QFSAE: CAL

Introduction

CAL, or CAN bus Abstraction Layer, was originally proposed by our Electrical Team lead Ethan Peterson back in 2020 and was the first major project I completed as part of the Queen's Formula SAE Team.

The purpose of CAL is simple. Provide a simplified interface for programmers to work with the CAN bus. CAL currently supports encoding and decoding of MoTec PDM (Power Distribution Module) and ECU (Engine Control Unit) as well as our QFSAE Dashboard messages.

In order to understand the workings of CAL, we must first understand the basics of the CAN bus itself, and how the MoTec ECU and PDM transmit different types of data.

The CAN Bus

The CAN (Controller Area Network) Bus was originally developed in 1986 to simplify automotive wiring. While many newer cars are transferring to ethernet as an alternative communication method, the CAN bus is still widely used for a variety of applications due to being able to hold its signal integrity within high noise environments.

Data is sent through the CAN bus in the format of CAN messages. CAN messages contain an 11 bit identifier followed by up to a maximum of 8 8-bit unsigned integers. The identifier is typically referred to as an ID and is used to differentiate sending devices. For example, the MoTec ECU uses ID's 0x118 and 0x119 in order to send out different data. The ECU uses ID's 0x7F0 and 0x7F1 for its data. Within the program, this data is stored within the CAN_msg_t structure, seen below.

```
typedef struct {
    uint32_t id;        /* 29 bit identifier                               */
    uint8_t  data[8];   /* Data field                                      */
    uint8_t  len;       /* Length of data field in bytes                   */
    uint8_t  ch;        /* Object channel(Not use)                         */
    uint8_t  format;    /* 0 - STANDARD, 1- EXTENDED IDENTIFIER            */
    uint8_t  type;      /* 0 - DATA FRAME, 1 - REMOTE FRAME                */
} CAN_msg_t;
```

The 8 8-bit unsigned integers are contained within an array, while the identifier is stored in a 32-bit integer. IDs are typically stored as hexadecimal values.

The QFSAE Dependency Library

Before continuing, we must briefly mention the QFSAE dependency library. As the CAN_msg_t structure needs to be accessed by both the CAL library, the st-f4CAN library, as well as future libraries, it was moved into its own library to reduce declaration conflicts between libraries. The dependencies library was then added as a dependency library of these libraries to allow passing of message structures between them. As well as allowing development to be more streamlined, this design choice also allows users of these libraries to pass data between one another using one-liners.

Internal Data Storage

Currently, CAL is capable of both decoding and encoding CAN bus data. In the original implementation of CAL, only decoding was possible. However, when starting to work on other QFSAE projects such as the dash, it became necessary to implement an encoding mechanism. As such, CAL went to being a collection of functions to a class.

The main reason for the move from a set of functions to a class implementation was so that CAL could be capable of internally storing data. In the end implementation, this allows users to call the update variable method, followed by the package method in order to encode and send messages across the bus. An example implementation is featured below.

```
#include "Arduino.h"
#include "can.hpp"
#include "cal.hpp"

CAL::CAL cal;

void setup() {
    bool ret = CANInit(CAN_500KBPS, 0, 2);
    while(!ret);
}
CAN_msg_t can_msg;

uint8_t can_ch1 = 1;
uint8_t can_ch2 = 2;

void loop() {
    cal.updateVar(CAL::DATA_ECU::EngineRPM,
(cal.returnVar(CAL::DATA_ECU::EngineRPM) + 100));
    CANSend(can_ch2, &cal.package(CAL::MOTEC_ID::ECU_2));
    delay(100);
}
```

Another benefit of storing data internally is that in removes the need for data to be stored by the end user. This provides the benefit of decreased memory usage, and the ability to retrieve the data from the last received message.

The Update Method

When using CAL, the first step is to pass data (in the form of CAN messages) into the library to be decoded. This can easily be done through the update package method.

```
    if(can.checkReceive() == CAN_MSGAVAIL){
      CAL::CAN_msg_t can_recv;
      can.readMsgBuf(&can_recv.len, can_recv.data);
      can_recv.id = can.getCanId();
      cal.updatePackage(can_recv);
    }
```

The main advantage of calling this method is that it automatically identifies the correct storage container based off the messages ID. At a user level, the implementation featured above is all that is required by the user. However, while the user may believe this is where decoding takes place, the CAL program does not actually decode any data at this step. The program simply updates its internal data structure with the data coming in off of the bus. If the message is unknown, IE there exists no internal data structure with a matching ID, CAL simply ignores the incoming message.

```
int CAL::CAL::updatePackage(CAN_msg_t &CAN_msg){
    switch (CAN_msg.id)
    {
    case MOTEC_ID::ECU_1:
        ecu1 = CAN_msg;
        break;
    case MOTEC_ID::ECU_2:
        ecu2 = CAN_msg;
        break;
    case MOTEC_ID::PDM_1:
        pdm1 = CAN_msg;
        break;
    case MOTEC_ID::PDM_2:
        pdm2 = CAN_msg;
        break;
    case CAN_ID::DASH:
        dash = CAN_msg;
        break;
    default:
        return 1;
        break;
    }
    return 0;
}
```

The Return Variable Method

Now that CAL is receiving data off of the CAN bus, the return variable function can be called in order to access the data. The function is overloaded to support different variable types and has two different ways it can return data.

The first way data can be returned is return by reference. The advantage of using this method is that it supports many data types including regular integers, floats, booleans and unsigned 8-bit integers.

```
  // Method 1 of returning data - by reference (works with all data types)
  cal.returnVar(CAL::DATA_ECU::EngineRPM, engineRPM);
  // Print RPM to Serial Terminal
  Serial.print(String("Engine RPM: ") + engineRPM);
```

The second method for returning data is by utilizing the standard function return. This method is useful for one-liners. However, it carries the main disadvantage that it can only return integers and boolean values in the form of integers. Therefore, it will not work when accessing floats.

```
  // Method 2 of returning data - return (ONLY WORKS WITH INTEGERS!!)
  if(cal.returnVar(CAL::DATA_ECU::EngineRPM) >= 12000)
```

The Update Variable Method

While the previous methods discussed extracting data off the CAN bus and out of CAN messages, the update method is used to encode data into CAN_msg_t packets that can then be sent over the bus. The update variable function takes two parameters, and similarly to the return variable method, is overloaded to support many data types. The first parameter is the data that you would like to update, and the second is the updated value. Examples using the update method are pictured below.

```
 // Update Variable (Boolean Overload)
 cal.updateVar(CAL::DATA_DASH::UpShift, true);
 // Update Variable (Integer Overload)
 cal.updateVar(CAL::DATA_ECU::EngineRPM, 3500);
  // Inline Update Variable Example (also uses inline return variable function)
  cal.updateVar(CAL::DATA_ECU::EngineRPM,
(cal.returnVar(CAL::DATA_ECU::EngineRPM) + 100));
```

The Package Method

The Package method, as its name implies, is used to return the internally stored CAN data for a given CAN ID. This function has also been overloaded to accept variables, however this method is not recommended for use as it can lead to double sending of messages among other errors. When using CAL alongside the st-f4CAN library, CAN messages can be sent out with a one liner.

```
    // Arduino Method to send CAN Data
    CAL::CAN_msg_t &msg = cal.package(CAL::DATA_ECU::EngineRPM);
    can.sendMsgBuf(msg.id, 0, msg.len, msg.data);

    // st-f4CAN One liner method
    CANSend(channel1, &cal.package(CAL::CAN_ID::DASH));
```