# Firmware Breakdown

There are many files attached to the main Arduino loop, Teensy_Firmware. In this section we will go over which files you need to be aware of and which you can ignore.

IMU: (From a documented Library)

The BNO055 takes up quite a few files. Adafruit_BNO055.cpp, Adafruit_BNO055.h, Adafruit_Sensor.h, imumaths.h, matrix.h, structs.h, quarternion.h, vector.h, BNO.cpp, and BNO.h all have to do with the IMU. If you need to look into how the IMU is handled start with BNO.cpp and BNO.h as these are the files that incorporate all the rest.

Indicator LED: (User made files)

Fault_handler.cpp and fault_handler.h hold the functions we use to initialize and set the on-board LED. There's not too much here, although if you're looking to change the LED color check out our pre-made selection in fault_handler.h.

Throttle/Steering Input and URF Distance Sensor : (User made files)

All handling of the radio receiver's PWM input is handled through input_handler.cpp and input_handler.h. Here you will find the interrupt that catches the starting timestamp of the PWM and the handling of the calculated difference between that and the ending. We limit the throttle input from –500 to 500 and the servo input from –400 to 400. The URF sensor's initializing and polling functions are also handled here.

Servo Output: (User made files)

Sending a PWM pulse to the servo happens through output_handler.cpp and output_handler.h. The useful functions here are initServo and writeServo.

CAN Library: (From a documented Library)

FlexCAN.cpp, FlexCAN.h, and kinetis_flexcan.h provide a CAN library that allowed us to simply call functions to start the Teesny's CAN bus and to read and send the CAN messages. All CAN library support comes from these two files.

Motor Controller Interfacing: (User made files)

Building from the CAN library in FlexCAN.cpp, uLaren_CAN_Driver.cpp and uLaren_CAN_Driver.h have the heavy task of creating all the specific CAN functionality to use the motor controllers. Setting up their network, initializing them, setting the target velocity and much more is found in these files.

Laser Sensor (From a documented Library)

Adafruit_VL53L0X.cpp, Adafruit_VL53L0X.h, and the twenty other files that start with "vl53l0x" all provide support for the VL53L0X Laser Sensor. The only notable files are the Adafruit_VL53L0X.cpp and the Adafruit_VL53L0X.h. The other files support these two.

Main Loop Support (User made files)

As the system is right now, we have one file that supports the main loop, loop.h. This file contains the states the main loop is allowed to cycle through.

### 1.1.1 Teensy Main Loop Brief

There are two critical things to know that pertain to the main loop file Teensy_Firmware: the state transitions and the global variables used. We'll start with explaining the state transitions.

INITIALIZE_PERIPHERALS

In this state we initialize any sensors we wish to use. The indicator led gets set to white here. Once we're done here we move on to INITIALIZE_CONTROLLERS.

INITIALIZE_CONTROLLERS

This state is used specifically for the Maxon motor controllers. We start by resetting the nodes in case they were previously in a fault state. We continue by initializing the CAN network for every node (motor controller) on the network and going around one by one and initializing them to various modes and settings. The notable settings are changing to Profile Velocity Mode and turning the controller into the Switched-On state. The code in uLaren_CAN_Driver.cpp is well documented for all settings. Once initialized, the code moves on to the WAIT_FOR_ARM state.

WAIT_FOR_ARM

As the name implies, we wait for the user to arm the system and motor controllers by turning and holding the steering as far right as possible. This is indicated on the LED by transitioning to a yellow color. Once armed, we then transition to the LINK_COMMUNICATION state.

LINK_COMMUNICATION

Here we set each motor controller to the Operation Enabled state which is the final state and enables the motors to be "running". In this state the indicator LED becomes red. The initial velocity here is set to '0' which gives the motors holding power. This stage is rather complex and sometimes we need to try to rearm the controller. The code here handles this case and is quite impressive in its robustness. Once all motor controllers are armed, we go to either the RUNNING_NOMINALLY state or RUNNING_SIMULINK state depending on whether the defined variable SIMULINK is set to a '0' or '1' respectively.

RUNNING_NOMINALLY

This stage is where the loop will stay at until we encounter an error. This state starts by checking to see if any CAN messages can be processed. It then attempts to write a new value to the motor controllers if 20 milliseconds have gone by; if it hasn't then it moves on. We then check the voltage level of the motor controllers to regulate the battery level. We decided to set a minimum voltage level of 22V and if this level isn't met the LED indicator changes to a purple color and shuts down all the motor controllers. In most cases we continue through the state and attempt to write to the servo if 10 milliseconds have gone by. In this state the indicator LED is green.

RUNNING_SIMULINK

This state is very similar to RUNNING_NOMINALLY. The difference is that instead of taking input directly from the radio receiver we send it to Simulink first and use the values that Simulink sends back. To implement this, we had to use the Raspberry Pi/Simulink as the SPI master while the Teensy was the receiver. We used a one-byte opcode to indicate to the Teensy which functionality the Raspberry Pi/Simulink is trying to use. To achieve this setting we incorporated a switch case using the one-byte prefixed opcode. Other than the switch case this state also incorporates all functionality that the RUNNING_NOMINALLY state has. Here the indicator LED is supposed to be cyan but it looks more white in reality.

INDICATE_AND_LOG_ERROR

WAIT_FOR_CLEAR

These states have yet to be implemented. They were developed in the prototype phase and serve as a base for future projects to use.

### 1.1.2 Teensy's Pertinent Variables

SIMULINK: The SIMULINK defined variable is used to switch between the RUNNING_NOMINALLY functionality and the RUNNING SIMULINK functionality by writing a '0' or '1' respectively.

MC_VOLTAGE_THRESHOLD: This variable is used to set a minimum limit that the perceived voltage by the motor controllers must not dip below. This value is counted in 0.1V and we recommend not changing its value.

CANbus: The CANbus global variable is the key to all CAN communication. Key functions regarding this variable are specified in FlexCAN.h.

Next_state: This crucial global variable is how we interpret which state we are currently in.

All data and output variables are self-explanatory and are meant to be apparent in their meaning.

Timing variables are used to control how often we exercise certain functionality. As an example, we use the motor's timing variables to write approximately 50 times per second.

PRINT: Located in uLaren_CAN_Driver.h, if set to a '1' many items will be printed in the serial port and is quite helpful for debugging purposes.

SCALE_FACTOR: Be careful with this defined variable as it controls the factor we scale the throttle by. For normal use we advise not going above 2 to maintain similitude. The motors however are capable of going much higher (up to 8 is theoretically possible but is very strongly advised against and could destroy the motors).