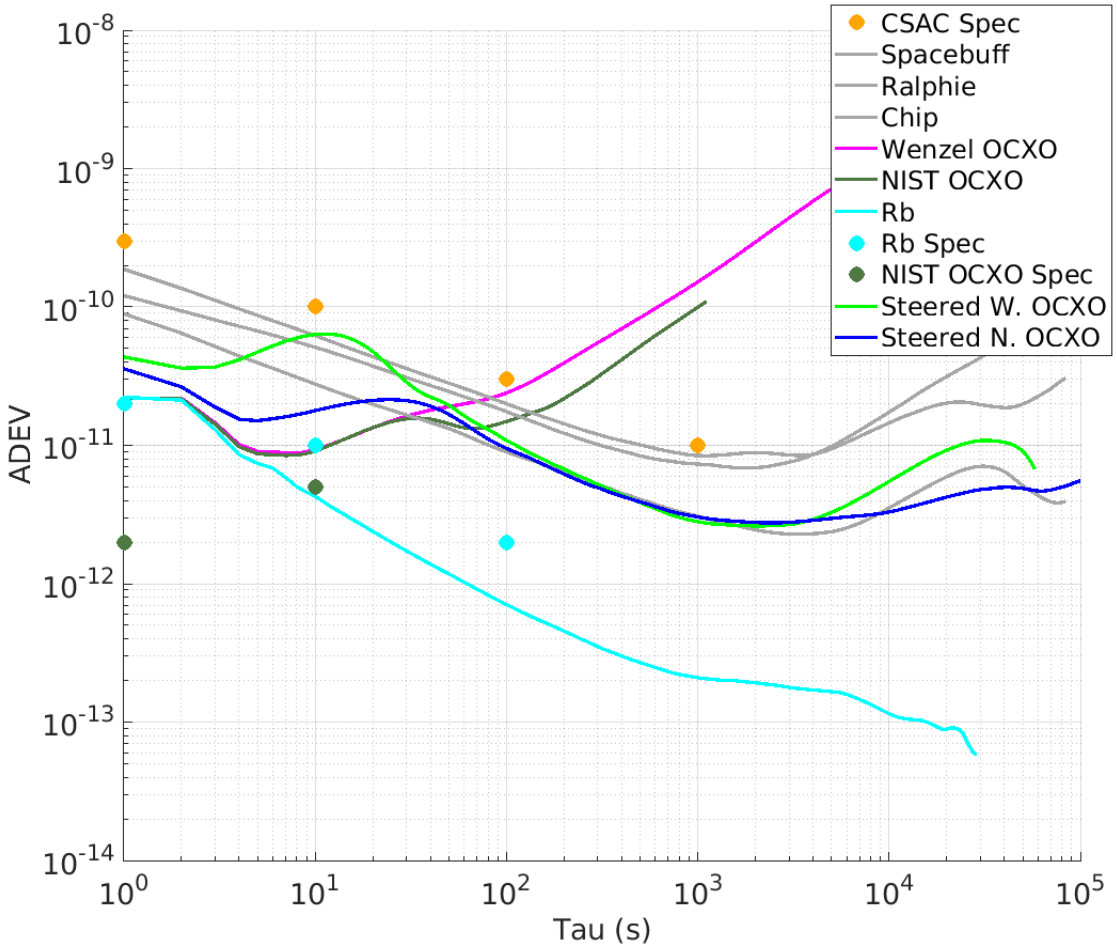


Figure 71. ADEV of Steered OCXOs

5 CONCLUSIONS

In this work we demonstrated the ability to characterize clocks using SDR metrology techniques, to programatically steer an OCXO, and form a small clock ensemble with the steered OCXO as the realization of the IEM. Our ability to accurately characterize clocks using SDRs depends on the stability of the oscillators under test, the stability of the reference clock, and the noise contribution of the measurement system. Here we showed that our best measurement system contributes noise on the picosecond scale at one second, which is much smaller than the short term stability of all our clocks. While the CSACs can readily be measured against the Rb with any of the measurement systems, our ability to measure the behavior of both OCXOs at short averaging intervals is limited by the stability of the rubidium frequency reference. In order to properly observe the behavior of the OCXOs we would need to procure a more stable reference oscillator or use alternative methods of clock characterization.

The noise contribution from the measurement system had a negligible effect on our ability to characterize oscillators, as it is smaller than the 1 second stability of all our clocks. If we were to use clocks with better stability or characterize multiple clocks simultaneously, the short term measurement noise would need to be considered.

The steering analysis showed the effect of the short term OCXO stability and the voltage source on the quality of the steered signal response. An OCXO with improved short term stability with respect to the target clock state will result in a better steered signal as the clock can be gently steered to rely on the improved free-running stability of the clock over the short averaging intervals. The source of the applied voltage will determine the smallest possible frequency adjustment and the smallest control rate. A smaller voltage resolution will result in a larger range of steering possibilities for a given voltage range. The settling time of the voltage source and the general latency of the OCXO steering system will determine the smallest control rate, with smaller control rates yielding better results. In our analysis we found that the NEL OCXO and DAC, posessing both of the aforementioned qualities, resulted in a more stable steered signal output than the Wenzel OCXO and benchtop power supply system.

In the case of this small clock ensemble where one oscillator is significantly more stable than the others, the IEM closely approximates the behavior of that oscillator. For the purpose of generating a clock signal better than the ensemble members, in this scenario it seems logical to just steer the OCXO to the best clock and not bother with ensembling. However, this ignores the resiliency benefits of using multiple clocks. With only a single clock there is a single point of failure in the timing system where errors in the CSAC will manifest in the steered output signal; when using multiple clocks, the effect of an error in any single clock on the steered output signal can potentially be mitigated by using a clock ensemble.

Future work will focus on the stability of the steered output signal in the presence of phase and frequency errors in the member clocks. The effect of clock errors on the steered OCXO signal both with and without detection / mitigation systems are of interest. Initial analysis will be conducted using CSAC phase measurements and artificially adding errors. The perturbed data sets will then be processed in simulation to understand the expected system response both with and without fault detection and mitigation systems. Additionally, we plan on expanding the clock ensemble using additional CSACS and SDRs. The expansion should yield a steered OCXO signal with noticeably improved stability for long averaging intervals.

REFERENCES

[1] "Strategy for the Department of Defense Positioning, Navigation, and Timing (PNT) Enterprise," Tech. Rep., 2019.

[2] P. H. Dana and B. Penrod, "The Role of GPS in Precise Time and Frequency Dissemination," *GPS World*, pp. 38–43, 1990.

[3] L. Mohon, *Deep Space Atomic Clock (DSAC) Overview*, Jul. 2020, [Online], available: https://www.nasa.gov/mission_pages/tdm/clock/overview.html.

[4] M. M. Rybak, P. Axelrad, J. Seubert, and T. Ely, "Chip Scale Atomic Clock–Driven One-Way Radiometric Tracking for Low-Earth-Orbit CubeSat Navigation," *Journal of Spacecraft and Rockets*, vol. 58, no. 1, pp. 1–10, 2020, DOI: 10.2514/1.A34684.

[5] C. Flood, M. Q. LaBarge, L. Schement, H. Dixon, and P. Axelrad, "A Testbed for Low-SWaP Atomic Clock Ensemble Development," *Proceedings of the 52nd Annual Precise Time and Time Interval Systems and Applications Meeting*, pp. 287–300, Jan. 2021, DOI: 10.33012/2021.17790.

[6] C. Flood, W. Watkins, and P. Axelrad, "Signal Generation in a Low-SWaP Atomic Clock Ensemble," *Proceedings of the 53nd Annual Precise Time and Time Interval Systems and Applications Meeting*, pp. 45–57, Jan. 2022, DOI: 10.33012/2022.18272.

[7] J. Sherman and R. Jordens, "Oscillator metrology with software defined radio," *Review of Scientific Instruments*, vol. 87, no. 054711, pp. 1–11, 2016.

[8] Marion Gödel and Johann Furthner, "Robust Ensemble Time Onboard a Satellite," *Proceedings of the 48th Annual Precise Time and Time Interval Systems and Applica-tions Meeting*, pp. 26–43, Jan. 2017, DOI: 10.33012/2017.15007.

[9] K. R. Brown, "The Theory of the GPS Composite Clock," *Proceedings of the 4th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 1991)*, pp. 223–241, Sep. 1991.

[10] M. Gödel, T. D. Schmidt, and J. Furthner, "Comparison between simulation and hardware realization for different clock steering techniques," *Metrologia*, vol. 56, no. 3, p. 035 001, May 2019, DOI: 10.1088/1681-7575/ab144d. [Online], available: https: //doi.org/10.1088/1681-7575/ab144d .

[11] *SA.45s CSAC and RoHS CSAC Options 001 and 003*, 090-02984-001(003), 900-00744-000 Rev E, Microsemi, Mar. 2019.

[12] *FS725 — Benchtop rubidium frequency standard*, FS725, Stanford Research Systems, Jul. 2022.

[13] *Premium 10 MHz-SC Streamline Crystal Oscillator*, 501-04609, Wenzel Associates, Jul. 2022.

[14] *Precision ultra low phase noise multi frequency ocxo reference module*, 1326A, NEL Frequency Controls, Inc.

[15] *Miniature, ultra-portable high precision performance atomic frequency source*, mRO-50, Orolia.

[16] *Chip-Scale Atomic Clock (CSAC) SA.45s User's Guide*, DS50003041A, Microsemi, 2020.

[17] "Red pitaya," [Online], available: `https://redpitaya.com/`.

[18] *N200/n210*, N200, https://kb.ettus.com/N200/N210, Ettus Research, Dec. 2020.

[19] *N300/n310*, N310, https://kb.ettus.com/N300/N310, Ettus Research, Dec. 2020.

[20] C. Zucca and P. Tavella, "The clock model and its relationship with the allan and related variances," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 52, no. 2, pp. 289–296, 2005, DOI: `10.1109/TUFFC.2005.1406554`.

[21] W. L. Brogan, "Modern control theory," in Prentice-Hall, 1991, pp. 448–457.

[22] *Gnu radio wiki*, https://wiki.gnuradio.org/index.php/Main*Page*, GNU Radio, Jun. 2022.

[23] B. Bloessl, M. Müller, and M. Hollick, "Benchmarking and profiling the gnuradio scheduler," *Proceedings of the GNU Radio Conference*, vol. 4, no. 1, 2019, [Online], available: `https://pubs.gnuradio.org/index.php/grcon/article/view/64`.

[24] J. Sherman *et al.*, *A resilient architecture for the realization and distribution of coor-dinated universal time to critical infrastructure systems in the united states: Method-ologies and recommendations from the national institute of standards and technology (nist)*, en, 2021-11-03 04:11:00 2021, DOI: `https://doi.org/10.6028/NIST.TN.2187`,[Online], available: `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=933488`.

[25] D. L. Mills, "Internet Time Synchronization: the Network Time Protocol," *Network Time Synchronization*, pp. 10–11, Oct. 1998.

[26] S. T. Watt, S. Achanta, H. Abubakari, and E. Sagen, "Understanding and Applying Precision Time Protocol," *4th Annual Clemson University Power Systems Conference, March 2015*, pp. 1–7, Dec. 2015.

[27] *IEEE 1588 Precise Time Protocol: The New Standard in Time Synchronization*, MSCC-0104-WP-01007-1.00-1117, Microsemi, Mar. 2017.

[28] T. Włostowski, "Precise time and frequency transfer in a White Rabbit network," Tech. Rep., May 2011.

[29] J. Savory, J. Sherman, and S. Romisch, "White Rabbit-Based Time Distribution at NIST," Tech. Rep., 2018.

[30] J. Serrano *et al.*, "The White Rabbit Project," Tech. Rep., 2009.

APPENDIX A - Red Pitaya Data Acquisition MATLAB Code

```
%% Define Red Pitaya as TCP/IP object
clear all
close all
clc
IP = '192.168.178.111';                 % Input IP of your Red Pitaya...
port = 5000;
RP = tcpclient(IP, port);               % Define Red Pitaya as TCP/IP object (connection to
% RP.InputBufferSize = 16384*32;        % Buffer sizes are calculated automatically with th

%% Open connection with your Red Pitaya

RP.ByteOrder = "big-endian";
configureTerminator(RP, 'CR/LF');

flush(RP, "input");
flush(RP, "output");

% Set decimation vale (sampling rate) in respect to you
% acquired signal frequency

writeline(RP,'ACQ:RST');
writeline(RP,'ACQ:DEC 1');
writeline(RP,'ACQ:TRIG:LEV 0');

% there is an option to select coupling when using SIGNALlab 250-12
% writeline(RP,'ACQ:SOUR1:COUP AC');    % enables AC coupling on channel 1

% by default LOW level gain is selected
% writeline(RP,'ACQ:SOUR1:GAIN LV');    % user can switch gain using this command


% Set trigger delay to 0 samples
% 0 samples delay set trigger to the center of the buffer
% Signal on your graph will have trigger in the center (symmetrical)
% Samples from left to the center are samples before trigger
% Samples from center to the right are samples after trigger

writeline(RP,'ACQ:TRIG:DLY 0');

%% Start & Trigg
% Trigger source setting must be after ACQ:START
```

```
% Set trigger to source 1 positive edge

writeline(RP,'ACQ:START');

% After acquisition is started some time delay is needed in order to acquire fresh samples
% Here we have used time delay of one second, but you can calculate the exact value by taki
% length and sampling rate

writeline(RP,'ACQ:TRIG CH1_PE');

% Wait for trigger
% Until trigger is true wait with acquiring
% Be aware of the while loop if trigger is not achieved
% Ctrl+C will stop code executing in MATLAB

while 1
        trig_rsp = writeread(RP,'ACQ:TRIG:STAT?')

        if strcmp('TD', trig_rsp(1:2))      % Read only TD

            break

        end
    end


% Read data from buffer
signal_str   = writeread(RP,'ACQ:SOUR1:DATA?');
signal_str_2 = writeread(RP,'ACQ:SOUR2:DATA?');

% Convert values to numbers.
% First character in the string is    {
% and the last 3 are 2 empty spaces and a    }    .

signal_num   = str2num(signal_str  (1, 2:length(signal_str)-3));
signal_num_2 = str2num(signal_str_2(1, 2:length(signal_str_2)-3));

plot(signal_num)
hold on
plot(signal_num_2,'r')
grid on
ylabel('Voltage / V')
xlabel('samples')

clear RP;
```

APPENDIX B - Testbed Simulation MATLAB Code

```
%{
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Margaret Rybak, Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022
%}

%% Generate Clock Signals
msrmtNoiseScenarios = (1E-12);

%Set-Up timing Vectors
dT = 1;
maxTime = 10000;
duration = maxTime*3;
timeVect = 0:dT:duration;
numSteps = length(timeVect);

run('Generate_Clock_Signals.m');

%% Plot True Clocks
figure;
hold on;
set(gcf, 'color', 'w');
plot(timeVect ./ 86400, trueClock(1, :), 'r');
plot(timeVect ./ 86400, trueClock(3, :), 'm');
plot(timeVect ./ 86400, trueClock(5, :), 'b');
plot(timeVect ./ 86400, trueClock(7, :), 'g');
xlabel('Time (Days)', 'FontSize', 20);
ylabel('Clock Bias (s)', 'FontSize', 20);
grid on;
set(gca, 'FontSize', 20);

%% Calculate Allan Deviations
maxTau = maxTime;
minTau = dT;
numADEV_Points = 1000;

trueClockOAD = NaN(4, numADEV_Points);
[trueClockOAD(1,:), Tau] = OAD_maxminTau(trueClock(1, :),numADEV_Points,minTau, maxTau);
[trueClockOAD(2,:), ~] = OAD_maxminTau(trueClock(3, :),numADEV_Points,minTau, maxTau);
[trueClockOAD(3,:), ~] = OAD_maxminTau(trueClock(5, :),numADEV_Points,minTau, maxTau);
[trueClockOAD(4,:), ~] = OAD_maxminTau(trueClock(7, :),numADEV_Points,minTau, maxTau);
```

```
%% ADEV plots
figure;
set(gcf, 'color', 'w');
hold on;
%these are the signals we want on every graph
csac1 = loglog(Tau, trueClockOAD(1,:), 'r', 'LineWidth', 2.5);
csac2 = loglog(Tau, trueClockOAD(3,:), 'm', 'LineWidth', 2.5);
csac3 = loglog(Tau, trueClockOAD(2,:), 'b', 'LineWidth', 2.5);
ocxo = loglog(Tau, trueClockOAD(4,:), 'g', 'LineWidth', 2.5);

set(gca, 'YScale', 'log', 'FontSize', 20);
set(gca, 'XScale', 'log', 'FontSize', 20);

legend([csac1, csac2, csac3, ocxo], {'CSAC 1', 'CSAC 2', 'CSAC 3', 'OCXO'}, 'FontSize', 16)

xlabel('Tau (s)', 'FontSize', 20)
ylabel('ADEV', 'FontSize', 20)
grid on

%% Combinations
%timeArray contains values for the control frequency
%timeArray = [1; 5; 10; 20; 50; 100];
timeArray = [1];

%generate the pole placement gain array
allEigenvalues = [.95, .949; .9, .901; .85, .849; .8, .801; .75, .749; .7, .701; .65, .649
%desiredEigenvalues = desiredEigenvalues(1:10, :);
%desiredEigenvalues = [.05, .049];
desiredEigenvalues = allEigenvalues([1, 19], :);

%A is the dynamics for the clocks, B is the control matrix
A = [1, dT; 0, 1];
B = [0; 1];
Garray = zeros(size(desiredEigenvalues, 1), 2);

for i = 1:size(desiredEigenvalues, 1)
    curEigenvals = desiredEigenvalues(i, :)';
    tempG = place(A, B, curEigenvals);
    Garray(i, :) = tempG;
end

%this will generate the combinations of control frequencies and gain values
gInds = 1:1:size(Garray, 1);
[m,n] = ndgrid(timeArray,gInds);
Z = [m(:),n(:)];

%preallocate information we want to store
```

```
secondKFOADHist = zeros(length(Z), numADEV_Points);
clockFourOADHist = zeros(length(Z), numADEV_Points);
iemOADHist = zeros(length(Z), numADEV_Points);

%iterate through all of the time and gain combinations
for i = 1:size(Z, 1)
    %get the current values of time and G
    controlInputCadence = Z(i, 1);
    curGInd = Z(i, 2);
    curGValues = Garray(curGInd, :);

    %call the sim function
    [truthClockHistory,secondKFEstimates, curIEM] = SimulateClockTestbed(numClocks,numClkSt
                            msrmtNoise,controlInputCadence,curGValues,STM,sqrtQ,Q,trueClock

    %compute ADEV
    [secondKFOAD, Tau] = OAD_maxminTau(secondKFEstimates(1, :),numADEV_Points,minTau, maxTa
    [clockFourOAD, ~] = OAD_maxminTau(truthClockHistory(7, :),numADEV_Points,minTau, maxTau
    [curIEMOAD, ~] = OAD_maxminTau(curIEM(1, :),numADEV_Points,minTau, maxTau);
    secondKFOADHist(i, :) = secondKFOAD;
    clockFourOADHist(i, :) = clockFourOAD;
    iemOADHist(i, :) = curIEMOAD;
end

%%Plot OAD: Round 1
for i = 1:length(timeArray)
    curTime = timeArray(i);
    figure;
    set(gcf, 'color', 'w');
    hold on;
    %these are the signals we want on every graph
    trueCSAC1 = loglog(Tau, trueClockOAD(1,:), 'r', 'LineWidth', 2.5);
    trueCSAC2 = loglog(Tau, trueClockOAD(2,:), 'm', 'LineWidth', 2.5);
    trueCSAC3 = loglog(Tau, trueClockOAD(3,:), 'b', 'LineWidth', 2.5);
    ocxo = loglog(Tau, trueClockOAD(4,:), 'g', 'LineWidth', 2.5);
    iem = loglog(Tau, iemOADHist(i,:), 'k', 'LineWidth', 2.5);

    loglog(1, CSAC_Tau_1, 'o', 'Color',[255/255,165/255,0], 'MarkerFaceColor', [255/255,165
    loglog(10, CSAC_Tau_10, 'o', 'Color',[255/255,165/255,0], 'MarkerFaceColor', [255/255,1
    loglog(100, CSAC_Tau_100, 'o', 'Color',[255/255,165/255,0], 'MarkerFaceColor', [255/255
    loglog(1000, CSAC_Tau_1000, 'o', 'Color',[255/255,165/255,0], 'MarkerFaceColor', [255/2
    loglog(10000, CSAC_Tau_10000, 'o', 'Color',[255/255,165/255,0], 'MarkerFaceColor', [255
    csacSpec = loglog(100000, CSAC_Tau_100000, 'o', 'Color',[255/255,165/255,0], 'MarkerFac
    set(gca, 'YScale', 'log', 'FontSize', 16);
    set(gca, 'XScale', 'log', 'FontSize', 16);

    %find array inds corresponding to current control input cadence
```

```
    relevantArrayInds = find(Z(:, 1) == curTime);
    for j = 1:length(relevantArrayInds)
        curComboInd = relevantArrayInds(j);
        loglog(Tau, clockFourOADHist(curComboInd,:), 'LineWidth', 2.5);
    end

    legend([trueCSAC1, trueCSAC2, trueCSAC3, ocxo, iem, csacSpec], {'CSAC1', 'CSAC2', 'CSA
    title(['OADEV for Simulated Clocks: t_{control} = ' num2str(curTime) ' second'], 'Font
    xlabel('Tau (s)', 'FontSize', 16)
    ylabel('ADEV', 'FontSize', 16)
    grid on
    set(gca, 'FontSize', 20);
end
```

APPENDIX C - Clock Signal Generation MATLAB Code

```
%{
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Margaret Rybak, Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022
%}

%This is where true clocks are simulated
msrmtNoise = msrmtNoiseScenarios(1);
savePlots = true;

%CSAC ADEV values from Microsemi
CSAC_Tau_1 =  3E-10;
CSAC_Tau_10 = 1E-10;
CSAC_Tau_100 = 3E-11;
CSAC_Tau_1000 = 1E-11;
CSAC_Tau_10000 = 3E-12;
CSAC_Tau_100000 = 1E-11;

%Define CSAC sigma white noise and random walk parameters based on the ADEV
%IEEE Std. 647-2006 IEEE Standard Specification Format Guide and Test
%Procedure for Single-Axis Laser Gyros
y = log10(CSAC_Tau_10000);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = CSAC_Tau_1; sigrwf = K;
var_WHFreq_CSAC =  sigwhf^2; var_RWFreq_CSAC =  sigrwf^2;

%Parameters for OCXO
y = log10(1e-9);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 1e-11; sigrwf = K;
var_WHFreq_ocxo =  sigwhf^2; var_RWFreq_ocxo =  sigrwf^2;

%Parameters for N200 with ext ref
y = log10(6e-11);
```

```
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 5.7e-9; sigrwf = K;
var_WHFreq_N200 =  sigwhf^2; var_RWFreq_N200 =  sigrwf^2;

%Parameters for DARPA ACES
y = log10(3e-15);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 3e-13; sigrwf = K;
var_WHFreq_ACES =  sigwhf^2; var_RWFreq_ACES =  sigrwf^2;

%Parameters for RAFS
y = log10(1e-14);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 1e-12; sigrwf = K;
var_WHFreq_RAFS =  sigwhf^2; var_RWFreq_RAFS =  sigrwf^2;

%Parameters for ORAFS
y = log10(1e-15);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 3e-13; sigrwf = K;
var_WHFreq_ORAFS =  sigwhf^2; var_RWFreq_ORAFS =  sigrwf^2;

%Parameters for H Maser
y = log10(1e-15);
x = log10(10000);
m = .5;
b =y-m*x;
logK = m*log10(sqrt(3)) + b;
K = 10^logK;
sigwhf = 6e-14; sigrwf = K;
var_WHFreq_Maser =  sigwhf^2; var_RWFreq_Maser =  sigrwf^2;
```

```
%Number of clocks and number of clock states to use
numClocks = 4;
numClkSt = 2;

%Stack the variance parameters (using repmat since all clocks are
%CSAC in this case), can add more clock states for clock
%signatures that have them, only using 2-state for CSAC (vars(nc,3) = 0)
%currently only coded to accept a 3rd clock state (see loop below)
vars = repmat([var_WHFreq_CSAC, var_RWFreq_CSAC, 0], numClocks, 1);
vars(4, :) = [var_WHFreq_ocxo, var_RWFreq_ocxo, 0];

%The clocks signals will be stacked in groups of 2 (phase & frequency)
%to select these I create the phase index for each clock - I'm sure there
%is a better way to do this
index = (1:numClocks)*numClkSt -(numClkSt -1);

%Loop through and make a stacked matrix for each clocks STM & Covariance
%Matrix (useful if using clocks with different properties), currently only
%coded up to accept a third clock state, would have to add other as
%necessary Zucca, C., and P. Tavella, The Clock Model and Its Relationship
%with Allan and Related Variances
for nc = 1:numClocks
    run('STM_Covariance_Matrices.m');
end

%Pull only the indexes for the 2-state scenario for CSAC
sqrtQ = sqrtQ(1:numClkSt*numClocks,1:numClkSt*numClocks);
STM = STM(1:numClkSt*numClocks,1:numClkSt*numClocks);
Cov = Cov(1:numClkSt*numClocks,1:numClkSt*numClocks);
Q = Q(1:numClkSt*numClocks,1:numClkSt*numClocks);

%% Create Clock Signatures

trueClock = zeros(numClocks*numClkSt, numSteps);

%Either load the saved or generate a new random vector
randVect = randn(numClocks*numClkSt,numSteps);

%Propagate the clock states
for ct = 2:numSteps
    trueClock(:,ct) = STM*trueClock(:,ct-1) + sqrtQ*randVect(:, ct-1);
end


%% CLOCK MEASUREMENTS
%Create the clock differenced measurements, uses the first clock as
```

```
%the reference
diffClocks = NaN(numClocks-1, numSteps);
noiseyDiffClocks  = NaN(numClocks-1, numSteps);
for nc = 2:numClocks
    diffClocks(nc-1,:) = trueClock(index(nc),:) - trueClock(1,:);
    noiseyDiffClocks(nc-1,:) = trueClock(index(nc),:) - trueClock(1,:) + msrmtNoise*randn(1
end
```

APPENDIX D - Clock STM Generation MATLAB Code

```
%{
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Margaret Rybak, Penina Axelrad
Last Edited:  28 June 2022
%}

%Loop through and make a stacked matrix for each clocks STM &
%Covariance Matrix (useful if using clocks with different
%properties), currently only coded up to accept a third clock
%state, would have to add other as necessary
%Zucca, C., and P. Tavella,  The  Clock Model and Its Relationship with Allan and Related V
vars123 = [vars(nc,1), vars(nc,2), vars(nc,3)];
STMi = [1, dT, (dT^2)/2; 0, 1, dT; 0, 0, 1];
q11 = vars(nc,1)*dT + (vars(nc,2)*(dT^3))/3 + (vars(nc,3)*(dT^5))/20;
q21 =  (vars(nc,2)*(dT^2))/2 + (vars(nc,3)*(dT^4))/8;
q31 = (vars(nc,3)*(dT^3))/6;
q12 = q21;
q22 = vars(nc,2)*dT + (vars(nc,3)*(dT^3))/3;
q32 = (vars(nc,3)*(dT^2))/2;
q13 = q31;
q23 = q32;
q33 = vars(nc,3)*dT;
Qi = [q11 q12 q13; q21 q22, q23; q31 q32, q33];
Q(index(nc):index(nc)+(numClkSt-1), index(nc):index(nc)+(numClkSt-1)) = Qi(1:numClkSt,1:nu
STM(index(nc):index(nc)+(numClkSt-1), index(nc):index(nc)+(numClkSt-1)) = STMi(1:numClkSt,1
Cov(index(nc):index(nc)+(numClkSt-1), index(nc):index(nc)+(numClkSt-1)) = diag(vars123(1:nu

%Singular value decomposition to convert covariance matrix to
%generate the correlated clock signature in the following loop
[U,S,Vh] = svd(Qi(1:numClkSt,1:numClkSt));
sqrtQ(index(nc):index(nc)+(numClkSt-1), index(nc):index(nc)+(numClkSt-1)) = U*sqrt(S)*Vh;
```

APPENDIX E - Allan Deviation MATLAB Code

```
%{
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Margaret Rybak, Penina Axelrad
Last Edited:  28 June 2022
%}

%NIST Handbook of Freq Stability Analysis, Riley (2008)
%5.2.4: Overlapping Allan Deviation (PG 16, EQ 11)
function [OAD, Tau] = OAD_maxminTau(signal,numPoints, minTau, maxTau)
%N is the number of data points in the input clock signal, this should be more
%than the maxTau (at least 4X) to prevent squirrely results at end of plot
N = length(signal);
%The averaging factor is multiplied by the minTau (from 1 to
%MaxTau/MinTau), this is equivalent to making a Tau Vector from MinTau to
%MaxTau spaced by MinTau
% maxAvgFactor = maxTau/minTau;
%Preallocate OAD and Tau Vectors
Tau = minTau:minTau:maxTau;
Tau = Tau(round(logspace(log10(1),log10(length(Tau)),numPoints)));
numTaus = length(Tau);
OAD = zeros(numTaus,1);
EB = zeros(numTaus,1);
%Run Loop through Averaging Factors from 1 to MaxAvgFactor
for nT = 1:numTaus
    %Define Tau as AveragingFactor (af) x MinTau
    m = Tau(nT)/minTau;
    %Preallocate Summation Loop as Number of Data Points - 2*AvgFactor
    %(Ensures window does not run out of data)
    xn = zeros(N-2*m,1);
    for ii = 1:N-2*m
        xn(ii) = (signal(ii+2*m) - 2*signal(ii+m) + signal(ii))^2
;
    end
    AD = sqrt((1/(2*(Tau(nT)^2)*(N-2*m)))*sum(xn,'omitnan'));
    OAD(nT) = AD;
end

end
```

APPENDIX F - Simulate Testbed Function MATLAB Code

```
%{
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Margaret Rybak, Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

%}

function [truthClockHistory,secondKFEstimates, IEM] = SimulateClockTestbed(numClocks,numCl
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

    %Preallocate Matricies
    numFiltStates = (numClocks-1)*numClkSt;
    clockStateEstimates = nan(numFiltStates,numSteps);
    secondKFEstimates = nan(2, numSteps);
    truthClockHistory = nan(8,numSteps);

    %Set the initial estimate at zeros
    xhat = zeros(6, 1);
    xhat2 = zeros(2, 1);
    secondKFEstimates(:, 1) = xhat2;

    %Size the intial covariance (I used the measurement noise in this case)
    P0 = diag(repmat([msrmtNoise^2, (msrmtNoise^2)/100], 1,numClocks-1));

    %Create the R matrix based on measurement noise
    R = diag((msrmtNoise^2)*ones(numClocks-2,1));

    %Generate H matrix  & initialize the implicit ensemble mean vector
    %based on Brown 1991 "The Theory of the GPS Composite Clock"
    %H = zeros(numClocks-1,numFiltStates);
    %H(:,1) = -1*ones(numClocks-1,1);
    H = [-1, 0, 1, 0, 0, 0;
        -1, 0, 0, 0, 1, 0];

    %set up the implicit ensemble mean
    IEM = nan(numClkSt,numSteps);
    IEM(:, 1) = zeros(2, 1);

    %initialize
```

```
prevClockStates = zeros(numClocks*numClkSt, 1);
truthClockHistory(:, 1) = prevClockStates;

firstH = [-1, 0, 1, 0, 0, 0, 0, 0;
    -1, 0, 0, 0, 1, 0, 0, 0];
%secondH = [-1, 0, 0, 0, 0, 0, 1, 0];

%dT = 1;
B = [0; 0; 0; 0; 0; 0; 0; 1];

controlInput = -G * xhat2;
I = eye(numFiltStates);

for kf = 1:numSteps-1

    %Clock State Evolution
    randomNums = randn(numClocks*numClkSt,1);
    truthClockStates = (STM*prevClockStates) + (B*controlInput) + (sqrtQ*randomNums);
    prevClockStates = truthClockStates;
    truthClockHistory(:, kf+1) = prevClockStates;

    %Make Measurements
    y = firstH*truthClockStates + msrmtNoise*randn(2,1);

    %First Kalman Filter
    %Propagate State Using Diff Eqns
    xbar = STM(1:6, 1:6)*xhat;

    %Calculate STM with Current State for Covariance Time Propagation
    Pbar = STM(1:6, 1:6)*P0(1:6, 1:6)*STM(1:6, 1:6)' + Q(1:6, 1:6); %STM*Cov*STM';

    %Measurement Update
    K = Pbar*H'*inv(H*Pbar*H' + R(1:2, 1:2));
    P0 = (I - K*H)*Pbar*(I-K*H)'+K*R*K';
    xhat = xbar + K*(y - H*xbar);

    %Brown covariance reduction
    Hstar = repmat(eye(2),numClocks-1,1);
    P0 = P0 - Hstar*inv(Hstar'*inv(P0)*Hstar)*Hstar';

    %store the estimate of the clock states
    clockStateEstimates(:,kf) = xhat;

    %Create the Implicit Ensemble Mean - can do this since it's all sim
    Omega = trueClock(1:6,kf+1) - xhat;
    W_IE = (Hstar'*(P0\Hstar))\Hstar'/(P0);
    IEM(:,kf+1) = W_IE*Omega;
```

```
        %compute the difference between the OCXO state and the IEM
        clockDiff = truthClockStates(7:8) - IEM(:, kf+1);

        %compute control input for clock we are steering
        %only apply control input every XX seconds
        if mod(kf, controlInputCadence) == 0
            %controlInput = -G * xhat2;
            controlInput = -G * clockDiff;
        else
            controlInput = 0;
        end
    end
end
```

APPENDIX G - Phasor Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr
import time
import math

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, sampleRate=4000.0, beatFreq=73.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Phasor Block',   # will show up in GRC
            in_sig=[np.float32,np.float32],
            out_sig=[np.float32, np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.beatFreq = beatFreq
        self.sampRate = sampleRate
        self.timeStep = 1 / sampleRate
        self.counter = 0
        self.two_pi = 2.0 * math.pi


    def work(self, input_items, output_items):

        #iterate over the length of the input
        for i in range(0, len(input_items[0])):
            measCosVal = input_items[0][i]
```

```
        measSinVal = input_items[1][i]

        #increment the running counter
        self.counter = self.counter + 1

        #compute nstar
        nStar = self.counter % self.sampRate

        #compute phase growth
        phaseGrowth = self.two_pi * self.beatFreq * nStar * self.timeStep

        #compute cosine and sine based on phase growth
        computedCosVal = math.cos(phaseGrowth)
        computedSinVal = math.sin(phaseGrowth)

        realVal = (measCosVal*computedCosVal) + (measSinVal*computedSinVal)
        imagVal = (measSinVal*computedCosVal) - (measCosVal*computedSinVal)

        #populate the first output with the first input
        output_items[0][i] = realVal
        output_items[1][i] = imagVal

    return len(output_items[0])
```

APPENDIX H - Phase Unwrap Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr
import math

class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Phase Unwrap Block',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.two_pi = 2.0 * math.pi
        self.wrapCounter = 0.0
        self.prevValue = 0.0
        self.firstCallBool = 1

    def work(self, input_items, output_items):
        if self.firstCallBool:
            self.prevValue = input_items[0][0]
            self.firstCallBool = 0

        for i in range(0, len(input_items[0])):
            #compute the difference between the current value and the previous value
            curPhaseVal = input_items[0][i]
```

```
        phaseDelta = curPhaseVal - self.prevValue

        if phaseDelta > math.pi:
            #shift the next values down accordingly
            numSteps = round(phaseDelta / self.two_pi)
            unwrappedPhase = curPhaseVal - (numSteps * self.two_pi)
        elif phaseDelta < (-1 * math.pi):
            #shift the next values up accordingly
            numSteps = round(phaseDelta / self.two_pi)
            unwrappedPhase = curPhaseVal + (numSteps * self.two_pi)
        else:
            #no unwrapping needed
            unwrappedPhase = curPhaseVal

        self.prevValue = unwrappedPhase

        #populate the output with the unwrapped phase
        output_items[0][i] = unwrappedPhase

    return len(output_items[0])
```

APPENDIX I - Ensemble Kalman Filter Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Kalman Filter Block:

This block is for use within GNU Radio Companion. This filter follows Godel's outline of
the Kalman Filter. It can take up to N clocks as input and either the 2 or 3 state clock
model.
"""

import numpy as np
import numpy.matlib
import math
from gnuradio import gr

"""Initialize the filter with the following variables"""
dT = .125
numclocks = 3 #number of clocks we're estimating
numclockstates = 2 #number of clock states (2 or 3)
totalStates = numclocks*numclockstates #for intializing the matrices/loops
numInputs = 2 #how many measurements are being input into the system


#Allan Deviation from CSAC documentation
SB_Tau_1 =  2E-10
SB_Tau_100000 = 1E-11

Ralphie_Tau_1 =  2E-10
Ralphie_Tau_100000 = 1E-11

Chip_Tau_1 =  9E-11
Chip_Tau_100000 = 3E-12

#This code below is from pulled from Margaret's CSAC simulation code - I still don't fully
#x and m are constant
x = math.log10(10000)
m = .5

#these values change - start with SB
ySB = math.log10(SB_Tau_100000)
```

```
bSB = ySB - m*x
logK_SB = m*math.log10(math.sqrt(3)) + bSB
K_SB = 10**logK_SB
sigwhf_SB = SB_Tau_1
sigrwf_SB = K_SB
q1_SB =  sigwhf_SB**2
q2_SB =  sigrwf_SB**2

#can just copy for now since Ralphie and SB behavior is near identical
q1_Ralphie = q1_SB
q2_Ralphie = q2_SB

#populate chip parameters
yChip = math.log10(Chip_Tau_100000)
bChip = yChip - m*x
logK_Chip = m*math.log10(math.sqrt(3)) + bChip
K_Chip = 10**logK_Chip
sigwhf_Chip = Chip_Tau_1
sigrwf_Chip = K_Chip
q1_Chip =  sigwhf_Chip**2
q2_Chip =  sigrwf_Chip**2

#Q is the process noise matrix for the clocks
Q = np.zeros([totalStates,totalStates])
Q[0][0] = q1_SB*dT+(q2_SB*(dT**3))/3
Q[0][1] = (q2_SB*(dT**2))/2
Q[1][0] = (q2_SB*(dT**2))/2
Q[1][1] = q2_SB*dT

Q[2][2] = q1_Ralphie*dT+(q2_Ralphie*(dT**3))/3
Q[2][3] = (q2_Ralphie*(dT**2))/2
Q[3][2] = (q2_Ralphie*(dT**2))/2
Q[3][3] = q2_Ralphie*dT

Q[4][4] = q1_Chip*dT+(q2_Chip*(dT**3))/3
Q[4][5] = (q2_Chip*(dT**2))/2
Q[5][4] = (q2_Chip*(dT**2))/2
Q[5][5] = q2_Chip*dT

#measurement matrix, our measurement is the difference between clock phase offsets
H = [[1,0,0,0,-1,0],[0,0,1,0,-1,0]]

#transpose of H for use within the filter
transH = np.transpose(H)
#identity matrix for use within the filter
ident = np.identity(totalStates)
```

```
#R is measurement noise in time units
R = 10**-20

Hs = np.matlib.repmat(np.identity(numclockstates),numclocks,1)
transHs = np.transpose(Hs)

initCovUncertainty = 10**-15
P = initCovUncertainty*np.identity(totalStates) #initialize error covariance matrix
pPrev = P

"""Loop through to created a stacked state transition matrix following Godel's model"""
STM = np.zeros([totalStates,totalStates])
xPrevHatMinus = np.zeros([totalStates, 1]);

for x in range(0,numclocks):
    #this is the code for the two state clock model
    STM[x*numclockstates][x*numclockstates] = 1.0
    STM[x*numclockstates][x*numclockstates+1] = dT

    STM[x*numclockstates+1][x*numclockstates] = 0
    STM[x*numclockstates+1][x*numclockstates+1] = 1.0

    #this is logic for the three state clock model
    if numclockstates==3.0:
        STM[x*numclockstates][x*numclockstates+2] = 0.5*dT*dT

        STM[x*numclockstates+1][x*numclockstates+2] = dT

        STM[x*numclockstates+2][x*numclockstates] = 0
        STM[x*numclockstates+2][x*numclockstates+1] = 0
        STM[x*numclockstates+2][x*numclockstates+2] = 1.0

transSTM = np.transpose(STM)

K = np.zeros([totalStates,1]) #intialize Kalman gain
transK = np.transpose(K) #transpose of Kalman gain for use within the filter

class blk(gr.sync_block):

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            # will show up in GRC
            name='Kalman Filter',

            #adding np.float32 more times changes the number of inputs
```

```
            in_sig=[np.float32,np.float32],

            #adding np.float32 more times changes the number of outputs
            out_sig=[np.float32,np.float32,np.float32,np.float32,np.float32,np.float32]

        )
        self.counter = 0
        self.seconds = 2
        self.messagerate = 8*self.seconds # want to send one message every X seconds


    def work(self, input_items, output_items):
        global xNextHatMinus
        global xPrevHatMinus
        global xNextHatPlus
        global pPrev
        global pPredict
        global pNext
        global STM
        global transSTM
        global K
        global transK

        if len(input_items[0]) == 1:
            remainder = self.counter % self.messagerate

            outputxhat = np.zeros((totalStates,1)) #to store xhat output for export out of

            z = np.zeros((numInputs, 1)) #create the measurement matrix for use within the

            for x in range (0,numInputs):
                z[x][0] = input_items[x][0] #add the input_items into the measurement matri

            """Process all measurements following Godel's model"""
            #state prediction: STM * xk
            xNextHatMinus = np.matmul(STM, xPrevHatMinus)

            #cov prediction: STM * P * STM' + Q
            pPredict = np.matmul(np.matmul(STM,pPrev),transSTM)+Q

            #Kalman Gain: K = P * H' * inv(H*P*H' + R)
            newMatCSACs = (np.linalg.inv(np.matmul(np.matmul(H,pPredict),transH)+R))

            K = np.matmul(np.matmul(pPredict,transH),newMatCSACs)
            transK = np.transpose(K)

            #state measurement update: xk+1 = xk + K*(y - H*xk)
```

```
        xNextHatPlus = xNextHatMinus + np.matmul(K, (z - np.matmul(H, xNextHatMinus)))

        #cov measurement update: (I - K*H)*P*(I - K*H)' + K*R*K'
        pMsmt = np.matmul(np.matmul((ident-np.matmul(K,H)),pPredict),(np.transpose(ider

        #this is a covariance reduction method used in the Godel paper, not normally us
        #P = P - Hs * inv(Hs' * inv(P) * Hs) * Hs'
        invPMsmt = np.linalg.inv(pMsmt)
        parenthQuant = np.matmul(np.matmul(transHs, invPMsmt), Hs)
        invParenthQuant = np.linalg.inv(parenthQuant)
        pMsmtReduced = pMsmt - (np.matmul(np.matmul(Hs,invParenthQuant), transHs))

        #update the previous state and covariance
        xPrevHatMinus = xNextHatPlus
        pPrev = pMsmtReduced

        self.counter = self.counter + 1

        for y in range (0,totalStates):
            output_items[y][0] = xNextHatPlus[y][0]

    elif len(input_items[0]) > 1:
        numSteps = len(input_items[0])

        #our output will be an array with all of the
        outputxhat = np.zeros((totalStates, numSteps))

        for i in range (0,numSteps):
            #initalize the measurement array
            z = np.zeros((numInputs, 1))

            #populate the current measurement vector with the appropriate indices from
            for x in range (0,numInputs):
                z[x][0] = input_items[x][i]
                if i == 0:
                    print("cur msmt: {}".format(z[x][i]))

            """Process all measurements following Godel's model"""

            #state prediction: STM * xk
            xNextHatMinus = np.matmul(STM, xPrevHatMinus)

            #cov prediction: STM * P * STM' + Q
            pPredict = np.matmul(np.matmul(STM,pPrev),transSTM)+Q

            #Kalman Gain
            #K = P * H' * inv(H*P*H' + R)
```

```
            K = np.matmul(np.matmul(pPredict,transH),(np.linalg.inv(np.matmul(np.matmul
            transK = np.transpose(K)

            #state measurement update: xk+1 = xk + K*(y - H*xk)
            xNextHatPlus = xNextHatMinus + np.matmul(K, (z - np.matmul(H, xNextHatMinus

            #cov measurement update: (I - K*H)*P*(I - K*H)' + K*R*K'
            pMsmt = np.matmul(np.matmul((ident-np.matmul(K,H)),pPredict),(np.transpose

            #this is a covariance reduction method used in the Godel paper, not normal
            #P = P - Hs * inv(Hs' * inv(P) * Hs) * Hs'
            pMsmtReduced = pMsmt - np.matmul(Hs,np.matmul(np.linalg.inv(np.matmul(trans

            #update the previous state and covariance
            xPrevHatMinus = xNextHatPlus
            pPrev = pMsmtReduced

            for y in range (0,totalStates):
                output_items[y][i] = xNextHatPlus[y][0]
                print(output_items[y][i])


    else:
        print("did not go through any logic")
        output_items[y][i] = xNextHatPlus[y][0]


    return len(output_items[0])
```

APPENDIX J - OCXO Kalman Filter Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Kalman Filter Block:

This block is for use within GNU Radio Companion. This filter follows Godel's outline of
the Kalman Filter. It can take up to N clocks as input and either the 2 or 3 state clock
model.
"""

import numpy as np
import numpy.matlib
import math
from gnuradio import gr

"""Initialize the filter with the following variables"""
dT = .125 #time step for data
numclocks = 1 #number of clocks we're estimating
numclockstates = 2 #number of clock states (2 or 3)
totalStates = numclocks*numclockstates #for intializing the matrices/loops
numInputs = 1 #how many measurements are being input into the system

ocxo_tau_1 = 2E-12
ocxo_tau_10000 = 1E-10

#This code below is from pulled from Margaret's CSAC simulation code - I still don't fully
y = math.log10(ocxo_tau_10000)
x = math.log10(10000)
m = .5
b = y-m*x
logK = m*math.log10(math.sqrt(3)) + b
K = 10**logK
sigwhf = ocxo_tau_1
sigrwf = K
q1 =  sigwhf**2
q2 =  sigrwf**2

#Q is the process noise matrix for the clocks
Q = np.zeros([totalStates,totalStates])
#This loop is populating the large process noise matrix with the CSAC process noise paramet
```

```
for x in range(0,numclocks):
    #First row
    Q[x*numclockstates][x*numclockstates] = q1*dT+(q2*(dT**3))/3
    Q[x*numclockstates][x*numclockstates+1] = (q2*(dT**2))/2
    #Second row
    Q[x*numclockstates+1][x*numclockstates] = (q2*(dT**2))/2
    Q[x*numclockstates+1][x*numclockstates+1] = q2*dT

#measurement matrix, our measurement is the ocxo phase
H = [[1,0]]

#transpose of H for use within the filter
transH = np.transpose(H)
#identity matrix for use within the filter
ident = np.identity(totalStates)

#R is measurement noise in time units
R = 10**-24

Hs = np.matlib.repmat(np.identity(numclockstates),numclocks,1)
transHs = np.transpose(Hs)

P = R*np.identity(totalStates) #initialize error covariance matrix
pPrev = P

"""Loop through to created a stacked state transition matrix following Godel's model"""
STM = np.zeros([totalStates,totalStates])
xPrevHatMinus = np.zeros([totalStates, 1]);

for x in range(0,numclocks):
    #this is the code for the two state clock model
    STM[x*numclockstates][x*numclockstates] = 1.0
    STM[x*numclockstates][x*numclockstates+1] = dT

    STM[x*numclockstates+1][x*numclockstates] = 0
    STM[x*numclockstates+1][x*numclockstates+1] = 1.0

    #this is logic for the three state clock model
    if numclockstates==3.0:
        STM[x*numclockstates][x*numclockstates+2] = 0.5*dT*dT

        STM[x*numclockstates+1][x*numclockstates+2] = dT

        STM[x*numclockstates+2][x*numclockstates] = 0
        STM[x*numclockstates+2][x*numclockstates+1] = 0
        STM[x*numclockstates+2][x*numclockstates+2] = 1.0
```

```
transSTM = np.transpose(STM)

K = np.zeros([totalStates,1]) #intialize Kalman gain
transK = np.transpose(K) #transpose of Kalman gain for use within the filter

class blk(gr.sync_block):

    def __init__(self):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            # will show up in GRC
            name='OCXO Kalman Filter',

            #adding np.float32 more times changes the number of inputs
            in_sig=[np.float32],

            #adding np.float32 more times changes the number of outputs
            out_sig=[np.float32,np.float32]

        )
        self.counter = 0
        self.seconds = 2
        self.messagerate = 8*self.seconds # want to send one message every X seconds

    def work(self, input_items, output_items):
        global xNextHatMinus
        global xPrevHatMinus
        global xNextHatPlus
        global pPrev
        global pPredict
        global pNext
        global STM
        global transSTM
        global K
        global transK

        if len(input_items[0]) == 1:
            remainder = self.counter % self.messagerate

            outputxhat = np.zeros((totalStates,1)) #to store xhat output for export out of

            z = np.zeros((numInputs, 1)) #create the measurement matrix for use within the

            for x in range (0,numInputs):
                z[x][0] = input_items[x][0] #add the input_items into the measurement matri
```

```
        """Process all measurements following Godel's model"""
        #state prediction: STM * xk
        xNextHatMinus = np.matmul(STM, xPrevHatMinus)

        #cov prediction: STM * P * STM' + Q
        pPredict = np.matmul(np.matmul(STM,pPrev),transSTM)+Q

        #Kalman Gain
        #K = P * H' * inv(H*P*H' + R)
        newMatOCXO = (np.linalg.inv(np.matmul(np.matmul(H,pPredict),transH)+R))
        K = np.matmul(np.matmul(pPredict,transH),newMatOCXO)
        transK = np.transpose(K)

        #state measurement update: xk+1 = xk + K*(y - H*xk)
        xNextHatPlus = xNextHatMinus + np.matmul(K, (z - np.matmul(H, xNextHatMinus)))

        #cov measurement update: (I - K*H)*P*(I - K*H)' + K*R*K'
        pMsmt = np.matmul(np.matmul((ident-np.matmul(K,H)),pPredict),(np.transpose(iden

        #update the previous state and covariance
        xPrevHatMinus = xNextHatPlus
        pPrev = pMsmt
        self.counter = self.counter + 1

        for y in range (0,totalStates):
            output_items[y][0] = xNextHatPlus[y][0]

    elif len(input_items[0]) > 1:
        print("In the multiple input section.")
        numSteps = len(input_items[0])

        #our output will be an array
        outputxhat = np.zeros((totalStates, numSteps))

        print("looping through all measurements")
        for i in range (0,numSteps):
            #initalize the measurement array
            z = np.zeros((numInputs, 1))

            #populate the current measurement vector with the appropriate indices from
            for x in range (0,numInputs):
                z[x][0] = input_items[x][i]
                if i == 0:
                    print("cur msmt: {}".format(z[x][i]))

            #state prediction: STM * xk
            xNextHatMinus = np.matmul(STM, xPrevHatMinus)
```

```
            #cov prediction: STM * P * STM' + Q
            pPredict = np.matmul(np.matmul(STM,pPrev),transSTM)+Q

            #Kalman Gain
            #K = P * H' * inv(H*P*H' + R)
            K = np.matmul(np.matmul(pPredict,transH),(np.linalg.inv(np.matmul(np.matmul
            transK = np.transpose(K)

            #state measurement update: xk+1 = xk + K*(y - H*xk)
            xNextHatPlus = xNextHatMinus + np.matmul(K, (z - np.matmul(H, xNextHatMinus

            #cov measurement update: (I - K*H)*P*(I - K*H)' + K*R*K'
            pMsmt = np.matmul(np.matmul((ident-np.matmul(K,H)),pPredict),(np.transpose

            #update the previous state and covariance
            xPrevHatMinus = xNextHatPlus
            pPrev = pMsmt

            for y in range (0,totalStates):
                output_items[y][i] = xNextHatPlus[y][0]
                print(output_items[y][i])


    else:
        print("did not go through any logic")
        output_items[y][i] = xNextHatPlus[y][0]


    #print('returning data')
    #print(len(output_items[0]))

    return len(output_items[0])
```

APPENDIX K - DAC Message Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
import pyvisa
import time
from gnuradio import gr
import serial

#set up initial communication with the device
ser = serial.Serial('/dev/ttyUSB0', 115200, timeout=1)
ser.write(b'-0.144\n')


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='DAC Message Block',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.example_param = example_param
        self.counter = 0
        self.seconds = 1
        self.messagerate = 8*self.seconds # want to send one message every X seconds
        self.vmin = -2.5
        self.vmax = 2.5
```

```
    #minimum voltage step size based on a 5V range and 18 bit DAC
    self.vstep = 0.000019
    #slope of the frequency response from the OCXO
    self.efcSlope = -2.18E-7 #frequency change per volt
    self.prevVoltage = -0.144
def work(self, input_items, output_items):

    #iterate over the length of the input
    for i in range(0, len(input_items[0])):
        #this remainder variable is how we send a message every second with an input ra
        remainder = self.counter % self.messagerate

        if remainder == 0:
            #send a command to the DAC
            print("sending command to DAC...")

            #convert the returned string to a float
            vFloat = self.prevVoltage

            #translate the command(the input) to a voltage adjustment
            steeringCommand = input_items[0][i]
            voltageAdjustment = steeringCommand / self.efcSlope

            #round the voltage adjustment to the nearest multiple of the step size
            quantizedVoltage = round(voltageAdjustment / self.vstep, 0) * self.vstep

            newVoltage = vFloat + quantizedVoltage
            newVoltage = round(newVoltage, 6)

            #make sure we are within the voltage min / max of the ocxo
            if newVoltage > self.vmax:
                newVoltage = self.vmax
            if newVoltage < self.vmin:
                newVoltage = self.vmin


            print("New voltage request is:")
            print(newVoltage)

            #create the voltage string
            voltString = str(newVoltage) + '\n'
            #send it to the DAC
            ser.write(voltString.encode('utf-8'))

            #reset the counter
            self.counter = 0
            self.prevVoltage = newVoltage
```

```
    self.counter = self.counter + 1

#output doesn't matter here
output_items[0][:] = input_items[0] * self.example_param
return len(output_items[0])
```

APPENDIX L - Power Supply Message Block

```
"""
Code developed by the Colorado Center for Astrodynamics Research (CCAR) at the University o

Authors: Christopher Flood, Penina Axelrad
Last Edited:  28 June 2022

Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
import pyvisa
import time
from gnuradio import gr

#initialization code to communicate with the power supply
rm = pyvisa.ResourceManager()
myInst = rm.open_resource('ASRL/dev/ttyS0::INSTR')
myInst.timeout = 10000
print(myInst.query('*IDN?'))
print('Channel 1 voltage is: ' + myInst.query('VOUT1?'))
print('Channel 2 voltage is: ' + myInst.query('VOUT2?'))


class blk(gr.sync_block):  # other base classes are basic_block, decim_block, interp_block
    """Embedded Python Block example - a simple multiply const"""

    def __init__(self, example_param=1.0):  # only default arguments here
        """arguments to this function show up as parameters in GRC"""
        gr.sync_block.__init__(
            self,
            name='Power Supply Message Block',   # will show up in GRC
            in_sig=[np.float32],
            out_sig=[np.float32]
        )
        # if an attribute with the same name as a parameter is found,
        # a callback is registered (properties work, too).
        self.example_param = example_param
        self.counter = 0
        self.seconds = 1
```

```
        self.messagerate = 8*self.seconds # want to send one message every X seconds
        self.vmin = 0.0
        self.vmax = 5.0
        self.vstep = .001
        self.efcSlope = -5E-8 #frequency change per volt

    def work(self, input_items, output_items):

        #iterate over the length of the input
        for i in range(0, len(input_items[0])):
            remainder = self.counter % self.messagerate

            if remainder == 0:
                #send a command to the power supply
                curV = myInst.query('VSET2?')

                print('Channel 2 voltage is set to: ' + curV)

                #convert the returned string to a float
                rmNewLine = curV.rstrip()
                rmV = rmNewLine.rstrip('V')
                vFloat = float(rmV)

                #translate the command(the input) to a voltage adjustment
                steeringCommand = input_items[0][i]
                voltageAdjustment = steeringCommand / self.efcSlope
                newVoltage = round(vFloat + voltageAdjustment, 3)

                #make sure we are above the voltage min / max of the ocxo
                if newVoltage > self.vmax:
                    newVoltage = self.vmax
                if newVoltage < self.vmin:
                    newVoltage = self.vmin


                print("New voltage request is:")
                print(newVoltage)

                voltString = ':SOURce2:VOLTage ' + str(newVoltage)
                print(voltString)

                myInst.write(voltString)
                #reset the counter
                self.counter = 0

            self.counter = self.counter + 1
```

```
output_items[0][:] = input_items[0] * self.example_param
return len(output_items[0])
```

APPENDIX M - DAC Arduino Code

```
/*!
Linear Technology DC1684AA Demonstration Board.
LTC2758: Dual Serial 18-Bit SoftSpan IOUT DAC

@verbatim
NOTES
  Setup:
   Set the terminal baud rate to 115200 and select the newline terminator.

   An external +/- 15V power supply is required to power the circuit.

  Explanation of Commands:
   1- Select DAC
   Select between DAC A, DAC B, or both.

   2- Change Span of Selected DAC
   |   Command   | Range Selected |
   | C3 C2 C1 C0 |                |
   |-------------|----------------|
   | 1  0  0  0  |    0V - 5V     |
   | 1  0  0  1  |    0V - 10V    |
   | 1  0  1  0  |   -5V - +5V    |
   | 1  0  1  1  |  -10V - +10V   |
   | 1  1  0  0  | -2.5V - +2.5V  |
   | 1  1  0  1  | -2.5V - +7V    |

   3- Voltage Output
   Displays the calculated voltage depending on the code input from user and
   voltage range selected.

   4- Square wave output
   Generates a square wave on the output pin. This function helps to measure
   settling time and glitch impulse.

USER INPUT DATA FORMAT:
 decimal : 1024
 hex     : 0x400
 octal   : 02000  (leading 0 "zero")
 binary  : B10000000000
 float   : 1024.0

@endverbatim
```

```
http://www.linear.com/product/LTC2758

http://www.linear.com/product/LTC2758#demoboards


Copyright 2018(c) Analog Devices, Inc.

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
 - Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
 - Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in
   the documentation and/or other materials provided with the
   distribution.
 - Neither the name of Analog Devices, Inc. nor the names of its
   contributors may be used to endorse or promote products derived
   from this software without specific prior written permission.
 - The use of this software may or may not infringe the patent rights
   of one or more patent holders.  This license does not release you
   from the requirement that you obtain separate licenses from these
   patent holders to use this software.
 - Use of the software either in source or binary form, must be run
   on or directly connected to an Analog Devices Inc. component.

THIS SOFTWARE IS PROVIDED BY ANALOG DEVICES "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, NON-INFRINGEMENT,
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL ANALOG DEVICES BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, INTELLECTUAL PROPERTY RIGHTS, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/*! @file
    @ingroup LTC2758
*/

// Headerfiles
#include "LT_SPI.h"
#include "UserInterface.h"
#include "LT_I2C.h"
```

```
#include "QuikEval_EEPROM.h"
#include "Linduino.h"
#include <SPI.h>
#include <Stream.h>
#include "LTC2758.h"

// Global variables
static uint8_t demo_board_connected;   //!< Set to 1 if the board is connected

float DACA_RANGE_LOW = 0;
float DACA_RANGE_HIGH = 5;

float DACB_RANGE_LOW = 0;
float DACB_RANGE_HIGH = 5;

float startVoltage = 0.0f;
// float startVoltage = -2.5f;

float stepSize = 0.5f;
float stepSize1mv = 0.009994f;

float currentVoltage;

uint32_t data;

uint8_t DAC_SELECTED = ADDRESS_DACA;

// Function Declarations
void print_title();
void print_prompt();
uint8_t menu1_select_dac();
void menu2_change_range();
uint8_t menu3_voltage_output();
uint8_t menu4_square_wave_output();

//! Initialize Linduino
void setup()
{
  char product_name[] = "LTC2758";  // Product Name stored in QuikEval EEPROM
  char board_name[] = "DC1684";     // Demo Board Name stored in QuikEval EEPROM

  quikeval_SPI_init();              // Configure the spi port for 4MHz SCK
  quikeval_SPI_connect();           // Connect SPI to main data port
  quikeval_I2C_init();              // Configure the EEPROM I2C port for 100kHz

  Serial.begin(115200);             // Initialize the serial port to the PC
  print_title();
```

```
  Serial.print("Initial voltage: ");
  Serial.print(startVoltage);
  Serial.print("V, Voltage Step Size: ");
  Serial.print(stepSize);
  Serial.print("V");

  // assign the starting voltage as the current voltage
  currentVoltage = startVoltage;

  demo_board_connected = discover_DC1684AB(product_name, board_name);
  if (demo_board_connected)
  {
    print_prompt();
  }
  LTC2758_write(LTC2758_CS, LTC2758_WRITE_SPAN_DAC, ADDRESS_DAC_ALL, 0);  // initialising a

  // LTC2758_write(LTC2758_CS, LTC2758_WRITE_SPAN_DAC, ADDRESS_DAC_ALL, 4);  // initialisi
  LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, ADDRESS_DAC_ALL, 0); // initiali

  // initally change range to something
  uint32_t span;
  Serial.println("\n| Choice | Range          |");
  Serial.println("|--------|----------------|");
  Serial.println("|    4   | -2.5 - +2.5 V |");

  span = (uint32_t)(4 << 2);

  // change DACA and B
  DACA_RANGE_LOW = -2.5;
  DACA_RANGE_HIGH = 2.5;

  DACB_RANGE_LOW = -2.5;
  DACB_RANGE_HIGH = 2.5;

  LTC2758_write(LTC2758_CS, LTC2758_WRITE_SPAN_DAC, ADDRESS_DAC_ALL, span);

}

//! Read the ID string from the EEPROM and determine if the correct board is connected.
//! Returns 1 if successful, 0 if not successful
uint8_t discover_DC1684AB(char *product_name, char *board_name)
{
  Serial.print(F("\nChecking EEPROM contents..."));
  read_quikeval_id_string(&ui_buffer[0]);
  ui_buffer[48] = 0;
  // Serial.println(ui_buffer);
```

```
  if (!strcmp(demo_board.product_name, product_name) && !strcmp(demo_board.name, board_name
  {
    Serial.print("\nDemo Board Name: ");
    Serial.println(demo_board.name);
    Serial.print("Product Name: ");
    Serial.println(demo_board.product_name);
    if (demo_board.option)
    {
      Serial.print("Demo Board Option: ");
      Serial.println(demo_board.option);
    }
    Serial.println(F("Demo board connected..."));
    Serial.println(F("\n\n\t\t\t\tPress Enter to Continue..."));

    // is this the source of the enter request?
    //read_int();
    return 1;
  }
  else
  {
    Serial.print("Demo board ");
    Serial.print(board_name);
    Serial.print(" not found, \nfound ");
    Serial.print(demo_board.name);
    Serial.println(" instead. \nConnect the correct demo board, then press the reset buttor
    return 0;
  }
}

//! Repeats Linduino loop
void loop()
{

  // this is the loop that we load onto the DAC during any steering
  float voltage;
  if (Serial.available())               // Check for user input
  {
  Serial.print("\nEnter voltage: ");
  while (!Serial.available());
    voltage = read_float();
    Serial.print(voltage);
    Serial.println(" V");

    data = LTC2758_voltage_to_code(voltage, DACA_RANGE_LOW, DACA_RANGE_HIGH);
    LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, data);
    Serial.print("\nDACA Output voltage = ");
```

```
    Serial.print(voltage);
    Serial.println(" V");
    Serial.print("\nDAC CODE: 0x");
    Serial.println(data, HEX);

  }


  // This loop is the default loop provided with this code.
  /*
  int16_t user_command;
  if (Serial.available())               // Check for user input
  {
    user_command = read_int();          // Read the user command
    Serial.println(user_command);
    Serial.flush();
    switch (user_command)
    {
      case 1:
        menu1_select_dac();
        break;
      case 2:
        menu2_change_range();
        break;
      case 3:
        menu3_voltage_output();
        break;
      case 4:
        menu4_square_wave_output();
        break;
      default:
        Serial.println(F("Incorrect Option"));
        break;
    }
    Serial.println(F("\n*************************************************************"));
    print_prompt();
  }
  */


}

//! Prints the title block when program first starts.
void print_title()
{
  Serial.println();
  Serial.println(F("*****************************************************************"));
```

```
  Serial.println(F("* DC1684A-A Demonstration Program
*"));
  Serial.println(F("*
*"));
  Serial.println(F("* This program demonstrates how to send data to the LTC2758
*"));
  Serial.println(F("* Dual Serial 18-bit Soft Span DAC
*"));
  Serial.println(F("*
*"));
  Serial.println(F("* Set the baud rate to 115200 and select the newline terminator.*"));
  Serial.println(F("*
*"));
  Serial.println(F("*****************************************************************\n"));
}

//! Prints main menu.
void print_prompt()
{
  Serial.println(F("\nCommand Summary:"));
  Serial.println(F("\n  1. Select DAC"));
  Serial.println(F("  2. Change Span of selected DAC"));
  Serial.println(F("  3. Voltage Output"));
  Serial.println(F("  4. Square wave output"));

  Serial.println(F("\nPresent Values:\n"));
  Serial.print(F("  DAC A Range: "));
  Serial.print(DACA_RANGE_LOW);
  Serial.print(F(" V to "));
  Serial.print(DACA_RANGE_HIGH);
  Serial.println(F(" V"));

  Serial.print(F("  DAC B Range: "));
  Serial.print(DACB_RANGE_LOW);
  Serial.print(F(" V to "));
  Serial.print(DACB_RANGE_HIGH);
  Serial.println(F(" V"));

  Serial.print(F("\n  Selected DAC: "));
  switch (DAC_SELECTED)
  {
    case ADDRESS_DACA:
      Serial.println(F("DAC A"));
      break;
    case ADDRESS_DACB:
      Serial.println(F("DAC B"));
      break;
```

```
    case ADDRESS_DAC_ALL:
      Serial.println(F("ALL DACs"));
      break;
  }

  Serial.print(F("\n\nEnter a command: "));
  Serial.flush();
}

//! Function to select DAC and set DAC address
uint8_t menu1_select_dac()
{
  uint8_t choice;
  Serial.println(F("\n1. DAC A"));
  Serial.println(F("2. DAC B"));
  Serial.println(F("3. All DACs"));
  Serial.print(F("\nEnter a choice: "));
  choice = read_int();          // Read the user command

  switch (choice)
  {
    case 1:
      DAC_SELECTED = ADDRESS_DACA;
      Serial.println("DAC A");
      break;
    case 2:
      DAC_SELECTED = ADDRESS_DACB;
      Serial.println("DAC B");
      break;
    default:
      DAC_SELECTED = ADDRESS_DAC_ALL;
      Serial.println("ALL DACs");
      break;
  }
  Serial.flush();
}

//! Function to choose the range of voltages to be used
void menu2_change_range()
{

  uint8_t choice;
  uint32_t span;
  Serial.println("\n| Choice | Range          |");
  Serial.println("|--------|----------------|");
  Serial.println("|    0   |    0 - 5 V     |");
  Serial.println("|    1   |    0 - 10 V    |");
```

```
Serial.println("|    2    |   -5 - +5 V    |");
Serial.println("|    3    |  -10 - +10 V   |");
Serial.println("|    4    | -2.5 - +2.5 V  |");
Serial.println("|    5    | -2.5 - +7.5 V  |");

Serial.print("\nEnter your choice: ");
choice = read_int();
Serial.println(choice);
span = (uint32_t)(choice << 2);
if (DAC_SELECTED == ADDRESS_DACA || DAC_SELECTED == ADDRESS_DAC_ALL)
{
  switch (choice)
  {
    case 0:
      DACA_RANGE_LOW = 0;
      DACA_RANGE_HIGH = 5;
      break;

    case 1:
      DACA_RANGE_LOW = 0;
      DACA_RANGE_HIGH = 10;
      break;

    case 2:
      DACA_RANGE_LOW = -5;
      DACA_RANGE_HIGH = 5;
      break;

    case 3:
      DACA_RANGE_LOW = -10;
      DACA_RANGE_HIGH = 10;
      break;

    case 4:
      DACA_RANGE_LOW = -2.5;
      DACA_RANGE_HIGH = 2.5;
      break;

    case 5:
      DACA_RANGE_LOW = -2.5;
      DACA_RANGE_HIGH = 7.5;
      break;

    default:
      Serial.println("\nWrong choice!");
  }
  Serial.print(F("Span Changed!"));
```

```
  }
  if (DAC_SELECTED == ADDRESS_DACB || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    switch (choice)
    {
      case 0:
        DACB_RANGE_LOW = 0;
        DACB_RANGE_HIGH = 5;
        break;

      case 1:
        DACB_RANGE_LOW = 0;
        DACB_RANGE_HIGH = 10;
        break;

      case 2:
        DACB_RANGE_LOW = -5;
        DACB_RANGE_HIGH = 5;
        break;

      case 3:
        DACB_RANGE_LOW = -10;
        DACB_RANGE_HIGH = 10;
        break;

      case 4:
        DACB_RANGE_LOW = -2.5;
        DACB_RANGE_HIGH = 2.5;
        break;

      case 5:
        DACB_RANGE_LOW = -2.5;
        DACB_RANGE_HIGH = 7.5;
        break;

      default:
        Serial.println("\nWrong choice!");
    }
    Serial.print(F("Span Changed!"));
  }
  LTC2758_write(LTC2758_CS, LTC2758_WRITE_SPAN_DAC, DAC_SELECTED, span);
}

//! Function to enter a digital value and get the analog output
uint8_t menu3_voltage_output()
{
  uint8_t choice;
```

```
uint32_t data;
float voltage;

Serial.println(F("\n1. Enter Voltage"));
Serial.println(F("2. Enter Code"));
Serial.print(F("\nEnter a choice: "));
choice = read_int();          // Read the user command
Serial.print(choice);

if (choice == 2)
{
  Serial.print("\nEnter the 18-bit data as decimal or hex: ");
  data = read_int();
  Serial.print("0x");
  Serial.println(data, HEX);

  if (DAC_SELECTED == ADDRESS_DACA || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    voltage = LTC2758_code_to_voltage(data, DACA_RANGE_LOW, DACA_RANGE_HIGH);
    Serial.print("\nDACA Output voltage = ");
    Serial.print(voltage);
    Serial.println(" V");
  }

  if (DAC_SELECTED == ADDRESS_DACB || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    voltage = LTC2758_code_to_voltage(data, DACB_RANGE_LOW, DACB_RANGE_HIGH);
    Serial.print("\nDACB Output voltage = ");
    Serial.print(voltage);
    Serial.println(" V");
  }
  LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, data);
}
else if (choice == 1)
{
  Serial.print("\nEnter voltage: ");
  while (!Serial.available());
  voltage = read_float();
  Serial.print(voltage);
  Serial.println(" V");

  if (DAC_SELECTED == ADDRESS_DACA || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    data = LTC2758_voltage_to_code(voltage, DACA_RANGE_LOW, DACA_RANGE_HIGH);
    LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, data);
    Serial.print("\nDACA Output voltage = ");
    Serial.print(voltage);
```

```
      Serial.println(" V");
      Serial.print("\nDAC CODE: 0x");
      Serial.println(data, HEX);
    }

    if (DAC_SELECTED == ADDRESS_DACB || DAC_SELECTED == ADDRESS_DAC_ALL)
    {
      data = LTC2758_voltage_to_code(voltage, DACB_RANGE_LOW, DACB_RANGE_HIGH);
      LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, data);
      Serial.println(DACB_RANGE_HIGH);
      Serial.print("\nDACB Output voltage = ");
      Serial.print(voltage);
      Serial.println(" V");
      Serial.print("\nDAC CODE: 0x");
      Serial.println(data, HEX);
    }
  }
  return 0;
}

//! Function to generate a square wave of desired frequency and voltage ranges
uint8_t menu4_square_wave_output()
{
  uint16_t freq;
  float time;
  float voltage_high, voltage_low;
  uint32_t code_high, code_low;
  uint8_t receive_enter;  // To receive enter key pressed

  Serial.print("\nEnter voltage_high: ");
  while (!Serial.available());
  voltage_high = read_float();
  Serial.print(voltage_high);
  Serial.println(" V");

  Serial.print("\nEnter voltage_low: ");
  while (!Serial.available());
  voltage_low = read_float();
  Serial.print(voltage_low);
  Serial.println(" V");

  Serial.print("\nEnter the required frequency in Hz: ");
  freq = read_int();
  Serial.print(freq);
  Serial.println(" Hz");

  time = (float)1000/freq;
```

```
  Serial.print("\nT = ");
  Serial.print(time);
  Serial.println(" ms");

  //! Converting voltage into data
  if (DAC_SELECTED == ADDRESS_DACA || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    code_high = LTC2758_voltage_to_code(voltage_high, DACA_RANGE_LOW, DACA_RANGE_HIGH);
    code_low = LTC2758_voltage_to_code(voltage_low, DACA_RANGE_LOW, DACA_RANGE_HIGH);
  }
  if (DAC_SELECTED == ADDRESS_DACB || DAC_SELECTED == ADDRESS_DAC_ALL)
  {
    code_high = LTC2758_voltage_to_code(voltage_high, DACB_RANGE_LOW, DACB_RANGE_HIGH);
    code_low = LTC2758_voltage_to_code(voltage_low, DACB_RANGE_LOW, DACB_RANGE_HIGH);
  }

  while (!Serial.available()) //! Generate square wave until a key is pressed
  {
    LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, code_high);
    delayMicroseconds(time * 500);
    LTC2758_write(LTC2758_CS, LTC2758_WRITE_CODE_UPDATE_DAC, DAC_SELECTED, code_low);
    delayMicroseconds(time * 500);
  }
  receive_enter = read_int();
  return 0;
}
```

DISTRIBUTION LIST

DTIC/OCP
8725 John J. Kingman Rd, Suite 0944
Ft Belvoir, VA 22060-6218 1 cy

AFRL/RVIL
Kirtland AFB, NM 87117-5776 1 cy

Official Record Copy
AFRL/RVB/Dr. Spencer E. Olson 1 cy

This page is intentionally left blank.