Section: CS

Lowering Avionics Bus Trust: Moving ARINC 429 Bus Architecture Towards Zero Trust

Matthew Preston

Problem Statement:

ARINC 429 bus architecture, like most bus architectures, is insecure. Any command on the bus will be inherently trusted and executed by receiver LRUs. This means that compromising legacy systems and software can potentially cause catastrophic effects. Since 2022, the United States government has stated intent to securing its assets with Zero Trust principles. Therefore, moving ARINC 429 to zero trust without having to design and implement a whole new architecture or standard would help closer align the cybersecurity posture of critical infrastructure to this intent.

Solution Statement:

This research will simulate an ARINC 429 bus (as a digital twin) as well as implement an ARINC 429 traffic/rules-based intrusion detection system to engineer a defense system for ARINC 429 that helps move the ARINC 429 bus architecture towards zero trust. It will have the following deliverables:

- A simulation of an ARINC 429 bus architecture of and airplane
- A rules-based IDS that can log and flag ARINC 429 words.

My project will only move ARINC429 towards zero-trust and not achieve a 100% zero-trust architecture for ARINC429. If fully achieved for ARINC429 in the future, a Zero Trust architecture is the right approach for securing the ARINC429 bus protocol because it would shift the security posture from assuming words on the wire are trusted commands to double checking each command attempt. Given the critical nature of aviation systems and the increasing sophistication of cyber threats, Zero Trust would ensure that only authenticated and authorized devices can interact with the ARINC429 bus. By continuously monitoring and validating all interactions between LRUs, a full Zero Trust ARINC429 architecture would enhance the overall security posture of internal aviation communication systems.

My rules-based IDS is geared towards alerting an aircraft crew of a cyber threat. If it flags a word or sequence of words as malicious I envision it alerting someone (i.e. a pilot, or trained flight attendant) to let them know of a cyber-attack. This would then allow for better situational awareness and hopefully control of a pilot crew to then land in an emergency and also for investigators after the fact to learn what exactly happened.

Completed Tasks (Last 2 Week):

Here's what I promised from the last progress report:

- Finalizing the flight simulator to have positional data to feed into LRUs in the future.
    - This one has been delayed/cancelled pending peer feedback, multiple group members said this was superfluous and not needed to prove my point.
- Create the code that will be the Flight Management Computer LRU simulation.
    - I completed this task: please see details below.
- Create the code that will be the weight and balance system LRU simulation.
    - I completed this task: please see details below.
- Create the code that will be the GPS LRU simulation.
    - I completed this task: please see details below.

- Create the code that will be the radio management system LRU simulation.
    - I completed this task: please see details below.
- Test data connection hookup between the EEC and W&BS LRUs and the flight simulator.
    - I ran into an issue with this task and will detail below. However, I do have some bus simulation going early and programmed TX and RX bus simulated voltages to and from blank LRUs.

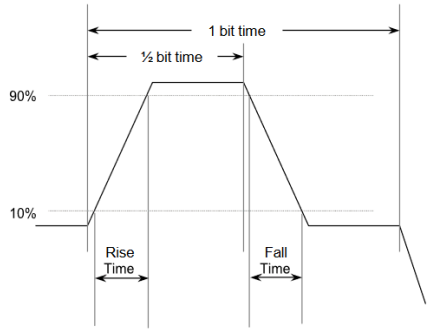
I created and completed the following Python classes for my simulation. In total, this amounts to 3035 lines of code written for the past 2 weeks (1100ish of which are part of test code). As a side note, when I refer to the bus channels "orange", "blue", "green", and "purple", please refer to the system model at the top of the appendix.

1. "`arinc429_voltage_sim.py`"
    a. This class simulates the voltage produced on a bus for ARINC429 within tolerances of the spec. It is the start of the bus simulation.
    b. It can create single bit voltage sets for a one a zero for high and low speed buses. The spec details voltage as:

```
# TX
# HIGH  (i.e. 1) =>      10.0 V +/- 1.0 V
# NULL                    0.0 V +/- 0.5 V
# LOW   (i.e. 0) =>     -10.0 V +/- 1.0 V

# RX
# HIGH  (i.e. 1) =>     [+ 6.5V, +13.0V]
# NULL                  [- 2.5V, + 2.5V]
# LOW   (i.e. 0) =>     [-13.0V, - 6.5V]
```

    c. It graphs the voltages over ½ microsecond intervals (µsec). Each bit is four phases: a rise/fall, hold at voltage, return to null, and then hold null over a defined amount of time for µsecs for each phase. So a 1 would be rise to +10 V, hold at +10V, fall to 0V and then hold at 0V. This is visualized as:



    d. The timing of voltage sets is implemented as the spec defines. This spec definition is as follows:
        i. High Speed:
            1. Phase 1: Rise/Fall is ½ to 2 µsecs.
            2. Phase 2: Hold at +/- 10V is 4 ½ to 5 ½ µsecs.
            3. Phase 3: Return to Null is ½ to 2 µsecs.
            4. Phase 4: Hold at Null is is 4 ½ to 5 ½ µsecs.
        ii. One full bit time is in the range [10 µsecs, 15 µsecs]

iii. Low Speed
    1. Phase 1: Rise/Fall is 5 to 15 μsecs.
    2. Phase 2: Hold at +/- 10V is 40 to 44 μsecs.
    3. Phase 3: Return to Null is 5 to 15 μsecs.
    4. Phase 4: Hold at Null is is 40 to 44 μsecs.
iv. One full bit time is in the range of [90 μsecs, 118 μsecs]

e. It translates a set of voltages over a set of times to a word.
f. It creates respective null time between words (spec details 4 bits worth of null time)
g. It can create a random word as a set of voltages over times.
h. It can create n random words as a set of voltages over times.
i. It can translate from a bitstring word, i.e. `"01010101111100000101010111110000"` to a set of voltages over times.
j. It can translate from an integer (type) in the set [`0b0`, `0b11111111111111111111111111111111`] to a set of voltages over times.
k. These are all used by the `BusQueue` and TX LRUs to create voltages to then "send" over the bus (simulated). So for example, if the GPS creates the word for a latitude of `N 75 Deg 59.9'`, or `0b00010000000100110011010101011100`, it can then turn that into a set of voltages over times for a high speed or low speed. Then it uses threading to send that voltages one by one to the bus.
l. See the appendix for visual graphics of the tests for this class.

2. "`BusQueue_Simulator.py`"
    a. This is the bus class that simulates the bus. It uses a FIFO queue of 100 voltages at once. This can be thought of as the wire that the voltages is travelling on. The TX LRU puts voltages onto this bus that it generates waiting ½ μsec in between voltages. So after 100 voltages the voltages drops off the bus. The RX LRUs have access to this queue and grab the voltage at index 75 in this bus to simulate a time delay of the voltage going down the wire. They then take the voltages received and decode them to words.
    b. It also has functionality to visualize the bus as the voltages come and go.
    c. It also has functionality to simulate EMI noise to the wire tampering with the voltages. This functionality is currently untested and unused but will be incorporated later into robustness testing during the IDS.
    d. Since it is an object class, this class can be instantiated to make multiple bus channels.
3. "`LRU_FAEC_Simulator.py`"
    a. This is the object class for the Full Authority Engine Control. It takes in words from the FMC and outputs thrust on the engines, etc.
    b. It decodes all the words pertaining to the FAEC and then actuates (mostly by printing out what its doing) upon them.
4. "`LRU_FMC_Simulator.py`"
    a. This LRU is based loosely on the United Electronic Industries DNx-429-516 computer, it has a FIFO mode where words are generated first in and sent first out to the buses, and scheduler mode where words are generated and then sent out based on criteria.
    b. It has pilot input to word generator where the arrow and "W" and "S" keys are the pilots way of pitching up, down, left, right, and accelerate and decelerate. This will send words to the W&BS and FAECs.

```
# UP -> Pitch plane up
if(keyboard.is_pressed('up')):
    self.generate_word_to_pitch_plane("UP")
# DOWN -> Pitch plane down
if(keyboard.is_pressed('down')):
    self.generate_word_to_pitch_plane("DOWN")
# LEFT -> pitch plane left
if(keyboard.is_pressed('left')):
    self.generate_word_to_pitch_plane("LEFT")
# RIGHT -> pitch plane right
if(keyboard.is_pressed('right')):
    self.generate_word_to_pitch_plane("RIGHT")
# W -> push plane forward
if(keyboard.is_pressed('w')):
    self.generate_word_to_pitch_plane("W")
# S -> slow plane down and go backwards
if(keyboard.is_pressed('s')):
    self.generate_word_to_pitch_plane("S")
```

   c. I haven't added it yet, but it will have the opportunity to have a maintenance mode where it opens a subprocess for the mx program (below) and generates words based on whatever that program tells it to.
   d. It can RX words from the blue bus and can send words to the green and purple bus.
5. "`LRU_GPS_Simulator.py`"
   a. This LRU generates a latitude and longitude (in real life this would be from a satellite, however because this is a simulation, the data is self made in this file) and converts that to words to send to the orange bus.
   b. It updates the position of the latitude and longitude.
6. "`LRU_RMS_Simulator.py`"
   a. This LRU decodes words and takes those commands from the bus and:
      i. Sets a specific radio frequency (just prints "new frequency for X radio")
      ii. Sets information for the ADS-B message like:

```
# ADS-B Broadcasts the following information from a plane
# Flight Number - BCD
# Latitude - BCD, BNR
# Longitude - BCD, BNR
# Altitude - BCD, BNR
# Ground Speed - BCD, BNR
# Vertical Speed - BCD
# Track Angle - BCD
# Magnetic Heading - BCD
# Emergency Status -> special.
# Ident Switch
#       -> A signal that the pilot can activate to highlight the aircraft on air traffic control screens.
# ICAO Address - DISC
# Aircraft Type - NOT OVER BUS
```

7. "`LRU_WnBS_Simulator.py`"
   a. This LRU takes given words from the bus and outputs print statements that show the plane pitching left or right.
   b. It is supposed to simulate the actuators for plane navigation control.
8. "`LRU_RX_Helper.py`"
   a. This class is a collection of common functions that recieve words across a bus channel in real time.
   b. This class receives a word from a bus channel object and converts it into a word integer.
   c. It gets a label from a word int object (*see problems).
   d. It validates a word by checking the parity.
   e. It grabs a single voltage from the wire.
   f. It visualizes/animates voltages received from the wire.
   g. These functions are all shared and used by RX LRUs, so it was easier to make them into one shared class resource instead of coding them all over again.
9. "`LRU_TX_Helper.py`"

    a. This class is a collection of common functions that transmit words across a bus channel in real time.
    b. It can transmit random words to the bus for testing purposes.
    c. It can transmit a given word to the bus.
    d. It validates words based on parity bit.
    e. It can transmit a single voltage to the bus wire.
    f. It makes labels for words (*see problems).
10. "`main.py`"
    a. This is going to be the file that runs the whole demo by creating each LRU class object instances, and then running the communication between them. Right now it is just an outline.
11. "`maintenance_program.c`"
    a. This is a vulnerable program that will be used in the FMC as an IP hookup. I have not started developing the attack for this, just the code.
12. "`MegaTest.py`"
    a. This is a large testing base for all the classes above. It uses the `pytest` module to assert that things are working.
    b. There are also tests that just show things working, since I will have to use multi-threading in this project. These tests have no assertation.

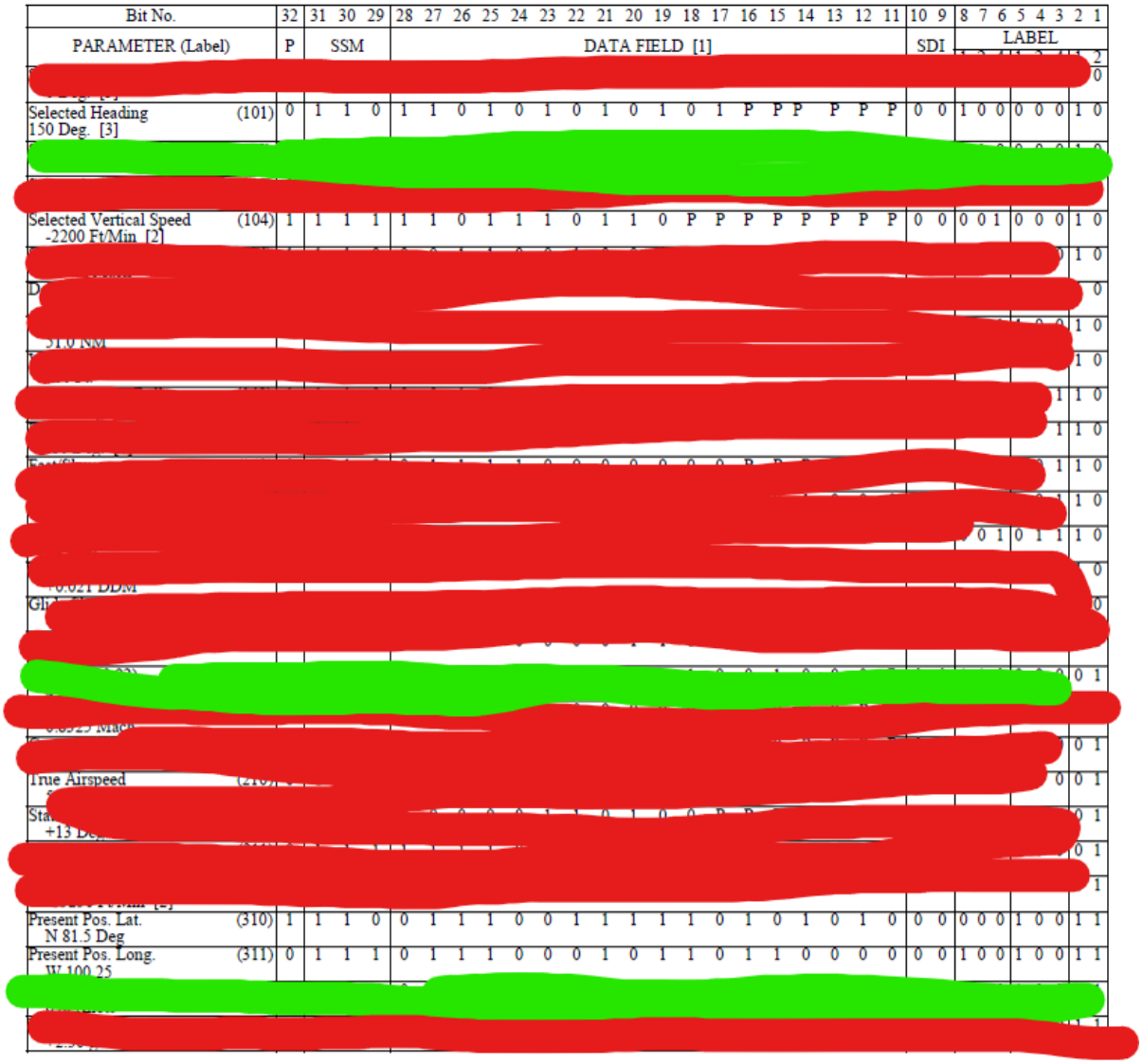
Questions I have or Issues I'm running into:

1. I implemented label encoding and decoding slightly incorrectly, which slowed me down. This is a self-reported issue in my git repository for this. This has slowed me down in testing connection between the FMC and the FAEC and W&BS. The way I encoded it was treating the full 8 bits as one number together and transmitting that backwards, rather than separating the bits into their respective digits. For example, label `0o011` is the label to transmit longitude. In binary that is: `0b00001001`, and labels are transmitted backwards from bit 8 to 1, so I encoded it to: `0b10010000`. However, the grouping of digits is digit 3: 8-7-6, digit 2: 5-4-3, digit 1: 2-1 for the label. So the label should be: digit 1 = one which is `0b01`, digit 2 = one which is `0b001`, and digit 3 = zero which is `0b000`. Ergo the bits in order from 1 to 8 are `0b01001000`, and transmitted backwards as `0b00010010`. You can see that this is not the same as my accidental encoding of `0b10001000 =/= 0b00010010`. This should be easy to fix as it is modifying two functions in the classes: "`LRU_RX_Helper.py`" and "`LRU_TX_Helper.py`".
2. Something that set be back a day of progress was implementing a bad and unreliable bus class (deleted from above). This bus was sending packets to the loopback address with the voltage and receiving them. It was unreliable so I had to switch to a queue method holding the most recent voltages for TX LRUs to send voltages to and RX LRUs to fetch voltages from, further down the queue. This wasted about 4-5ish hours of work, so not too much of the total time spent this week.
3. I am having trouble receiving bus words over the high speed option. This is not a task that I needed to get done this week, but I will probably need to implement a better voltage to word decoder in `arinc429_voltage_sim.py` soon.
4. I am using Python 11. This version of python's matplotlib is unstable in IDLE, and craps out if you try to graph a lot of things at once. Here's a screenshot from testing and graphing outputs of all the functions in `arinc429_voltage_sim.py`:

Thankfully, PyCharm does not have the same issue, but the graphs are not as customizable with the GUI interface (as opposed to coding the size).

5. I use both PyCharm and Python 11 for the graphics, and PyCharm is not without its own issues with matplotlib via threading. It cannot refresh graphs, and I haven't found a solution online. Thankfully, I can use IDLE to animate the voltage across the wire. Ironically, IDLE works better here with matplotlib if it's only 1 or 2 graphics on screen. Both problems (#4 and #5) are small, and I have found work arounds, as noted here. The animation is shown in my video posted this week as it is hard to attach a video file to the appendix.
6. Some of the LRUs, in particular the FMC and RMS were much larger of an endevor than I initially thought they would be, adding more workhours needed to get them done.
7. I need to better formalize my evaluation on my IDS and bus simulator, and I am not sure how to make a formal framework for something like that, that is also feasible.
8. I cannot understand some of the ARINC 429 BNR encoding data from the spec sheet. For example:

Table 6-27 BNR DATA ENCODING EXAMPLES



Bit No.		32	31 30 29	28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11	10 9	8 7 6 5 4 3 2 1
PARAMETER (Label)		P	SSM	DATA FIELD [1]	SDI	LABEL
Selected Heading 150 Deg. [3]	(101)	0	1 1 0	1 1 0 1 0 1 0 1 0 1 0 1 P P P P P P	0 0	1 0 0 0 0 0 1 0
Selected Vertical Speed -2200 Ft/Min [2]	(104)	1	1 1 1	1 1 0 1 1 1 0 1 1 0 P P P P P P P P	0 0	0 0 1 0 0 0 1 0
Present Pos. Lat. N 81.5 Deg	(310)	1	1 1 0	0 1 1 1 0 0 1 1 1 1 1 0 1 0 1 0 1 0	0 0	0 0 0 1 0 0 1 1
Present Pos. Long. W 100.25	(311)	0	1 1 1	0 1 1 1 0 0 0 1 0 1 1 0 1 1 0 0 0 0	0 0	1 0 0 1 0 0 1 1

How does the data on the right translate into the given values? I have zero clue. Here, a `0bP` bit means it is not used, like it's a `pass` bit. Usually, it's set to zero.

- Heading data: `0bPPPPPP101010101011`, SSM: `0b011` (the zero at the beginning of the SSM is the sign) -> this somehow translates into 150 degrees. However, this is 2731 in base 10, and is not enough bits to meet IEEE floating point notation.
- Similarly for Vertical Speed: -2200 =?= `0bPPPPPPPPP0110111011`, Latitude: 81.5 degrees =?= `0b010101011111001110` and Longitude: 100.25 =?= `0b000011011010001110`.

Thankfully for these data types I was able to use and implement the BCD labels equivalent, so it's not a huge deal; I can just use a different word format.

Methodology Paragraph Summary:

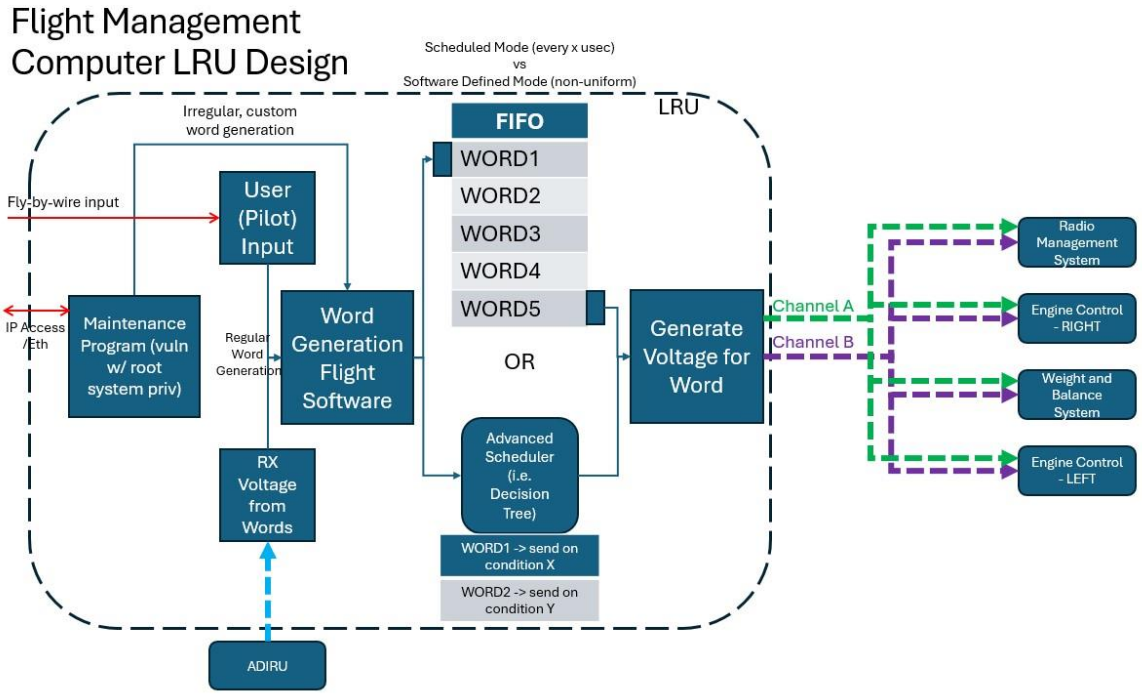
The process I will be employing is:

1. Plan structure of next component I need to build
2. Build component based on plan
3. Test component functionality by making test cases
4. Rework component based on test
5. Test component working with other components

Repeat until each component is finished.

Here's an example with the FMC.

- First I designed the component below, based on a description from UEI (similarly to DNx-429-516):



- Then I coded each component in python to the "`LRU_FMC_Simulator.py`".
- Then I used my test file to test various functionality of it, i.e. the pilot input from my keys, the FIFO queue, the scheduler, the voltage generation, the word generation etc. From there I fixed what needed to be fixed.
- Finally, in my same test file, I will test interaction between it and other components, i.e. if the word sent can be received ok, and decoded ok by the other attached components in the picture.

For evaluation, I will be using a data-driven approach:

1. Gather data during evaluation phase.
    a. Gather word generation data.
    b. Gather voltage generation data.
    c. Label malicious words as malicious based on my attack generated in that dataset.
    d. Label good words as good in that dataset.
2. Use data to see if the simulation/solution is robust, impacts functionality, and is accurate.
    a. Robustness should be:
        i. can the bus simulator operate at or close to real-world speed?

ii. Is the bus simulator slowed down or hampered by adding each additional component?
iii. Is the bus simulator slowed down or hampered by an IDS?
iv. I will be using a separate test class file and python's time module to check this, once I get the bus up and running by reporting how much time it slows down per component if at all.
v. Also, I want robustness based on simulated voltage. Can the bus and components handle EMI noise interference to spec? That is a function of the `BusQueue` class, which adds noise.
vi. If the IDS can catch bad words, how long would it take to alert someone?

b. Functionality will be evaluated based on how closely it aligns with specs. In particular, how many words can each LRU handle based on specifications on what it should be able to?
   i. Can the bus receive words at both High and Low bus speeds?

c. Accuracy:
   i. Does the bus create words based on the spec?
   ii. Can the IDS accurately tell between malicious words and good words?

Timeline:

Week #	**Description of Task**	**Status**
Week 1 Monday, 13 May 2024 – Sunday 19 May 2024	Research topic by reading articles. Focus the research on understanding how the ARINC 429 protocol works.	Completed
	Research topic by reading articles. Focus research on various bus architecture security solutions, to see what has been developed. This research has helped narrow down the architecture from the list of ARINC 429, CAN bus, 1553 bus, and ARINC 629 to just ARINC 429.	Completed
	Research topic by reading articles. Focus the research on understanding zero trust cybersecurity, and if any zero trust implementation has been done for the above architectures. This helps inform a template for ARINC 429	Completed
	Draft Initial Proposal	Completed
Week 2 Monday, 20 May 2024 – Sunday 26 May 2024	Finalize Proposal	Completed
	Research topic by reading articles. Focus the research on any cybersecurity that has been done on ARINC 429, and if any of those help ARINC 429 implement zero trust principles/tenets.	Completed

	Simulate a receive LRU, the electronic engine control, that would be taking ARINC 429 commands and outputting actuations (for simulator).	Completed – Renamed to `full_authority_engine_control` as that was a more accurate LRU
	Start flight simulation software that should interface with LRU outputs. It should at this point just be an outline of the following functionality: - A tick/step-based time system that calculates the airplanes positional data from the last tick. Given the following attributes: altitude, x/y position, roll, yaw, pitch, forward velocity, and jet engine thrust, it should calculate the next x, y, altitude positional data. - Start at position 0, 0, 500 - Plot continuously the plan's positional data - A way to take in data from the actuators LRUs (jet engines, balancing LRUs for fins, etc) and translate that to the calculations. An outline here consists of creating the python file, putting in the math equations for steps/ticks for this, outlining a function that should take in data from an actuator and setting up the plotting given 3-d coordinates. This will be the codebase to test the ARINC429 simulation actuators from.	Completed
Week 3 Monday, 27 May 2024 – Sunday 2 June 2024	Simulate a transmitter LRU, the Flight Management Computer. It should have the following features: - Multiple TX channels - Multiple RX channels - Basic flight functionality software that generates ARINC 429 words based on desired direction to go	Completed
	Finalize functionality for the flight simulator above.	Cancelled – based on peer feedback.

	Simulate a receiver LRU that will be the weight and balance system on the plane. It should output actuator data for the simulation.	Completed
Week 4 Monday, 3 June 2024 – Sunday 9 June 2024	Simulate a transmitter LRU, the GPS that reports actual positional x/y/altitude data. It should get this back from the simulator above.	Completed
	Simulate a receiver LRU, the radio management system.	Completed
	Test interaction and hookup between the electric engine control LRU, and weight and balance system to the simulator to make sure they can input and influence the motion of the plane.	In Progress
Week 5 Monday, 10 June 2024 – Sunday 16 June 2024	Simulate a transmitter LRU, the ADIRU, and test its hookup to the flight simulator to make sure it can input data correctly.	Not started
	Create vulnerable program for FMC and a simple buffer overflow / rop hack for it to gain system access to the FMC.	In Progress
	Create the orange ARINC429 bus.	In Progress
	Create the blue ARINC429 bus.	Not started
Week 6 Monday, 17 June 2024 – Sunday 23 June 2024	Create final report outline.	Not Started
	Create the purple/channel B ARINC 429 bus	Not Started
	Create the green/channel A ARINC 429 bus	Not Started
	Add functionality to the attack above that when system access is gained on the FMC, start transmitting ARINC 429 words from the FMC to the FAECs ~~EEC~~s that pitch the plane into a downward trajectory.	Not Started
	Start the rules-based IDS system. Implement the functionality to it: - Create syntax for IDS rules - Create functionality for logging bus traffic	Not Started
	Start logging ARINC 429 traffic for data collection.	Not Started
Week 7	Create first draft of final report	Not Started

Monday, 24 June 2024 – Sunday 30 June 2024	Add the following functionality to the IDS: - Ability to generate alarms based on transmission ID - Ability to generate alarms based on parity & parity correctness - Ability to generate alarms based on sign/statis matrix -> normal operation (N/S, E/W), functional test, failure warning, no computed data - Ability to generate alarms based on data (hopefully also granulate that based on flight directional data, etc. Combine with previous words to see if there is a suspicious word). - Ability to generate alarms based on source/destination field - Ability to generate alarms based on label (data type)	Not Started
Week 8 Monday, 1 July 2024 – Sunday 7 July 2024	Revise the first draft of the final report into second draft.	Not Started
	Create a mode in the IDS that can reset the bus upon any of the alarms from IDS	Not Started
Week 9 Monday, 8 July 2024 – Sunday 14 July 2024	Create final demo presentation. It should include a demo of the simulation, attack, and defense implementations working with the attack.	Not Started
Week 10 Monday, 15 July 2024 – Sunday 21 July 2024	Create / post final demo video.	Not Started
Week 11 Monday, 22 July 2024 – Thursday 25 July 2024	Finish project final report from second draft.	Not Started

Evaluation:

To be delivered by progress report 4.

Report Outline:

To be delivered by progress report 5.

References:

R. Vincent. “ARINC-429 RX Implementation in Labview FPGA.” *Arinc-429 RX Implementation in LabVIEW FPGA*, NI Community, 28 Nov. 2023, https://forums.ni.com/t5/Example-Code/Arinc-429-Rx-Implementation-in-LabVIEW-FPGA/ta-p/3507624

aeroneous. “PyARINC429.” *Discover PyARINC429, a simple Python module for encoding and decoding ARINC 429 digital information.* 17 Jul. 2018, https://github.com/aeroneous/PyARINC429

Peña, Lisa; and Shipman, Maggie. “Episode 64: Zero-Trust Cybersecurity for Vehicles.” *Technology Today Podcast*, Southwest Research Institute, Feb. 2024, https://www.swri.org/podcast/ep64

“ARINC-429 with Cyber and Wirefault Protection” *ARINC-429 Solutions*. Sital Technology, https://sitaltech.com/arinc-429/

“Understanding Cyber Attacks on MIL-STD-1553 Buses” Sital Technology, https://sitaltech.com/understanding-cyber-attacks-on-mil-std-1553-buses/

“1553 Network and Cybersecurity Testing.” Alta Data Technologies LLC, 19 Jan. 2021, https://www.altadt.com/wp-content/uploads/dlm_uploads/2020/10/1553-Network-and-Cybersecurity-Testing.pdf

Tilman, Bill. “Why You Need to Secure Your 1553 MIL-STD Bus and the Five Things You Must Have in Your Solution.” Abaco Systems, 14 Dec. 2021, Original Link: https://abaco.com/blog/why-you-need-secure-your-1553-mil-std-bus-and-five-things-you-must-have-your-solution, Accessible Here: https://web.archive.org/web/20240223161240/https://abaco.com/blog/why-you-need-secure-your-1553-mil-std-bus-and-five-things-you-must-have-your-solution

Waldmann, B. “ARINC 429 Specification Tutorial.” *Avionics Databus Solutions*, Version 2.2, AIM Worldwide, Jul. 2019, https://www.aim-online.com/wp-content/uploads/2019/07/aim-tutorial-oview429-190712-u.pdf, https://www.aim-online.com/products-overview/tutorials/arinc-429-tutorial/

“ARINC-429 tutorial: A Step-by-Step Guide.” KIMDU Technologies, 26 Jun. 2023, https://kimdu.com/arinc-429-tutorial-a-step-by-step-guide/

“ARINC-429 Tutorial & Reference” *Understanding ARINC-429*, United Electronic Industries/AMETEK, https://www.ueidaq.com/arinc-429-tutorial-reference-guide

Biden, Joesph R. Jr. “Executive Order on Improving the Nation’s Cybersecurity.” *Briefing Room, Presidential Actions*, The White House, 12 May 2021, https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

Rose, Scott; Borchert, Oliver; Mitchell, Stu; and Connelly, Sean. “Zero Trust Architecture.” *NIST Special Publication 800-207*, National Institue of Standards and Technology, U.S. Department of Commerce, Aug. 2020, https://doi.org/10.6028/NIST.SP.800-207

Young, Shalanda D. “Moving the U.S. Government Toward Zero Trust Cybersecurity Principles” *MEMORANDUM FOR THE HEADS OF EXECUTIVE DEPARTMENTS AND AGENCIES*, Version M-22-09, Executive Office of the President; Office of Management and Budget, 26 Jan. 2022, https://whitehouse.gov/wp-content/uploads/2022/01/M-22-09.pdf

"Avionics Databus Tutorials." *Ballard Technology*, Astronics AES, https://www.astronics.com/avionics-databus-tutorials

maewert. "Interfacing Electronic Circuits to Arduinos." *Circuits; Arduino*, Autodesk Instructables, https://www.instructables.com/Interfacing-Electronic-Circuits-to-Arduinos/

Airlines Electronic Engineering Committee. "ARINC Specification 429 Part 1-17: Mark 33 – Digital Information Transfer System (DITS)." *ARINC Document*, Aeronautical Radio Inc. 17 May 2004, Original Link: https://read.pudn.com/downloads111/ebook/462196/429P1-17_Errata1.pdf , Accessible here: https://web.archive.org/web/20201013031536/https://read.pudn.com/downloads111/ebook/462196/429P1-17_Errata1.pdf

D. De Santo, C.S. Malavenda, S.P. Romano, C. Vecchio, "Exploiting the MIL-STD-1553 avionic data bus with an active cyber device." *Computers & Security*, Volume 100, 2021, 102097, ISSN 0167-4048, https://doi.org/10.1016/j.cose.2020.102097. (https://www.sciencedirect.com/science/article/pii/S0167404820303709)

Gilboa-Markevich, N., Wool, A. (2020). "Hardware Fingerprinting for the ARINC 429 Avionic Bus." In: Chen, L., Li, N., Liang, K., Schneider, S. (eds) Computer Security – ESORICS 2020. ESORICS 2020. Lecture Notes in Computer Science(), vol 12309. Springer, Cham. https://doi.org/10.1007/978-3-030-59013-0_3

Kiley, Patrick. "Investigating CAN Bus Network Integrity in Avionics Systems." Rapid7, 30 Jul. 2019, https://www.rapid7.com/research/report/investigating-can-bus-network-integrity-in-avionics-systems/

Appendix

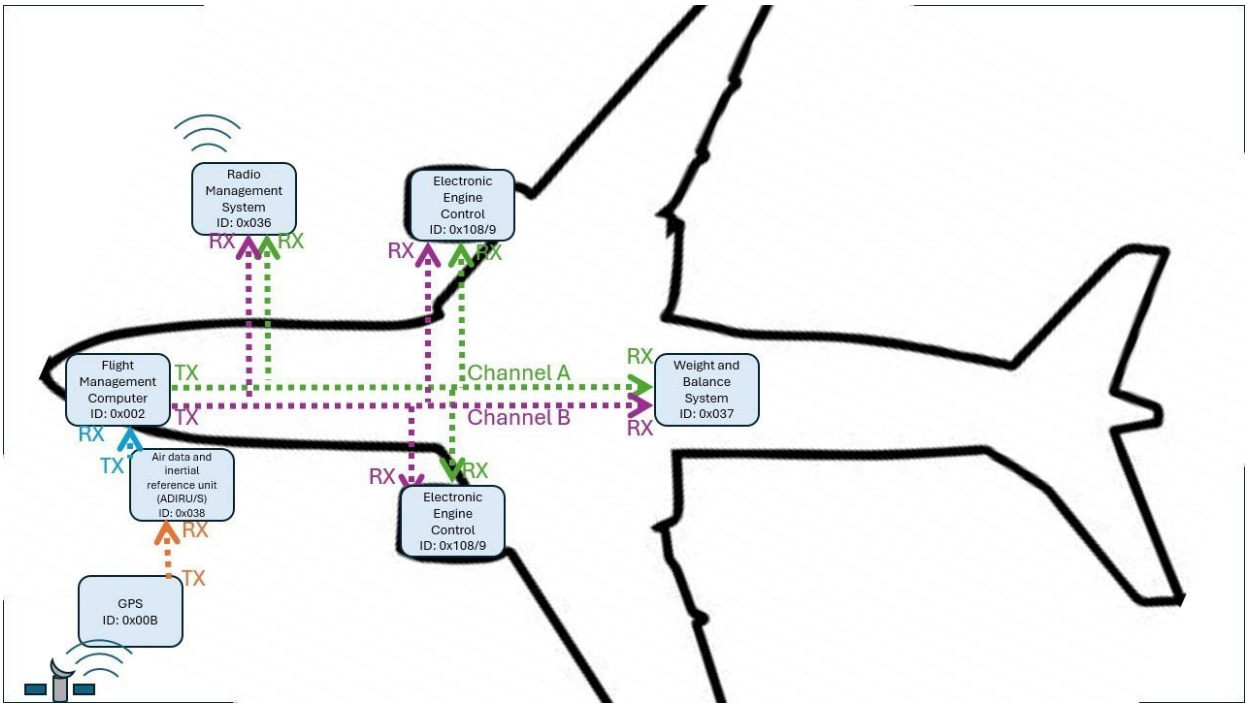


Figure 1: System Model

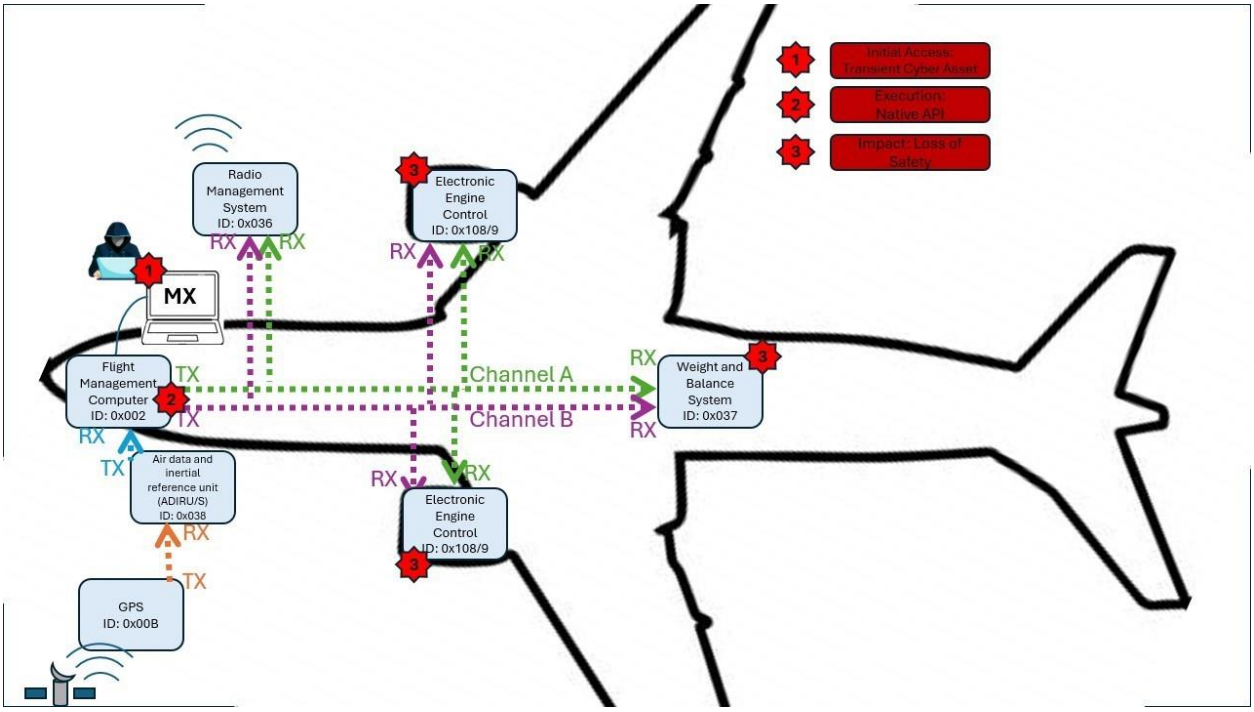


Figure 2: Threat Attack Model

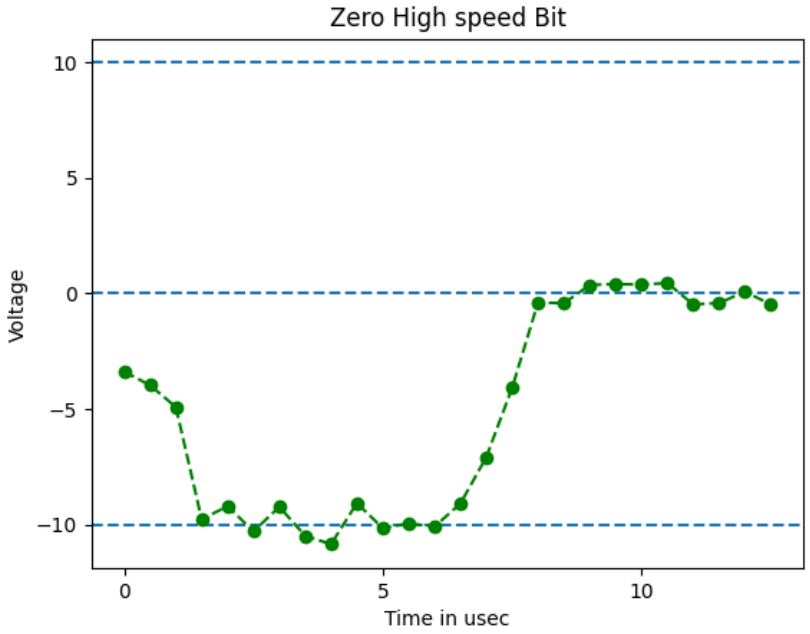


Figure 3: A set of voltages over time that represents a High Speed 0 bit.

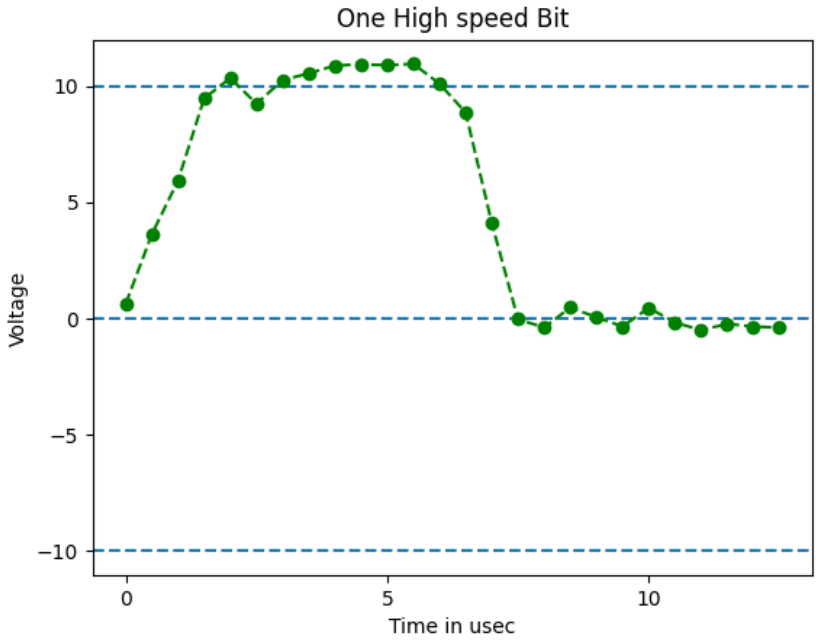


Figure 4: A set of voltages over time that represents a High Speed 1 bit.

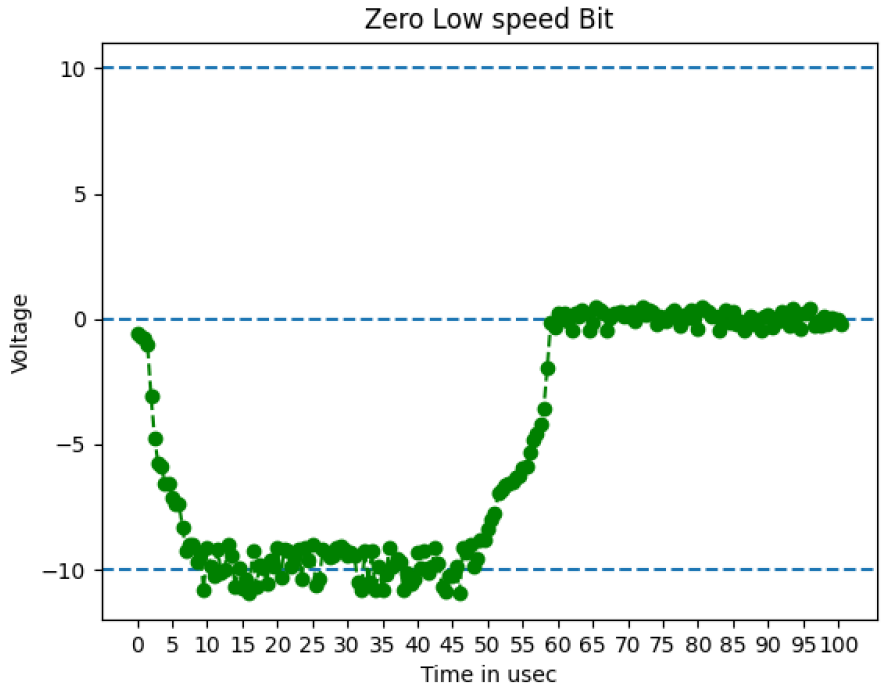


Figure 5: A set of voltages over time that represents a Low Speed 0 bit.

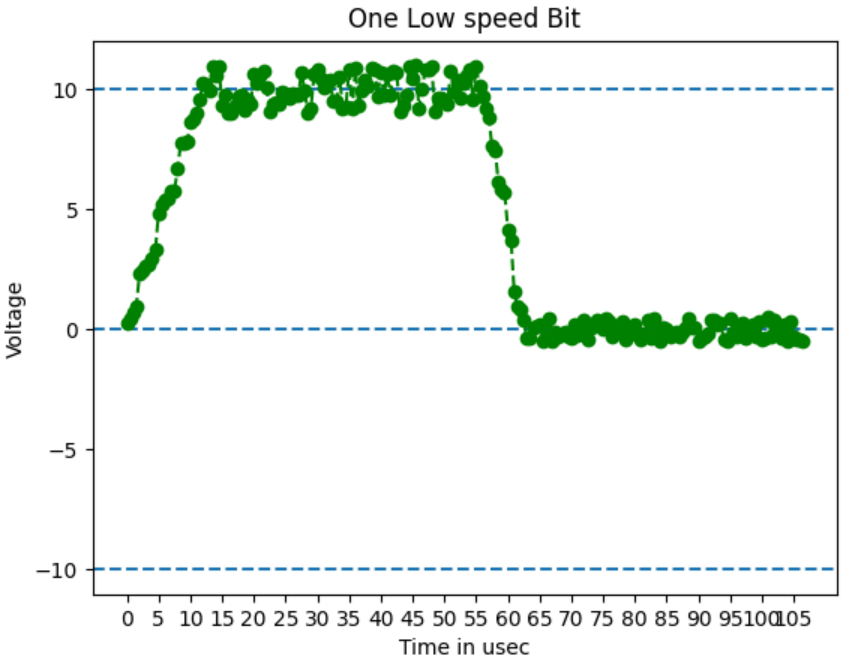


Figure 6: A set of voltages over time that represents a Low Speed 1 bit.

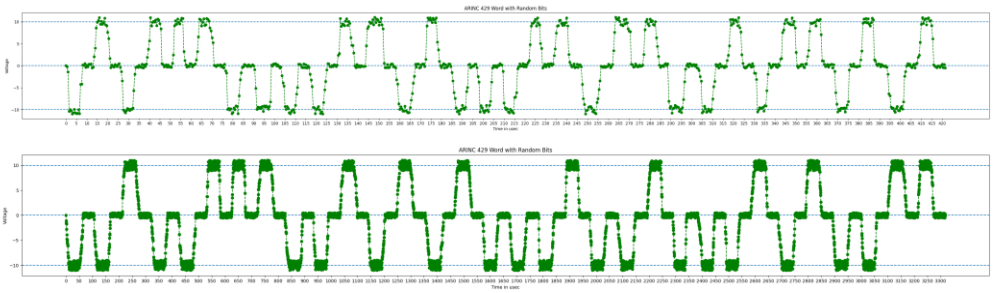


Figure 7 & 8: A set of voltages over times that represent a random word for respectively High and Low speed data buses.

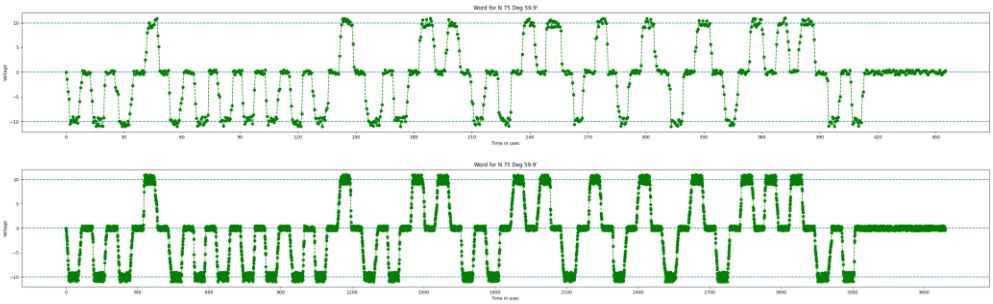


Figure 9 & 10: A set of voltages over time that represents a latitude information word at respectively High and Low Bus Speeds.

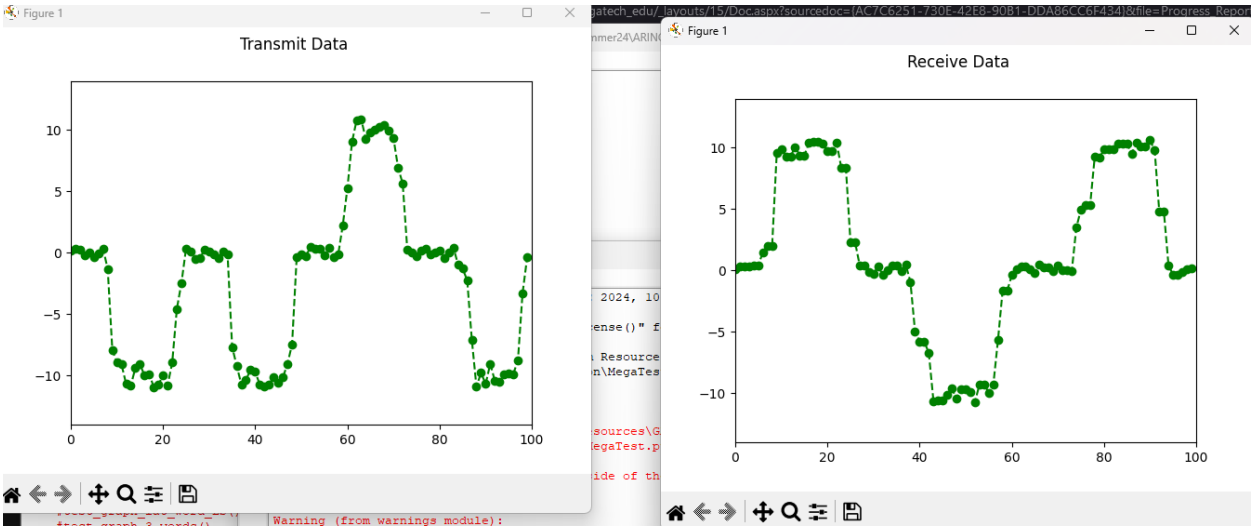


Figure 11: A screenshot of the bus TX and RX bus data over time. Notice the RX is delayed from the TX graph.