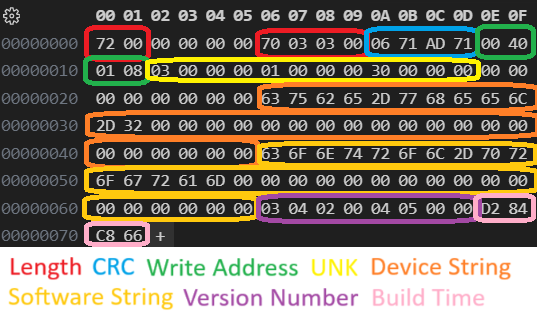
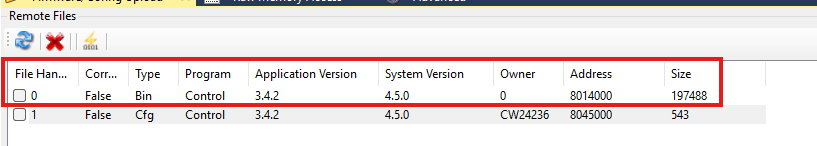
RE Engineering Process

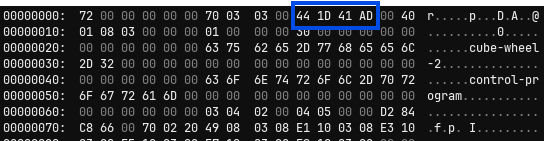
Uploading Modified Firmware

1. Using the PowerPoint provided by Tyler and the associated information included within, we are able to identify the various header information provided by the provided bootloader and firmware .cs files.

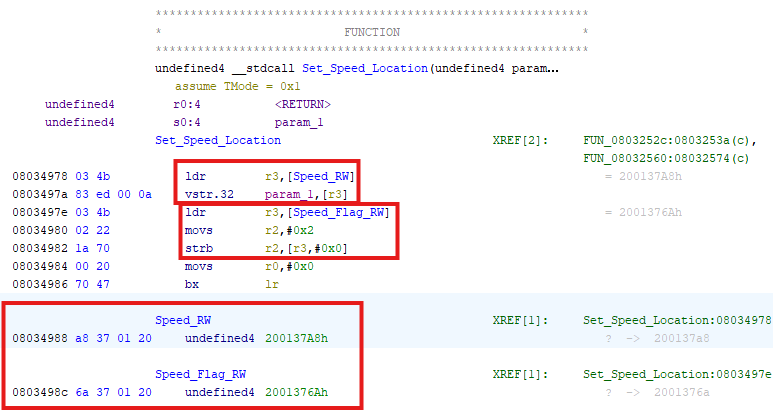


1. Through repeated upload attempts of modified firmware we were able to identify that the only check on the firmware is that CRC value. (Note it throws an error if the CRC does not match the CRC that is compiled by the STM32 chip)
2. Following this a modified firmware file along with modified CRC information in the header was successful to upload.

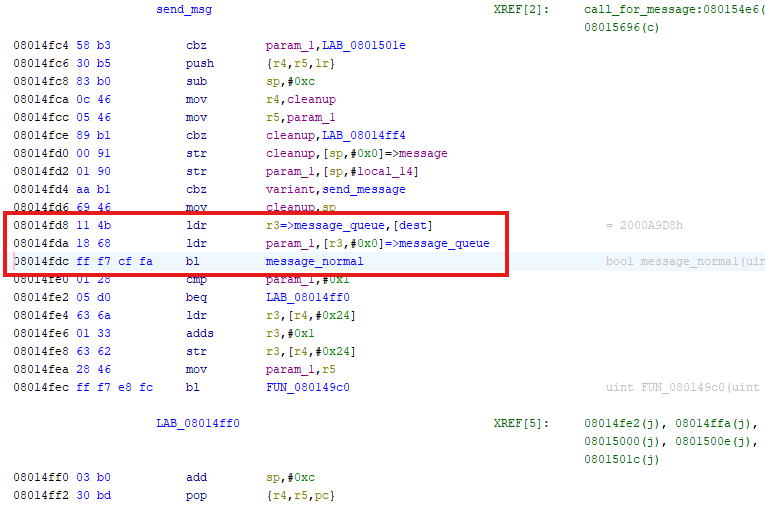


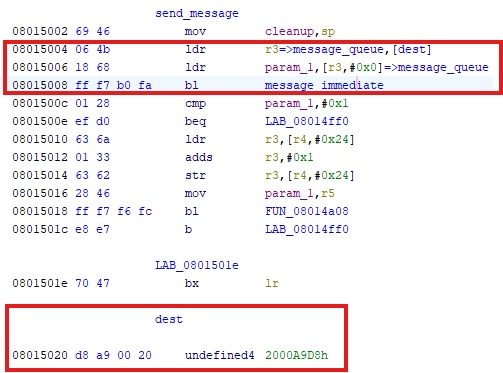


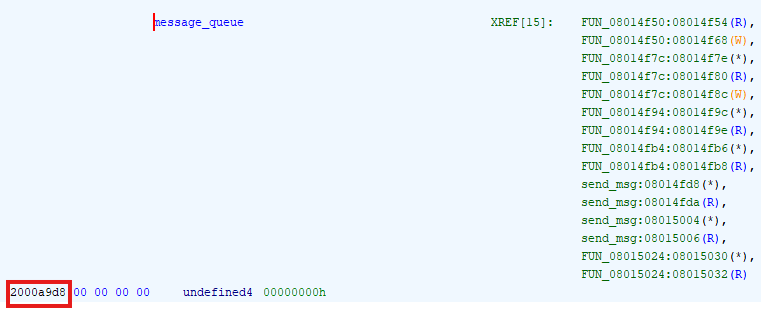
1. Through further investigation we have concluded that the program sets the RW speed at 0x200137a8 and sets a flag to 0x2 at 2001376a in order to change the RW speed.



1. We have also identified the send message function that takes the inputs: destination, message, and mode. The destination is the location the you input the message into the queue(0x2000a9d8). The message is the data or bytes you would like to send and the mode is the optional input that indicates whether you would like to send it immediately(0) or normally. This process seems to be too complex to insert our own message into without better decomplication or a much better understanding of the process.

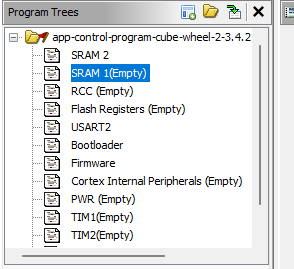






Identification Of Infiltration Point

1. The Firmware(app-control-program-cube-wheel-2-3.4.2-freertos-release-bin.cs) file was uploaded into Ghidra and decompiled using the arm-cortex 32 little endian decompiler.
2. The Bootloader(app-base-bootloader-52-cube-common-1-node-1.7-none-release-src-bin.cs) and SRAM2(0x10000000\_0x10007FFF.bin) were added to the program and resolved much of the decomplication issues with the original firmware.
3. Other areas of Memory were added to the memory map in order to resolve and trace references. The locations of these other spots in memory can be found in the STM32L452CEU3 documents(STM32\_doc in the Flexstat Directory) under Memory Mapping.



1. We then analyzed the decompiled instructions and found that the starting position was ambiguous and unclear. This made it difficult to be certain of what paths to trace and not all of the code was able to be reached by Ghidra.
2. From there multiple avenues of Reverse Engineering were traced via the decompiled ARM instructions. The section of the code that writes and reads from the USART instruction was unclear and didn’t lead to any relevant developments at this time.
3. From there we reviewed the instructions that are sent over USART to identify common or differing elements of the HEX commands. This was accomplished via a virtual COM port(created via Virtual Serial Ports Emulator) and a python that connects the COM port of the Reaction Wheel(RW) and the CubeSupport tool. The script passes the information from one COM port to another and allows us to read/write bytes to either COM port. The HEX commands were identified to contain the following format and options.

2 Bytes 1 Byte Variable Bytes 2 Bytes

[Start Flag] [Command Code] [Data] [End Flag]

Example Command: 1f7f4000401c461fff

List Of Known Command Codes:

40 :Change Speed (Speed number data is handled in IEEE 754 format)

41 :Duty Cycle

80 :Identification

80 :Refresh Home Telemetry, Get Info

81 :Serial Number

82 :Error Log Index

83 :Error Log Entry

84 :Error Log Settings

85 :Current Unix Time

86 :Persist Config Diagnostic

88 :Version

89 :Boot Status

8A :Telecommand Acknowledge

8B :Common Error Codes

8C :Identification 2

B7 :Wheel Position Data

B8 :Wheel Model

B9 :Wheel Torque

BA :Wheel Reference Torque

BB :Control Mode

BC :Wheel Speed

BD :Health Telemetry

BE :Wheel Data

BF :PWM Gain

C0 :Backup Gain

C1 :Main Gain

C2 :Status Error Flags

C3 :Wheel Commanded Duty Cycle

C4 :Wheel Reference Speed

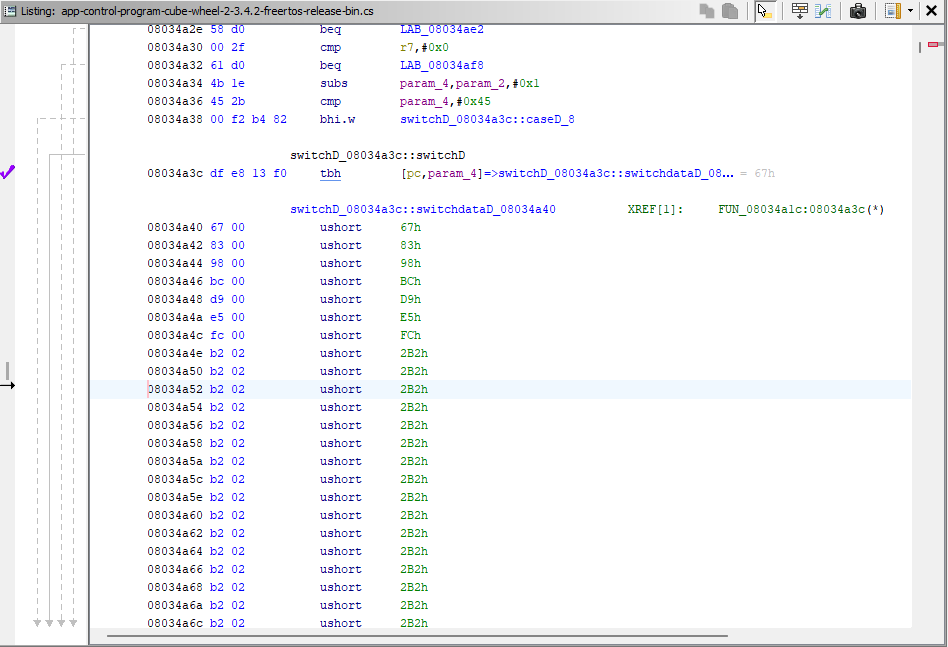
C5 :Motor Power Switch

C6 :Backup Wheel Codes

C7 :Stator Data

C8 :Wheel Reference Speed Ramp Rate Limit

1. With this information we inferred that the commands come in ascending/descending Hex format (i.e. 0x01 0x02 … 0x09 0x0A etc…)
2. Looking for scalars with the more unique Command Codes lead us to a Switch statement located at memory address 0x08034a3c that parses the command codes and directs them to their relevant functions.



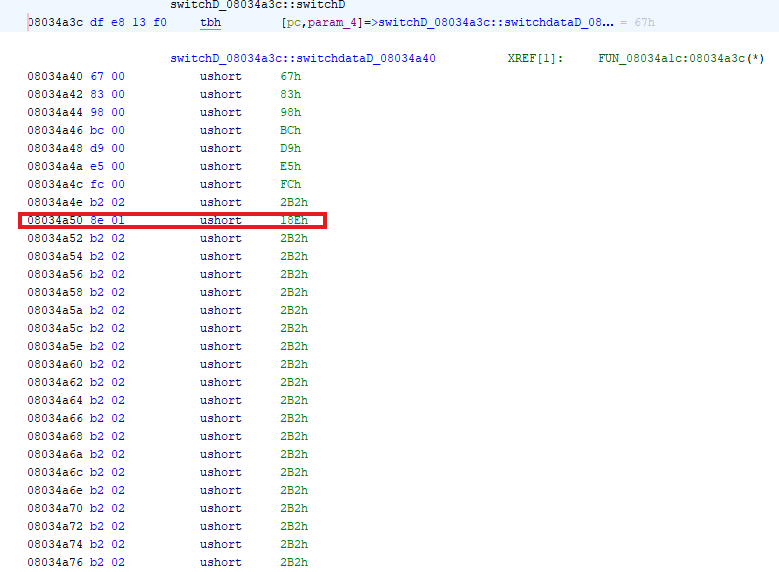
1. This switch statement uses an array of jump distances from index 0 [pc] to the designated function. To find the location you perform the following operations:

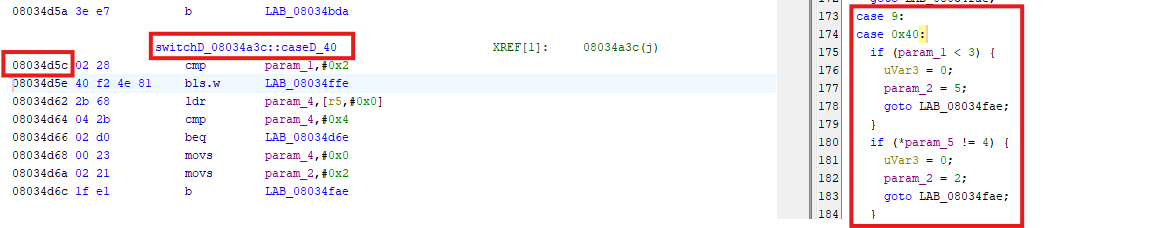
([Goal Address] – 0x08034a40)/2

And the index is found via:

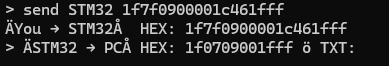
[Command Code] – 1

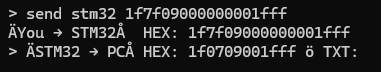
1. Using this logic we were able to create a custom command at index 09 that allows us to edit the RW speed by sending it to the Set Speed command function(Code: 40).

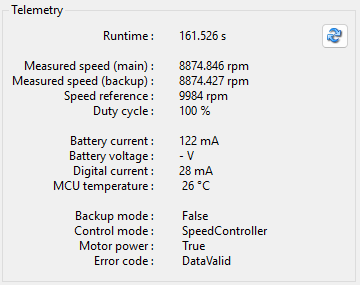




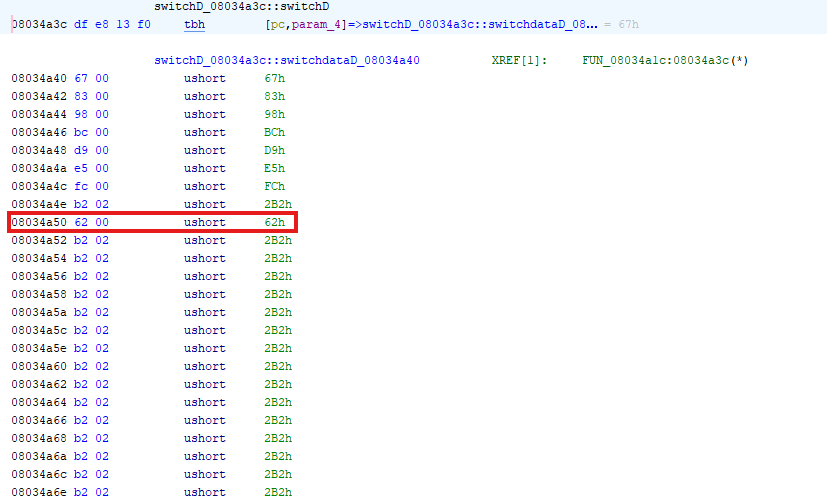
1. After this change we were able to successfully send a command using the virtual COM port that utilized 09 to change the speed rather than the 40 code.

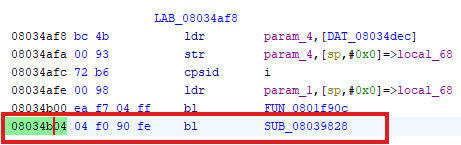


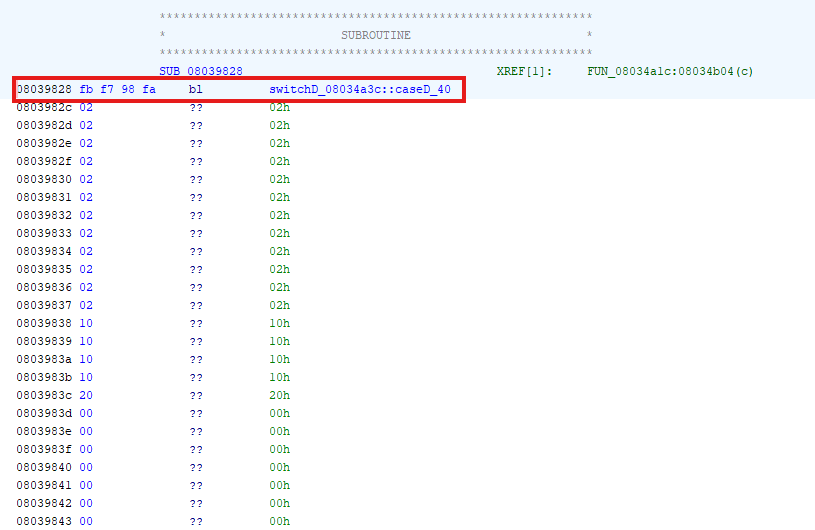




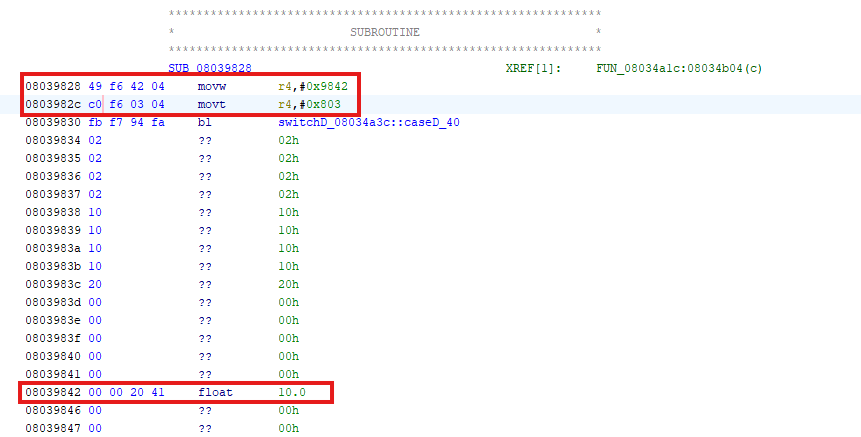
1. Using this switch statement and the associated logic we should be able insert a Command Code that jumps to our own function/malware that we’ve imbedded into the code.
2. The end of the Firmware file is too far from the switch statement so we will commandeer a debug breakpoint in order to jump to the end of file.



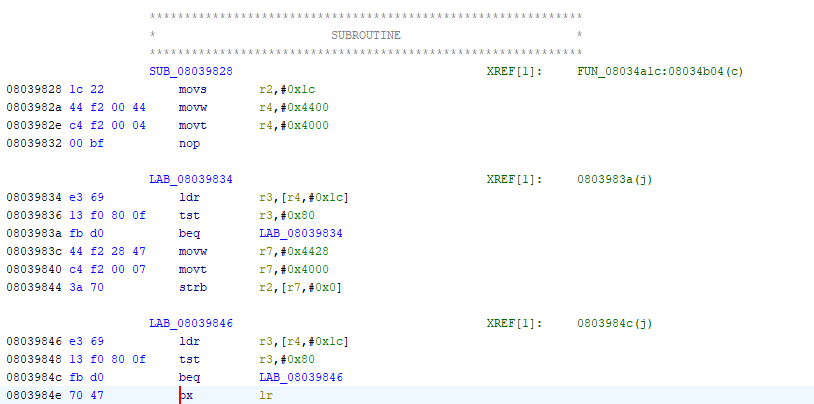


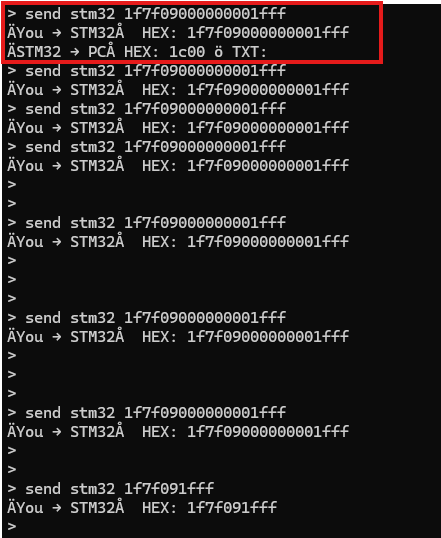


1. Continuing this train in logic we can override the registers in the Set Speed Function (Case 40) with our own values located in the subroutine. This can potentially be used for our own “Command Code”" that permanently affect the set speed code.

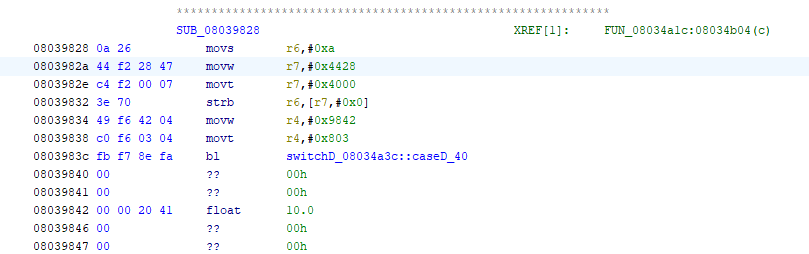


1. After locating the communication protocol (USART 2) at 0x40004400 and writing to offset 0x28 when offset 0x1c is not set to 1 we are able to get a response from their communication protocol then lock the USART communication method via corrupting the data by writing to it in an unsafe manner.

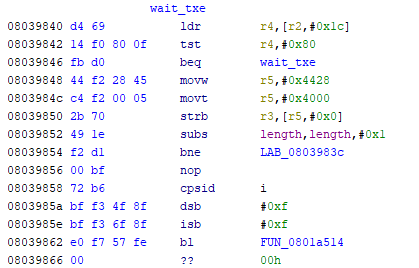


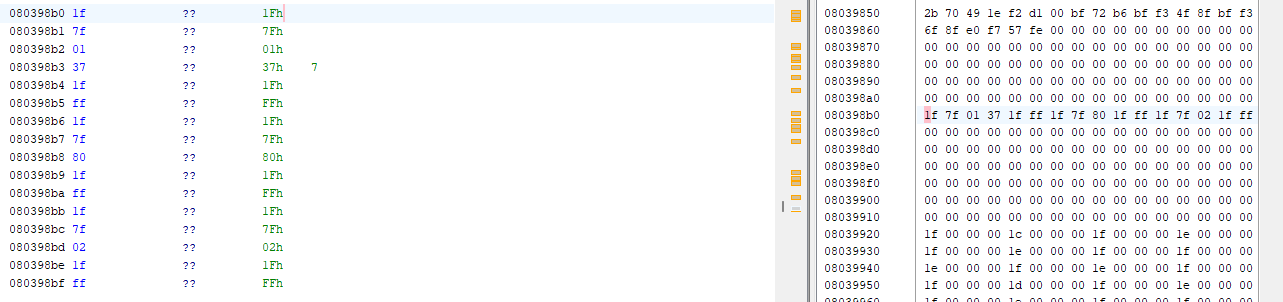


1. Combining the set speed functionality and the USART communication lock we are able to permanently set the speed to a given RPM and then deny the users ability to change it until the device is powered off and on.



1. Further more since communication with the RW in only done with a standard set of bytes sent over communication interfaces then we can use our device to control other RW’s attached to the same interface (CAN Bus/USART). We have built an example of this to send another reaction wheel into the bootloader but this could potentially be used for replacing another devices firmware with our own due to the excess space on the chip and the standardized byte communication method.





Sources

Flexstat Documentation

<https://www.st.com/resource/en/reference_manual/rm0394-stm32l41xxx42xxx43xxx44xxx45xxx46xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf>

<https://www.st.com/resource/en/datasheet/stm32l452cc.pdf>