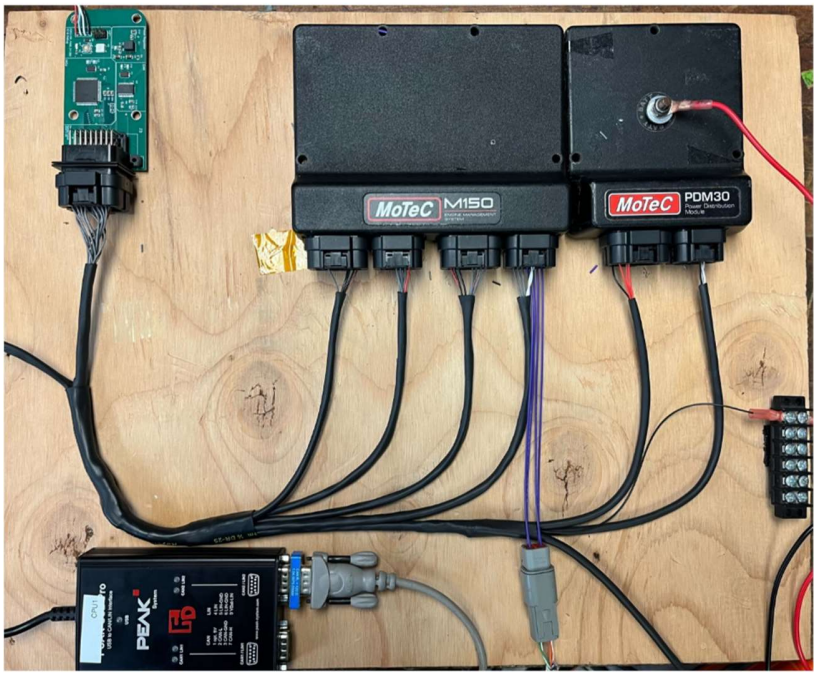
Panther Racing - PR037

Firmware – Automated Testing Platform

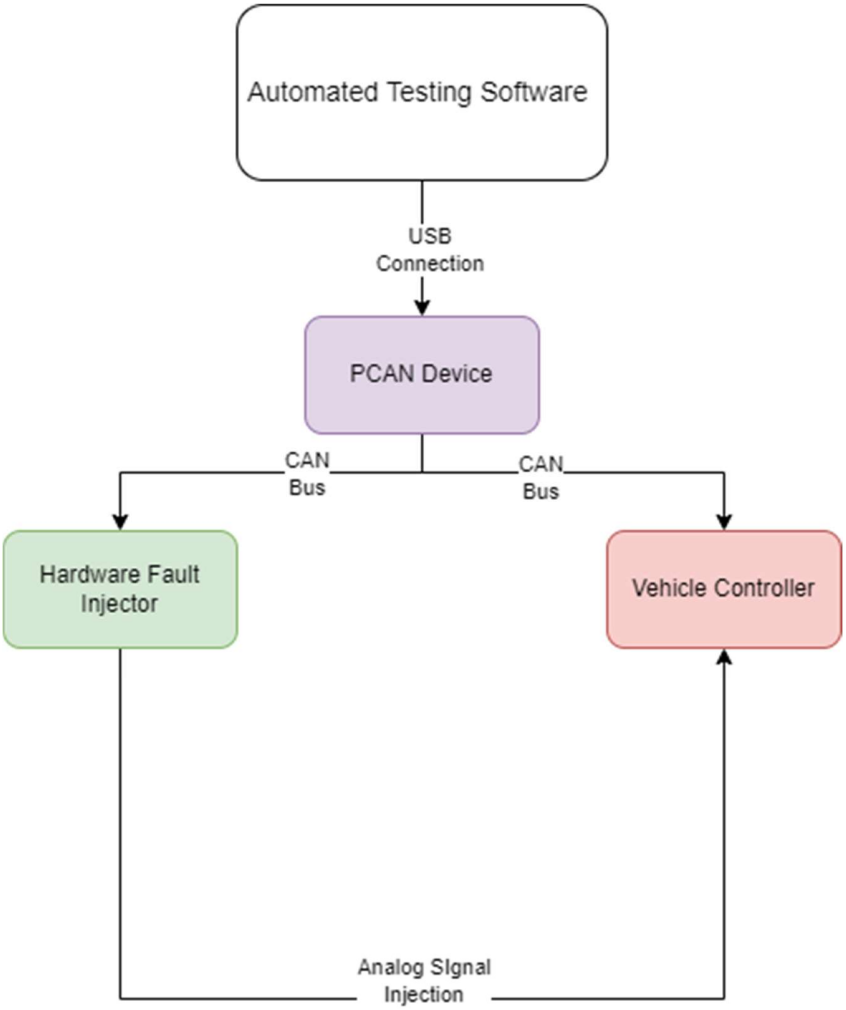
Automated Testing Platform Overview

The Automated Hardware In Loop Testing platform was developed in order to simplify and speed up unit and regression testing of the FSAE EV Vehicle Controller. The firmware that is written for the vehicle controller must be thoroughly tested through all normal and abnormal test cases every time a new firmware change is made. This testing is to ensure rules compliance and safety compliance and can take a lot of time and effort to do every time there is a change to firmware. Therefore, in order to speed up the process the Automated Testing Platform was created.

The platform has two portions; the Hardware Fault Injector Module(HFI) which allows us to simulate various sensors on the car and the software portion which handles sending CAN messages, running PyTest scripts, simulating other software aspects of the car and also controlling the hardware fault injector module. We also used a Peak Systems PCAN USB along with the software to be able to inject CAN messages into the CAN bus for testing.



High Level Diagram



The above diagram is a high level overview of the entire system. The automated testing software will be the main controller of the entire testing system. It then communicates to the PCAN which has two main functions; it allows the software to be able to inject CAN messages into the vehicle controller's CAN bus and it also allows us to communicate with the Hardware Fault Injector to allow us to inject analog signals into the vehicle controller. The Hardware Fault Injector communicates with the software over CAN.

Explanation of a Sample Test Case

Here is a sample test cases that can be written on the Automated Testing Platform and an explanation of how it works.

```
def test_accu_pressure_sensors():
    from data.message_data import MessageData
    hfi_dbc = os.path.join(os.getcwd(), 'Software\DBC Tools\HardwareInjector.dbc')
    hfi_db = cantools.database.load_file(hfi_dbc)
    vcu_torque_dbc = os.path.join(os.getcwd(), 'Software\DBC Tools\M150_VCU_TORQUE.dbc')
    vcu_torque_db = cantools.database.load_file(vcu_torque_dbc)
```

The first thing you must do in order to write a test case is make a function that starts with the words ”test” in the test_regrassion.py file. Then we import the required libraries, in this case MessageData. Afterwards we need to open the DBC file and load it into the database(db) object which can be seen on the last 4 lines in the image above.

```
# Create the HFI messages
hfi_command_mux0 = MessageData(message=hfi_db.get_message_by_name("HARDINJ_command"),
                               data={
                                   'HARDINJ_mux': 0,
                                   'HARDINJ_output1Control': 127,
                                   'HARDINJ_output2Control': 127,
                                   'HARDINJ_output3Control': 127,
                                   'HARDINJ_output4Control': 200,
                                   'HARDINJ_output5Control': 200,
                                   'HARDINJ_output6Control': 200,
                               })
hfi_command_mux1 = MessageData(message=hfi_db.get_message_by_name("HARDINJ_command"),
                               data={
                                   'HARDINJ_mux': 1,
                                   'HARDINJ_output7Control': 127,
                                   'HARDINJ_output8Control': 127,
                                   'HARDINJ_output9Control': 127,
                                   'HARDINJ_output10Control': 200,
                                   'HARDINJ_output11Control': 200,
                                   'HARDINJ_output12Control': 200,
                               })
```

If we are injecting analog signals to the vehicle controller then we must create te CAN messages to send to the Hardware Fault Injector. The values that can be sent right now are 8 bit values that will translate to a 0-5V signal. In the future this will be edited to allow for the voltage to be directly typed in rather than the 8 bit value.

```
# create and start CAN interface
can_interface = CANInterface()
can_interface.start_send_and_update_100Hz()
# Add and send HFI messages
can_interface.add_message_100Hz(hfi_command_mux0)
can_interface.add_message_100Hz(hfi_command_mux1)

# Now we wait for a period of time for the systems to catch up
time.sleep(1)
```

We then create the CANInterface object, which will connect to the PCAN and setup all the internal variables to allow the system to send and receive CAN messages. Afterwards we start the send and update thread that sends CAN messages and then we add the messages to the send list so it can be sent on the CAN bus. Finally, we wait a small amount of time for everything to initialize.

```
# Now we test and compare
can_interface.start_receive_and_sort(can_db=vcu_torque_db, timeout=2)
time.sleep(1) #wait for systems to respond
value = can_interface.get_signal_from_dictionary('VCU_accuCoolantPressOut')
```

Now we start the receive and sort thread which receives CAN messages and stores the signals in a dictionary. We wait a small amount of time again to ensure that messages are flowing in and then we can retrieve the values of the signals that we want form the dictionary.

```
try:
    assert(475.0 <= value <= 525.0)
finally:
    print(value)
    can_interface.stop_receive_and_sort()
    can_interface.stop_send_and_update_100hz()
    del can_interface
```

Finally, we use the assert function to ensure that the values we receive are within our

expected range. The output of this assert will pass or fail the test. We then stop the threads running in the background and delete the object to end the test gracefully.

Software

CAN Communications

The CAN communication was done using a Peak System PCAN module. This module connects to the computer and is able to both send and receive CAN messages. Integration of the PCAN into the Automated Testing System was done using the python can library. In addition to the library, I wrote some wrappers around the library to make it easier to interface with the PCAN and send CAN messages, this can be seen later on in this document.

CAN Message Sending

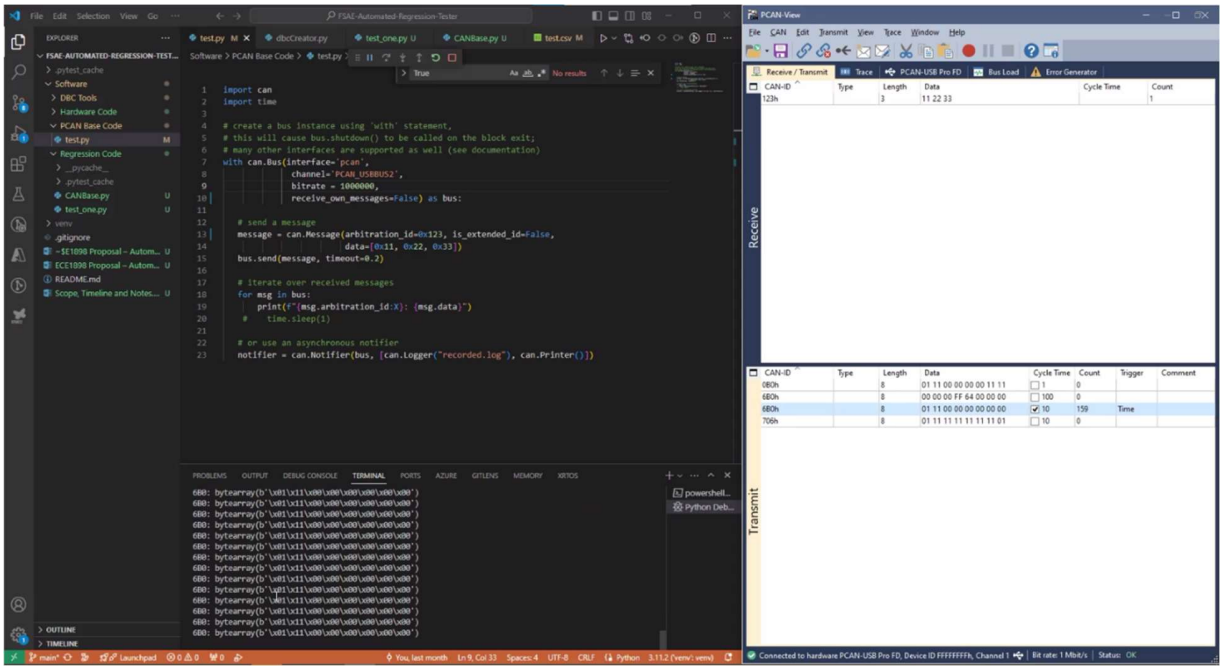
I went through many design revisions of code that was able to send CAN messages onto the bus, starting from a very simple program that could just send one CAN message to sending multiple messages concurrently without holding up the rest of the program. This module is what is used to be able to both control the Hardware Fault Injector and also simulate other software components of the car such as CAN expansion modules and steering wheel modules.

Design Revision 1

In order for this to be possible I started off with some basic python code to purely send CAN messages using the PCAN. In order to do this, I connected the two CAN channels on the PCAN together and read the message that I was sending on PCAN View, which is a GUI developed to manually send CAN messages and view the messages being received on the

CAN bus. With this initial revision code I also decided to write code to be able to read the CAN messages with python and print them out to the terminal.

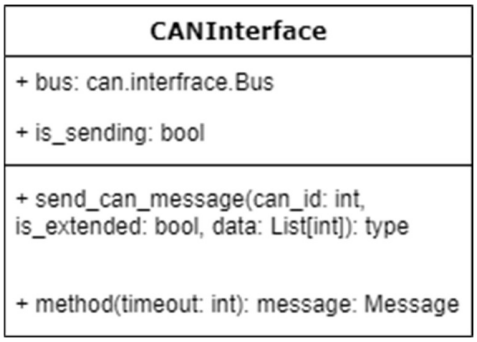
The 1st revision code created a CAN message object using the *can.Message()* function, in this case I used a random CAN ID and data frame just for testing purposes when making the message. I then sent the CAN message onto the bus using *can.Send().* For receiving CAN messages I used the *can.Notifier()* function which created a asynchronous thread that would wait for CAN messages to arrive and then print them out onto the terminal window.



Link to video

Design Revision 2

Once I had confirmed that I was able to send and receive CAN messages I went about writing a class for the CAN functionality so I could later integrate it easier into the larger Automated Regression Testing Software.



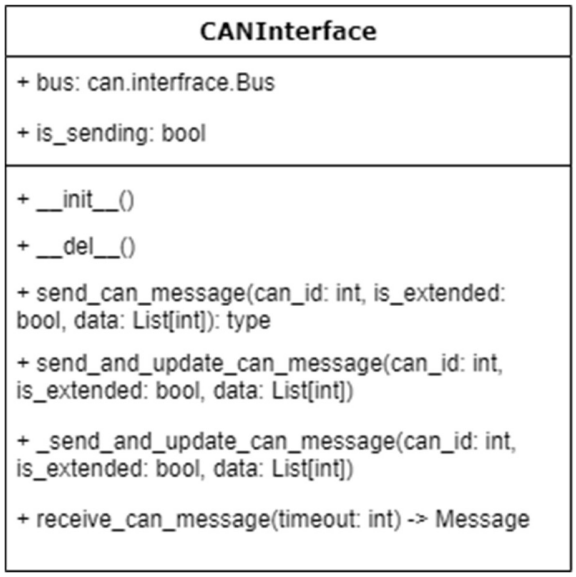
The above image is the initial CANInterface class diagram. It was very basic allowing us to create an object that holds the Bus interface that allows us to communicate with the PCAN hardware and two functions that could be called to send a CAN message and to receive the latest message on the CAN bus.

```python
import can


class CANInterface:
    def __init__(self):
        # create a bus instance using 'with' statement,
        # this will cause bus.shutdown() to be called on the block exit;
        # many other interfaces are supported as well (see documentation)
        self.bus = can.interface.Bus(interface='pcan',
                    channel='PCAN_USBBUS2',
                    bitrate = 1000000,
                    receive_own_messages=True)

    def __del__(self):
        self.bus.shutdown()

    def send_can_message(self, can_id, is_extended, data):
        # send a message
        self.message = can.Message(arbitration_id=can_id, is_extended_id=is_extended,
                                   data=data)
        self.bus.send(self.message, timeout=0.2)


    def receive_can_message(self, timeout=None):
        return self.bus.recv(timeout=timeout)

```

Initial CANInterface class code

Design Revision 3

While developing the basic automated testing code using PyTest I realized that that it would be much more user friendly for the end user to have a function they can call to send specific CAN messages continuously at a predetermined rate. I then went about implementing a function in the CANInterface class that would open a thread to be able to send a CAN message in the background without the user having to intervene.



The above image is the new CANInterface class diagram. There are two new functions in it; send_and_update_can_message and _ send_and_update_can_message. The second one is meant to be a private function that handles the actual sending on the can message and the timing between messages. The first one is meant to be the public function that handles starting the thread to allow the messages to be send in the background.

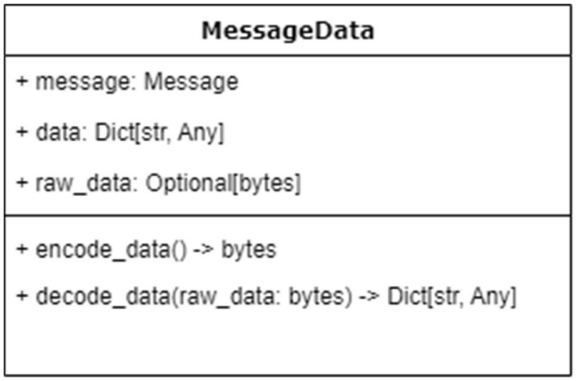
```python
def send_and_update_can_message(self, can_id, is_extended, data):
    '''
    Start sending CAN messages at a constant rate

    :param can_id: CAN message ID
    :param is_extended: Is the CAN ID using extended messaging or not
    :param data: The payload of the CAN message
    :param rate: CAN message send rate in Hz
    '''

    if not self.is_sending:
        self.is_sending = True
        self.send_and_update_thread = threading.Thread(target=self._send_and_update_can_message, args=(can_id, is_extended, data))
        self.send_and_update_thread.start()

def _send_and_update_can_message(self, can_id, is_extended, data):
    while self.is_sending:
        self.message = can.Message(arbitration_id=can_id, is_extended_id=is_extended,
                                   data=data)
        print("AUTOSEND")
        try:
            self.bus.send(self.message)
        except Exception as e:
            logging.error(f"Send and Update Error: {e}")

        time.sleep(0.01)
```

CAN message send code to continuously send messages

Design Revision 4

I then went about implementing further functionality in order to be able to send different CAN messages at a predetermined rate automatically. This was to improve user experience by allowing them to add the message to a list and then calling a function that would start a thread to send the messages. The functions use a thread lock in order to be able to write and read to a common list that holds the CAN message objects.


CANInterface
+ bus: can.interfrace.Bus
+ is_sending: bool
+ message_list_100Hz: List[MessageData]
+ message_list_100Hz_lock: threading.Lock
+ message_list_100Hz_isRunning: bool
+ send_and_update_100hz_thread: threading.Thread
+ send_and_update_thread: threading.Thread
+ __init__()
+ __del__()
+ send_can_message(can_id: int, is_extended: bool, data: List[int])
+ start_send_and_update_100Hz()
+ stop_send_and_update_100hz()
+ add_message_100Hz(can_message: MessageData)
+ send_and_update_can_message(can_id: int, is_extended: bool, data: List[int])
+ receive_can_message(timeout: Optional[float])
+ _send_and_update_can_message(can_id: int, is_extended: bool, data: List[int])
+ _send_and_update_can_message_100Hz()


MessageData
+ message: Message
+ data: Dict[str, Any]
+ raw_data: Optional[bytes]
+ encode_data() -> bytes
+ decode_data(raw_data: bytes) -> Dict[str, Any]

The above images show the new CANInterface and MessageData class diagram. The MessageData class acts as a struct to hold the cantools Message object as well as data that corresponds to the Message object. This gives us an easy way to access the CAN Message object as well as the data for the message, thus making it more organized and easier to store both the Message information and the payload in in message_list_100Hz list. The message_list_100Hz is then read in the _send_and_update_can_message_100hz function that sends the CAN messages on the bus. We also have an add_message_100Hz function that allows the user to add a message to be sent repetitively on the bus.

```
'''
Internal send and update function
This will run in send_and_update_100hz_thread
@return: None
'''
def _send_and_update_can_message_100Hz(self):
    while self.message_list_100Hz_isRunning:
        with self.message_list_100Hz_lock:
            for message in self.message_list_100Hz:
                message_to_send = can.Message(arbitration_id=message.message.frame_id,
                                              is_extended_id=message.message.is_extended_frame,
                                              data=message.encode_data())
                try:
                    self.bus.send(message_to_send)
                except Exception as e:
                    logging.error(f"Send and Update Error: {e}")
                time.sleep(0.01)
```

CAN message send code to continuously send messages from message_list_100Hz

```
'''
Add 100Hz messages to the array to be sent

@can_message: the cantools Message object that you want to send
'''
def add_message_100Hz(self, can_message: MessageData):
    if self.message_list_100Hz_isRunning:
        with self.message_list_100Hz_lock:

            #check if message doesnt exist in the array, if it doesnt append. If it does replace the message
            if not any(message.message.frame_id == can_message.message.frame_id for message in self.message_list_100Hz):
                self.message_list_100Hz.append(can_message)
                print(f"Appended {can_message.message.name} to 100Hz List")
            else:
                for i in range (len(self.message_list_100Hz)):
                    if self.message_list_100Hz[i].message.frame_id == can_message.message.frame_id:
                        self.message_list_100Hz[i] = can_message
                        print(f"Updated {can_message.message.name} in 100Hz List")
                        break
```

Code to add MessageData objects to the message_list_100Hz

```
#data/message_data.py
from dataclasses import dataclass, field
from typing import Any, Dict, Optional
import cantools

@dataclass
class MessageData:
    message: cantools.db.Message
    data: Dict[str, Any] = field(default_factory=dict) #we use default_factiry=dict so that each object ahs a different dictionary and t
    raw_data: Optional[bytes] = None

    '''
    encode_data will encode the data that exists in the message object
    @return: raw_data
    '''
    def encode_data(self):
        self.raw_data = self.message.encode(self.data)
        return self.raw_data

    '''
    deode_data will decode the data that is passed in
    @raw_data: bytes of the raw data you want decoded
    @return: data
    '''
    def decode_data(self, raw_data: bytes):
        self.raw_data = raw_data
        self.data = self.message.decode(raw_data)
        return self.data
```

Code for the MessageData class, it holds the Message object and the data for the object

Design Revision 5

I further improved on the previous algorithm where I was able to send multiple CAN messages at a predefined rate. The major change was in the add_message_100Hz function where I was able to add support for multiplexed messages.

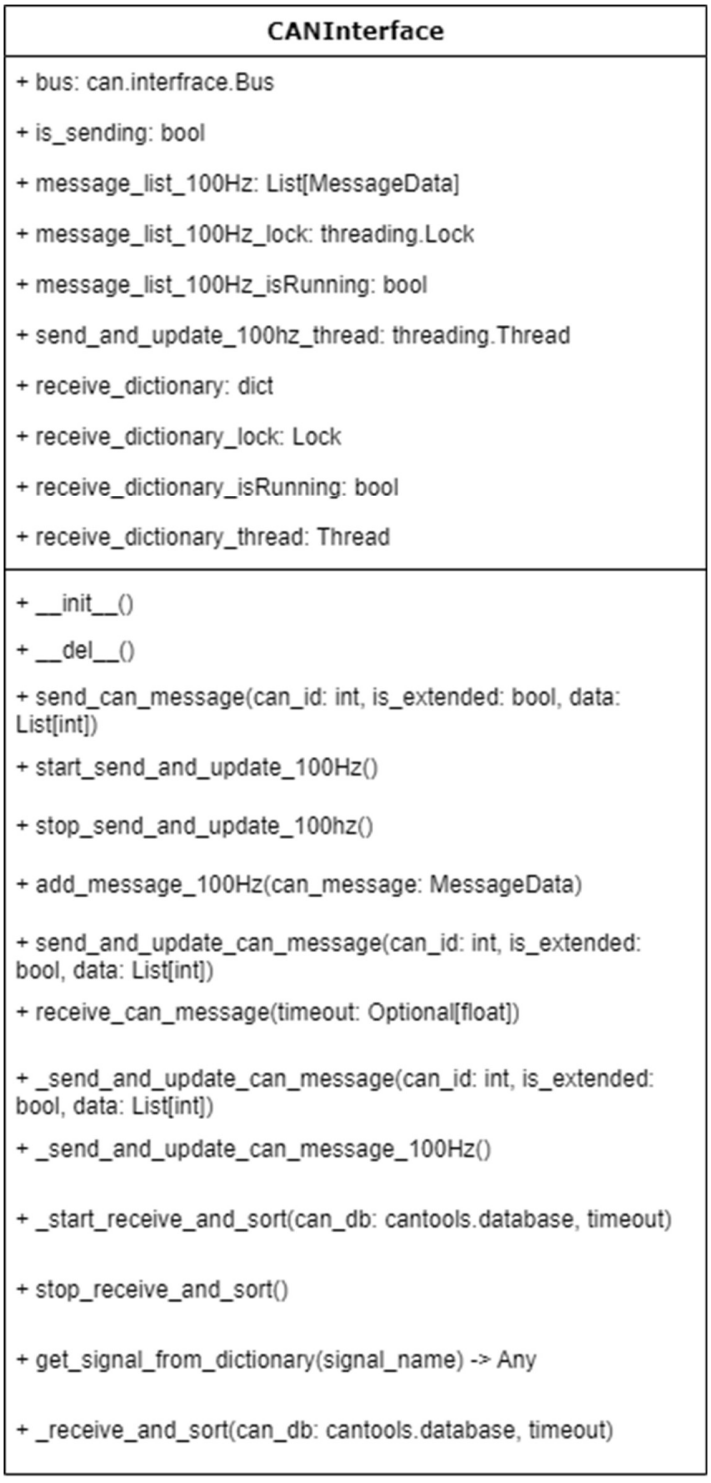
```
def add_message_100Hz(self, can_message: MessageData):
    mux_updated = False

    if self.message_list_100Hz_isRunning:
        with self.message_list_100Hz_lock:

            #check if message doesnt exist in the array, if it doesnt append. If it does replace the message
            if not any(message.message.frame_id == can_message.message.frame_id for message in self.message_list_100Hz):
                #if the message doesnt exist we dont need to worry about any mutiplexors so we just add the message
                self.message_list_100Hz.append(can_message)
                print(f"Appended {can_message.message.name} to 100Hz List")
            else:
                # we will handle muiplexed messages sepeartely from the standard ones
                # check if the message is multiplexed, if so we are going to look at whether the specific muxed message is already add
                if can_message.message.is_multiplexed():
                    # if multipled loop through and find the matching frame_id
                    for i in range (len(self.message_list_100Hz)):
                        if self.message_list_100Hz[i].message.frame_id == can_message.message.frame_id:
                            # if frame_id is found then we can look for the multiplexor signal and then see if the value of the mux
                            #   is the same as what we are trying to add. If it is we will replace that whole message
                            for signal in self.message_list_100Hz[i].message.signals:
                                if signal.is_multiplexer:
                                    if self.message_list_100Hz[i].data[signal.name] == can_message.data[signal.name]:
                                        print("Found Mux signal in message, updating message")
                                        self.message_list_100Hz[i] = can_message
                                        mux_updated = True

                    # if we didnt find the frame and/or mux ID in the array we will add it to it
                    if not mux_updated:
                        self.message_list_100Hz.append(can_message)
                        print("Mux signal not found in message, appending message")

                # If its a standard meessage and was found in the array then we just update the message here
                else:
                    for i in range (len(self.message_list_100Hz)):
                        if self.message_list_100Hz[i].message.frame_id == can_message.message.frame_id:
                            self.message_list_100Hz[i] = can_message
                            print(f"Updated {can_message.message.name} in 100Hz List")
                            break
```

Code for new add_message_100Hz function

The boxed section in the above code was added for multiplexed message functionality. How it works is by first checking if the message is multiplexed by looking at the properties of the message object. Then if it is, it will look to see if the multiplexor ID exists in the list. If it does, then it will update that message object in the list with the latest message object. If it doesn't then it will append that message object to the message list. This functionality is extremely vital to be able to fully utilize the benefits of the Automated testing Platform as many messages read and sent by the Vehicle Controller are multiplexed.

CAN Message Receive

The CAN message Receive code went through fewer design iterations than the send code. This was due to a couple of factors such as time and experience. However, experience had the bigger effect since majority of the receive code was written after the send code was written and therefore, I was able to reuse a lot of the code structure from the send code and used a very similar way to handle thread locking and data storage.



CANInterface class diagram with receive functions added

The above class diagram is the latest iteration of the CANInterface class, it has all the functions required to be able to asynchronously send and receive CAN messages.

```python
def _receive_and_sort(self, can_db:cantools.database, timeout):
    self.receive_dictionary = {}
    while self.receive_dictionary_isRunning:
        with self.receive_dictionary_lock:
            message = self.bus.recv(timeout=timeout)

            if message:
                try:
                    # Get the corresponding message object from the DBC
                    db_message = can_db.get_message_by_frame_id(message.arbitration_id)

                    # Decode the message data
                    decoded_data = db_message.decode(message.data)

                    # Check if the message is multiplexed
                    if db_message.is_multiplexed:
                        multiplexer_signal = None

                        for signal in db_message.signals:
                            if signal.is_multiplexer:
                                multiplexer_signal = signal.name
                                break

                        if multiplexer_signal:
                            # Get the current multiplexor value
                            multiplexer_value = decoded_data[multiplexer_signal]

                            # Extract only the signals for the current multiplexor value
                            # print(f"Message is multiplexed with {multiplexer_signal}={multiplexer_value}")
                            for signal in db_message.signals:
                                if signal.multiplexer_ids is None or multiplexer_value in signal.multiplexer_ids:
                                    self.receive_dictionary[signal.name] = decoded_data[signal.name]

                    else:
                        # For non-multiplexed messages, store all signals
                        for signal_name, signal_value in decoded_data.items():
                            self.receive_dictionary[signal_name] = signal_value

                    # print(f"Updated signal dictionary: {self.receive_dictionary}")

                except KeyError:
                    # print(f"Message with ID {hex(message.arbitration_id)} not found in DBC.")
                    pass
```

_receive_and_sort Function for Receiving CAN Messages

The above image is the code responsible for receiving CAN messages. This was something I put a lot of thought into before it was implemented. I wanted a way to be able to extract CAN messages real time and also allow the user to be able to get this data and interact with it easily. Therefore, I created a dictionary that would hold the data so that the user can access it. Now just being able to access data by using the CAN ID of the message is convenient to the user but it is not as helpful as typing out the name of the signal that they are looking to access. I then used the cantools library to be able to decode the CAN data that is coming in, which allowed me to store the data in a dictionary with the signal data

name and the value that it holds. This made it much easier to retrieve data and can be seen in the image below.

```
value = can_interface.get_signal_from_dictionary('VCU_accuCoolantPressOut')
```

As seen in the image above, only one line is required to retrieve data, and the user just needs to type in the name of the signal they wish to receive.

Automated DBC Creation Tool

In order to further simplify and speed up the process of making DBC files to use with the Automated Testing Platform, I took the time to write some code that will parse through our CSV files that hold the DBC message structure and automatically make a DBC file from it. To use it you just need to run the dbcCreator.py file inside of the DBC Tools folder(FSAE-Automated-Regression-Tester\Software\DBC Tools).

Automated DBC Creation Tool Explanation

```
# Read the CSV file
with open('test.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)

    messages = {}

    for row in reader:
        message_id = int(row['Message ID'], 16)
        message_name = row['Message Name']
        message_length = int(row['Message Length'])

        # Create or get the message
        if message_id not in messages:
            message = Message(
                frame_id=message_id,
                name=message_name,
                length=message_length,
                signals=[],
                cycle_time=100
            )
            messages[message_id] = message
            db.messages.append(message)
        else:
            message = messages[message_id]
```

The tool first opens the CSV file and the reads in the message information using the header information. It then looks to see if the message already exists in the list, if it does not it will append to the list, if it does then it will extract the message object from the list.

```
# Function to convert a CSV row to a Signal object
def csv_row_to_signal(row):
    name = row['Signal Name']
    start = int(row['Start Bit'])
    length = int(row['Length'])
    byte_order = 'little_endian' if row['Byte Order'].lower() == 'little' else 'big_endian'
    is_signed = row['Signed'].lower() == 'true'
    factor = float(row['Scale'])
    offset = float(row['Offset'])
    minimum = float(row['Minimum']) if row['Minimum'] else None
    maximum = float(row['Maximum']) if row['Maximum'] else None
    unit = row['Unit']
    muxSignal = row['Multiplexer Signal']
    muxID = row['Multiplex ID']
    isMux = row['Is Multiplexor'].lower() == 'true'
    print(isMux)
    print(muxSignal)
    conversion = BaseConversion.factory(scale=factor, offset=offset)

    if muxID=='':
        muxID=None
    if muxSignal=='':
        muxSignal=None


    return Signal(
        name=name,
        start=(start+((length-1)%8)) if row['Byte Order'].lower() == 'big' else start,
        length=length,
        byte_order=byte_order,
        is_signed=is_signed,
        conversion=conversion,
        minimum=minimum,
        maximum=maximum,
        unit=unit,
        multiplexer_ids=muxID,
        multiplexer_signal=muxSignal,
        is_multiplexer=isMux
    )
```

The software then runs the csv_row_to_signal function that will extract the signal data from the CSV file. This data is then use to create the Signal object that is returned from the function.

```
# Convert CSV row to Signal and add to the message
signal = csv_row_to_signal(row)
message.signals.append(signal)
```

We then take the signal object and add it to the message object that we previously extracted from the list that holds all the message objects. This is then written to a DBC file.

Software > DBC Tools > test.csv > data

```
Message ID,Message Name,Message Length,Signal Name,Is Multiplexor,Multiplex ID,Multiplexer Signal,Start Bit,Length,Byte Order,Signed,Scale,
0x001,HARDINJ_command,8,HARDINJ_mux,TRUE,,,0,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Mux,
0x001,HARDINJ_command,8,HARDINJ_output1Control,FALSE,0,HARDINJ_mux,8,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 1 Control,
0x001,HARDINJ_command,8,HARDINJ_output2Control,FALSE,0,HARDINJ_mux,16,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 2 Control,
0x001,HARDINJ_command,8,HARDINJ_output3Control,FALSE,0,HARDINJ_mux,24,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 3 Control,
0x001,HARDINJ_command,8,HARDINJ_output4Control,FALSE,0,HARDINJ_mux,32,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 4 Control,
0x001,HARDINJ_command,8,HARDINJ_output5Control,FALSE,0,HARDINJ_mux,40,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 5 Control,
0x001,HARDINJ_command,8,HARDINJ_output6Control,FALSE,0,HARDINJ_mux,48,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 6 Control,
0x001,HARDINJ_command,8,HARDINJ_output7Control,FALSE,1,HARDINJ_mux,8,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 7 Control,
0x001,HARDINJ_command,8,HARDINJ_output8Control,FALSE,1,HARDINJ_mux,16,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 8 Control,
0x001,HARDINJ_command,8,HARDINJ_output9Control,FALSE,1,HARDINJ_mux,24,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 9 Control,
0x001,HARDINJ_command,8,HARDINJ_output10Control,FALSE,1,HARDINJ_mux,32,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 10 Control,
0x001,HARDINJ_command,8,HARDINJ_output11Control,FALSE,1,HARDINJ_mux,40,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 11 Control,
0x001,HARDINJ_command,8,HARDINJ_output12Control,FALSE,1,HARDINJ_mux,48,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 12 Control,
0x001,HARDINJ_command,8,HARDINJ_output13Control,FALSE,2,HARDINJ_mux,8,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 13 Control,
0x001,HARDINJ_command,8,HARDINJ_output14Control,FALSE,2,HARDINJ_mux,16,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 14 Control,
0x001,HARDINJ_command,8,HARDINJ_output15Control,FALSE,2,HARDINJ_mux,24,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 15 Control,
0x001,HARDINJ_command,8,HARDINJ_output16Control,FALSE,2,HARDINJ_mux,32,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 16 Control,
0x001,HARDINJ_command,8,HARDINJ_output17Control,FALSE,2,HARDINJ_mux,40,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 17 Control,
0x001,HARDINJ_command,8,HARDINJ_output18Control,FALSE,2,HARDINJ_mux,48,8,Big,FALSE,1,0,0,255,,Hardware Signal Injector,Output 18 Control,
```

CSV file for DBC Creation Tool

Software Unit Testing

In order to ensure that the classes and methods that I wrote were functioning as it should, I also went about writing test cases for them as i built the program. This approach allowed me to isolate and test each function out before integration, thus making the process much smoother and easier.

All of these test cases for the software are in a test folder in the directory ***FSAE-Automated-Regression-Tester\Software\Regression Code\test***

Analog Fault Injection Hardware

The device is meant to simulate analog sensors and inputs to the Vehicle Controller, it works by receiving a CAN message from the regression suit and then uses the data in the message to set a PWM output on a specific pin.

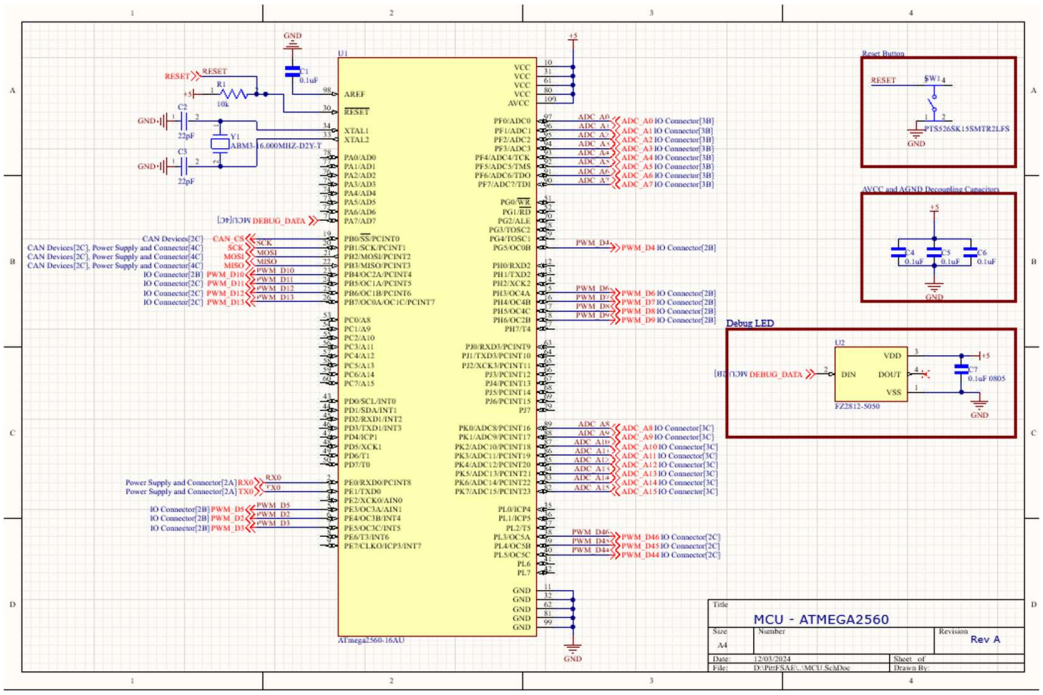
Design Concept

The initial design concept for the Analog Fault Injection Hardware included using an Arduino UNO to simulate sensors that would be connected to the Vehicle Controller. The hardware would take in a UART signal from the computer running the Automated Regression Testing Suite and use the PWM output pins on the ATMEGA328P to send an analog voltage to the sensor input pins on the vehicle controller. The diagram below shows a high-level overview of the system.

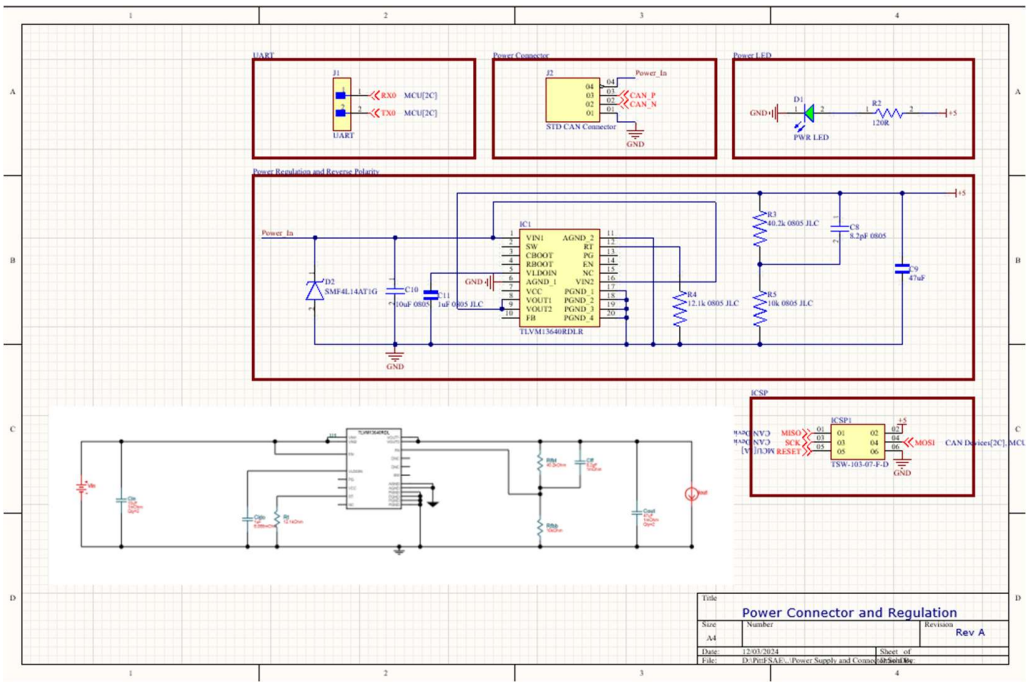
Fault Injection Hardware - RevA

Fault Injection Hardware Schematic – RevA

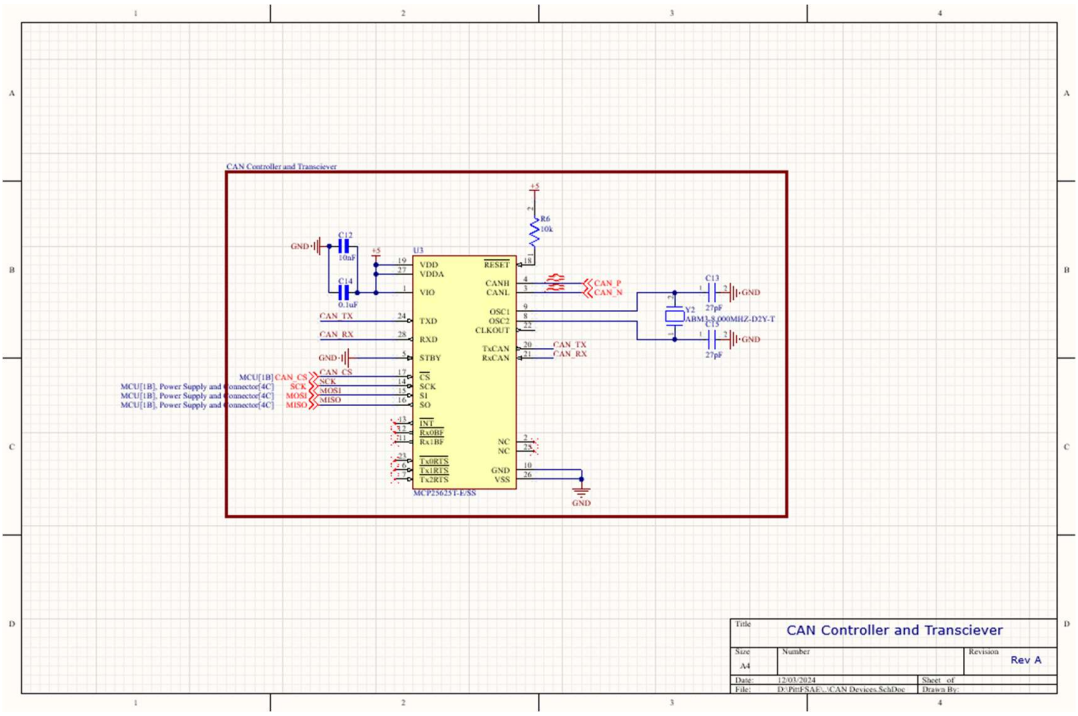
The following images show the schematic of the Fault Injector Hardware.



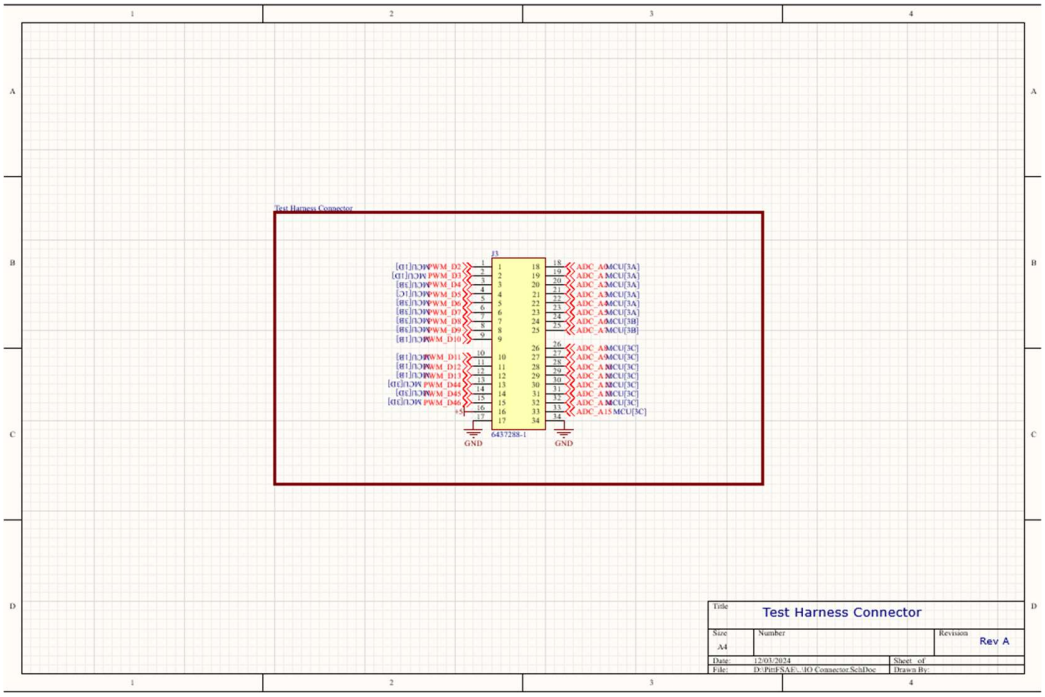
MCU Schematic



Power Connectors and Power Regulator Schematic



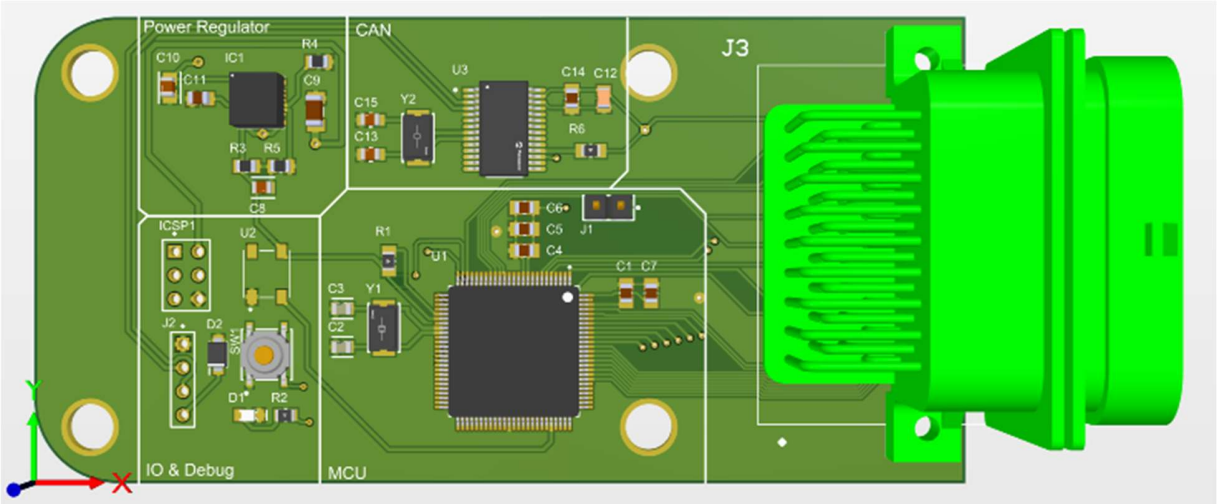
CAN Controller and Transceiver Schematic



Test Harness Connector Schematic

Fault Injection Hardware PCB – RevA

This is the RevA PCB design for the hardware fault injector. It is designed as a 4 layer PCB since there were a lot of connections from the ATMEGA2560 chip and i wanted to keep signal interference as low as possible. The main connector that was selected is the same model that is used on the vehicle control unit that this board is meant to connect to. The reason this connector was picked is so that a single set of tools can be used to manufacture a harness that allows the hardware injector to be connected to the vehcile controller.

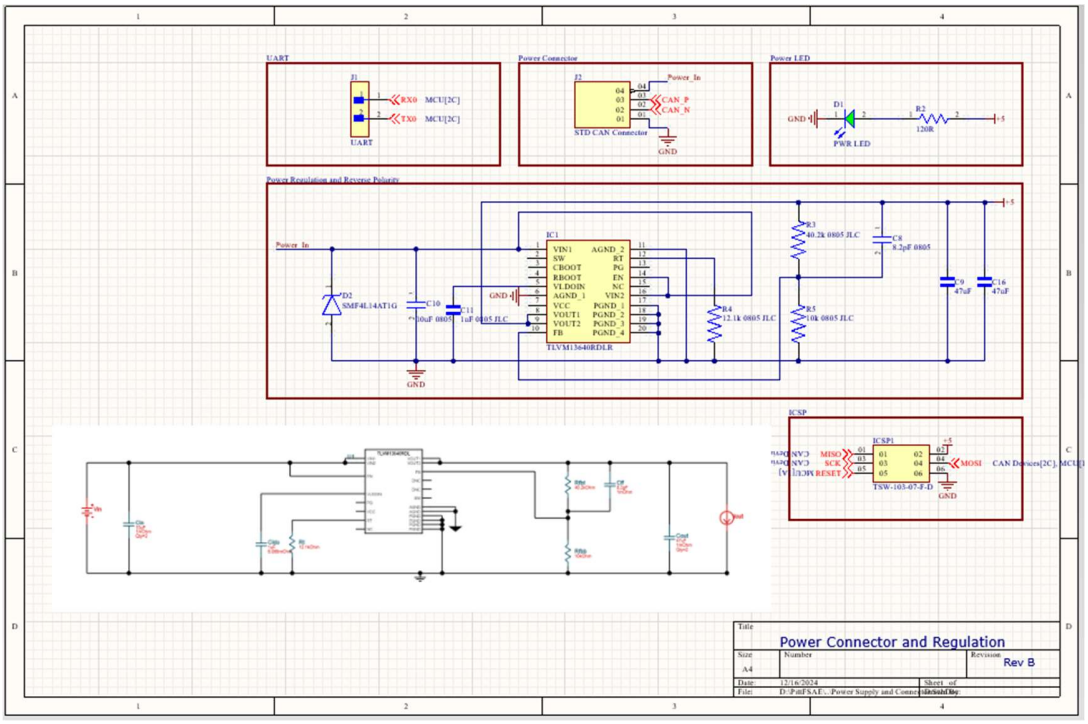


Hardware Fault Injector PCB RevA

Fault Injection Hardware - RevB

Fault Injection Hardware Schematic – RevB

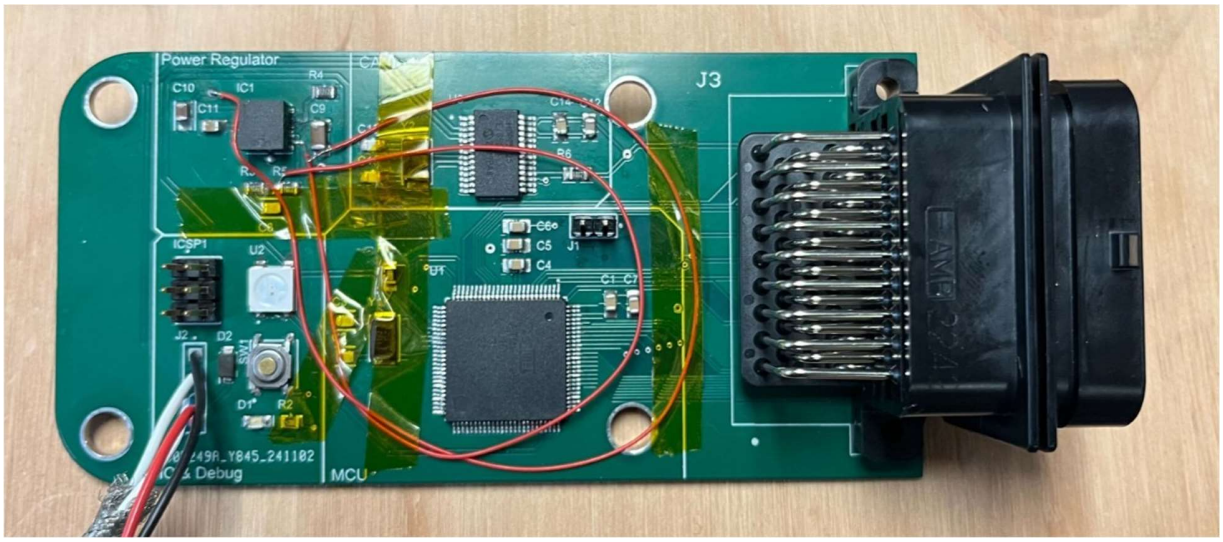
The revision B of the schematic is the same as Revision A except for a small change that has been made in the power schematic. There were two wires added, one from the feedback pin to the connection between R3 and R5 and then another wire from EN to VIN.



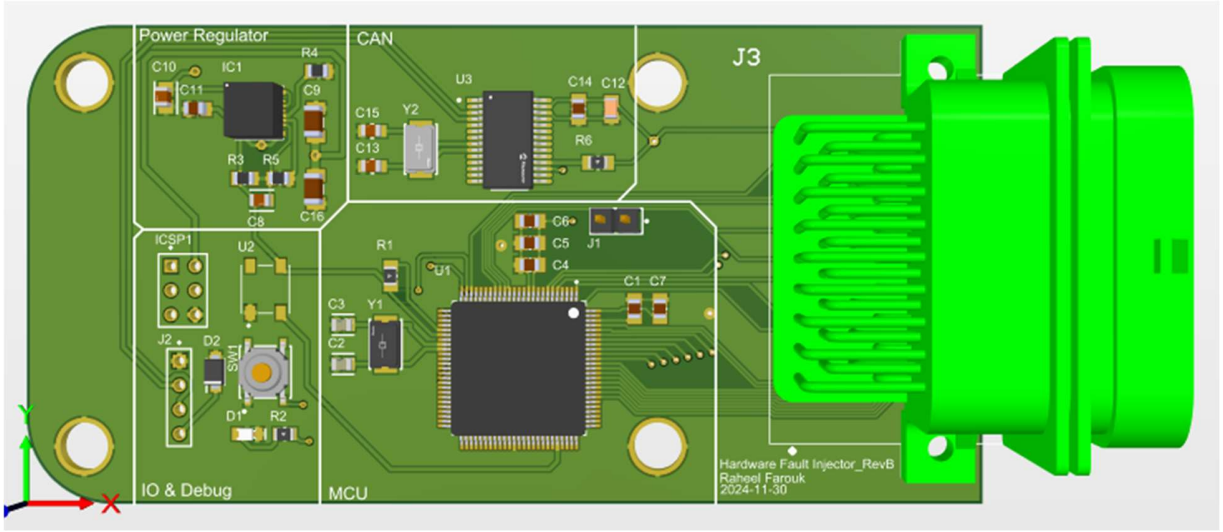
Power Connector and Regulation Schematic RevB

Hardware Fault Injector PCB - RevB

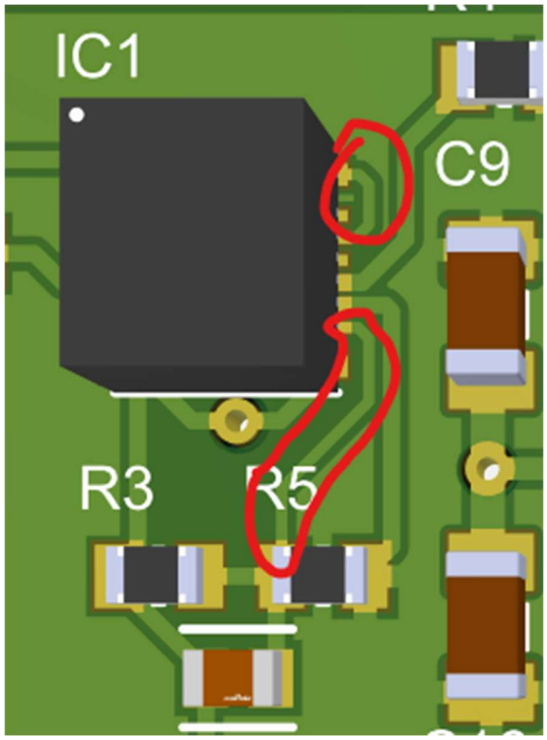
Small changes were made to the RevB that included adding two short traces in the power regulation section of the board.



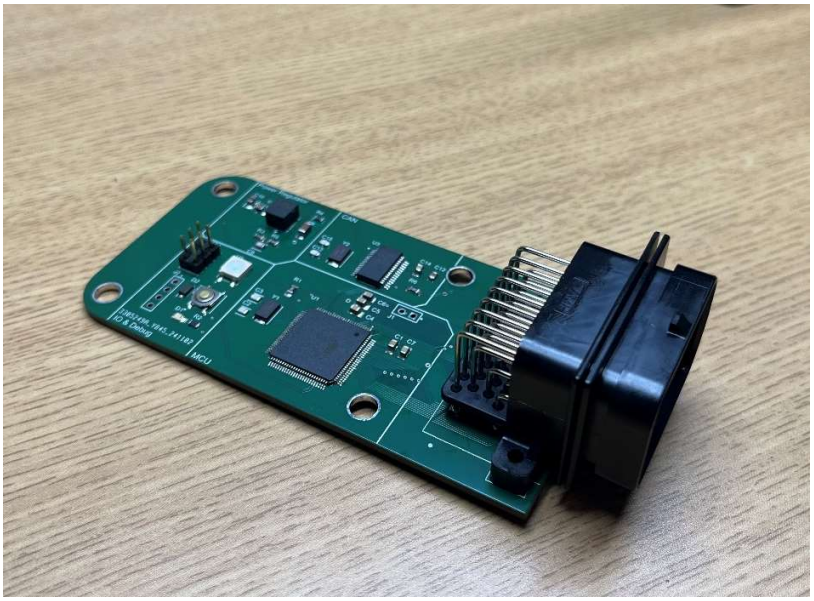
Hardware Fault Injector RevA with some temporary fixes for the missing traces



Hardware Fault Injector RevB



The circled traces are the two newly added traces



Completed RevB board

Hardware Fault Injection Embedded Software

For the embedded software of the Hardware Fault Injector, I used the Arduino IDE to write the majority of my code. This was because I had a lot of experience with the IDE and it has native support for the ATMEGA2560 microprocessor.

```
void recieveCANCommand()
{
  if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) // To receive data (Poll Read)
  {
    // Serial.print(canMsg.can_id);
    // Serial.print("     ");
    // Serial.print(canMsg.can_dlc);
    // Serial.print("   ");
    if(canMsg.can_id == 0x001)
    {
      setDebugColour(CRGB::Green);
      Serial.print("Got Mux ID: "); Serial.println(canMsg.data[0]);
      if(canMsg.data[0] == 0x00)
      {
        setOutput(2, canMsg.data[1]);
        setOutput(3, canMsg.data[2]);
        setOutput(4, canMsg.data[3]);
        setOutput(5, canMsg.data[4]);
        setOutput(6, canMsg.data[5]);
        setOutput(7, canMsg.data[6]);
      }
      else if(canMsg.data[0] == 0x01)
      {
        setOutput(8, canMsg.data[1]);
        setOutput(9, canMsg.data[2]);
        setOutput(10, canMsg.data[3]);
        setOutput(11, canMsg.data[4]);
        setOutput(12, canMsg.data[5]);
        setOutput(13, canMsg.data[6]);
      }
      else if(canMsg.data[0] == 0x02)
      {
        setOutput(44, canMsg.data[1]);
        setOutput(45, canMsg.data[2]);
        setOutput(46, canMsg.data[3]);
      }
      setDebugColour(CRGB::Black);
```

The receiveCANCommand function is one of the most important pieces of the embedded code on the Hardware Fault Injector, it handles receiving and decoding the CAN messages that will control the output pins on the HFI. It works by waiting for the message to come in, then verifies that the message is actually for the Hardware Fault Injector by looking at the CAN ID and then it looks at the multiplexor ID to determine which outputs it needs to control.

```
void setDebugColour(CRGB::HTMLColorCode code)
{
  leds[0] = code;
  FastLED.show();
}
```

I also wrote another function to be able to set the LED color of the addressable RGB LED which is used as a heartbeat and to show error messages.

Future Improvements

Many of the future improvements revolve around making the more complex functions and lines of code abstracted away from the user. This is to make it easier for a new user to be able to fully utilize the system and make the tests shorter to write. One example of this are the lines of code to close the connections and end the tests gracefully, I would like this to either be handled automatically at the end of a test or at the very least have the user only need to write one line of code to end the test gracefully. Another feature I would like to add is to be able to generate a PDF report of what tests passed and what failed and why they failed.

On the hardware side, something I noticed is that the PWM frequency of the ATGMEGA2560 chip is a little too slow in comparison to the vehicle controller sampling rate and therefore instead of seeing an analog voltage we see the PWM signal. A future fix to this would be to either fix this in software and increase the PWM frequency or to add a hardware fix by adding a capacitor to smoothen the signal coming out of the Hardware Fault Injector.

Links

Github: https://github.com/RaheelFarouk/FSAE-Automated-Regression-Tester