Operation Guide: CAN Communication + Kelly Controllers

Formula Hybrid Senior Design Team, 5/7/2020

Introduction: the benefits of using CAN communication

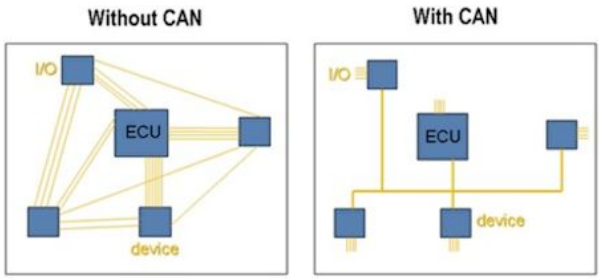

Figure 1. CAN network topology minimizes wiring (National Instruments, 2019)

CAN Bus protocol enables many IO devices/ECUs called nodes within an automotive vehicle or other system to communicate with each other with minimal wiring required (see Fig. 1). Each node broadcasts a message to all other nodes on the bus, similar to the operation of walkie-talkies at the simplest level. Some nodes, such as the engine or brake controls, relay more critical information than others, such as the air conditioning, and need priority if two nodes try to send a message at the same time. The arbitration process will determine which node can transmit its message first. All nodes can ignore messages that do not apply to them with buffers and filtering set up in the microcontroller. Overall, the low-cost, centralized processing, and robustness to electromagnetic interference makes CAN protocol so appealing for car manufacturers.

Defining Terms and Acronyms

CAN - Controller Area Network

CAN 2.0 - most recent publication of CAN protocol from Bosch in 1991

- CAN 2.0A - standard identifier (11 bits)
- CAN 2.0B - extended identifier (29 bits)

CAN 1.0 FD - Flexible Data Rate from Bosch in 2012

- Compatible with CAN 2.0, meaning that CAN 1.0 FD devices can be on the same network as CAN 2.0 devices

SAE J1939 - higher level protocol developed for trucks

- SAEJ1939 goes into more depth on the message formats, source addressing, broadcasting control, and acknowledgement of signals
    - necessary, relevant information from this protocol is specified in the Kelly Controls CAN datasheet linked below

CAN Signal

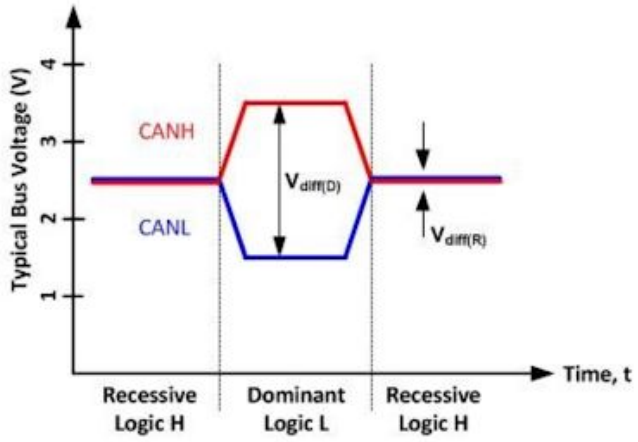

Figure 2. Differential CAN bus signal (Griffith, 2015)

Because CAN messages are differential signals, the CAN network topology contains two wires. One wire is the high wire, the other wire is the low wire. For a 5 V CAN system, the bus is in a recessive (idle) state at 2.5 V. A recessive bit is equal to a logic 1. When a dominant bit is sent, the CAN high signal goes up to 5 V and the CAN low signal goes down to 0 V. Dominant bits are equal to logic 0. This differential signal makes the bus more immune to errors from electromagnetic interference (EMI).

Hardware Set-Up

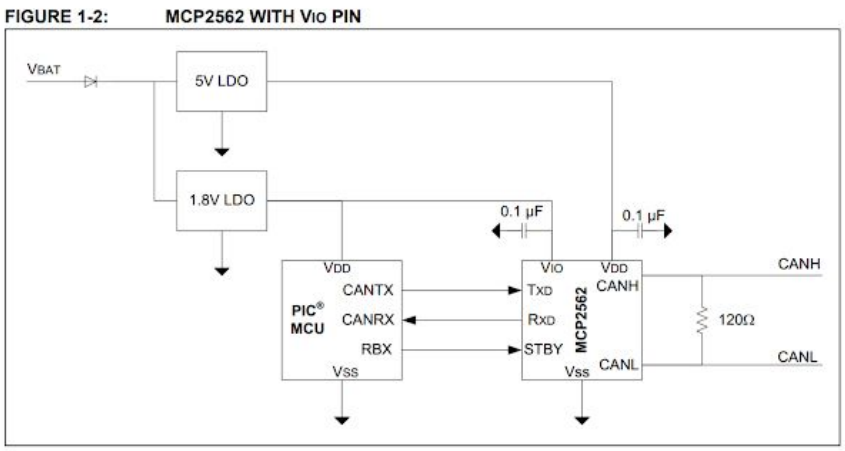

Figure 3. CAN network set-up with PIC32 and MCP2562

Microchip CAN hardware

The PIC32MX795F512H (the microcontroller on the main motherboard of the Hybrid car and the microcontroller on many Senior Design demo boards) has two CAN controllers integrated into its hardware. The PIC32MX695F512H does NOT. Be careful when selecting demo boards to test with CAN to use the 795 version.

MCP2515/MCP2510 - CAN controller with SPI interface

- Standalone CAN controller for interfacing with non-CAN enabled microcontrollers. Converts CAN data to SPI data to communicate with microcontroller. The MCP2515 board contains a transceiver on the PCB. We have two of these devices and they are useful for testing the CAN bus with the Arduino MEGA board.

MCP2561/MCP2562 - CAN transceiver

- The transceiver interfaces with the CAN controller, essentially converting the raw, differential signal into one with a logic-level signal with 1s and 0s. Do not confuse the CANTX/TXD and CANRX/RXD with UART communication. The CANTX connects to the TxD and CANRX connects to the RxD which is different from UART where the Rx connects to the Tx and vice-versa. The MCP2562 connected to one of the 795 SD kit boards will form a CAN node.

Create a CAN node with the Arduino

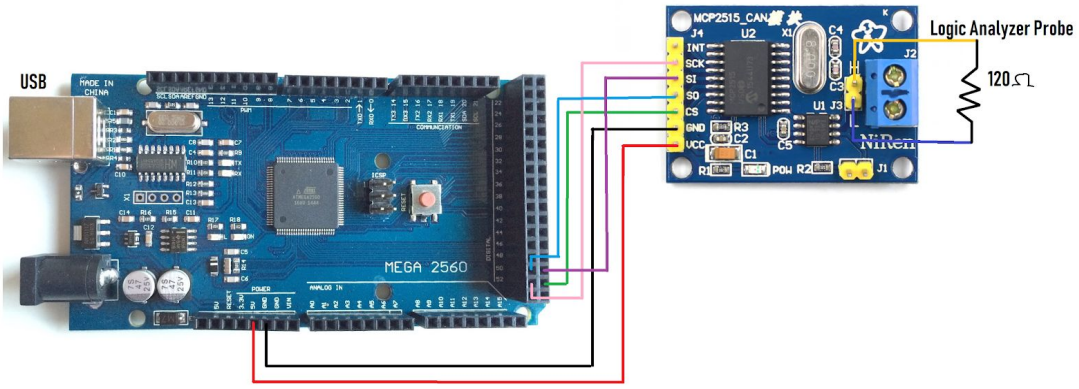

Figure 4. Wiring diagram for Arduino CAN node

1. With the current buffer and filter set up of the code, using the Arduino CAN node to debug CAN issues instead of the Kelly controllers is not recommended. This tutorial should help you understand CAN protocol and learn how to use the logic analyzer and/or the Keysight oscilloscope.
2. Materials needed: Arduino MEGA 2560, MCP 2515, 120 Ω resistor, jumper wires
3. Configure the Arduino and the CAN controller module as shown in Fig. 4.

Code to Configure Arduino MEGA 2560 as a CAN Transmitter

```
#include <SPI.h>
#include <mcp_can.h>

const int spiCSPin = 53; // specify slave select pin for Arduino MEGA
int ledHIGH    = 1;
int ledLOW     = 0;

MCP_CAN CAN(spiCSPin);

void setup()
{
    Serial.begin(115200); // UART baud rate

    while (CAN_OK != CAN.begin(CAN_250KBPS))
    {
        Serial.println("CAN BUS init Failed");
        delay(100);
    }
    Serial.println("CAN BUS Shield Init OK!");
}

unsigned char stmp[8] = {1, 0, 1, 0, 1, 0, 1, 0};

void loop()
{
  Serial.println("In loop");
  CAN.sendMsgBuf(0x43, 1, 8, stmp);
  delay(1000);
}
```

This code is adapted from:
https://www.electronicshub.org/arduino-mcp2515-can-bus-tutorial/

4. Download MCP_CAN_lib library (linked in Software Downloads) and program the Arduino with the code shown above. This program allows the Arduino and CAN controller to send CAN messages.
5. When using the logic analyzer to detect the CAN signals, be careful with the CAN rate. If 250,000 kbps does not work, try 125,000 kbps and 500,000 kbps. We are not sure why 250,000 kbps does not work because that is the correct bit rate, but it should work with one of these numbers.
6. When using the logic analyzer in general, make sure the voltage trigger level is correct for your application.
7. Examples of output and hardware are shown in Fig. 6 and Fig. 7. Note the LED and push button were removed later as they were unnecessary.
8.

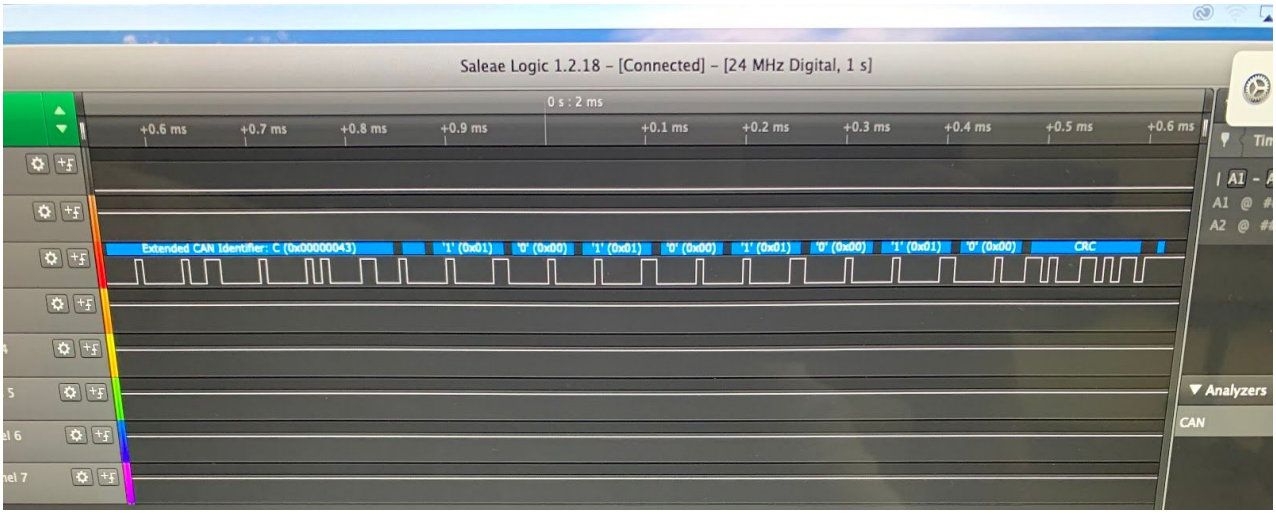

Figure 6. Logic analyzer output waveforms

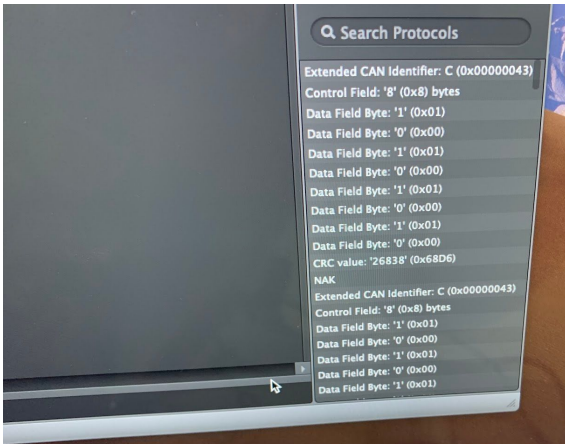

Figure 7. Logic analyzer data output

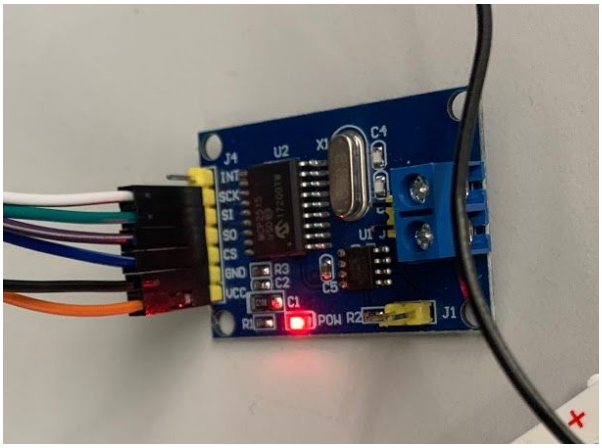

Figure 8. Enlarged view of MCP2515

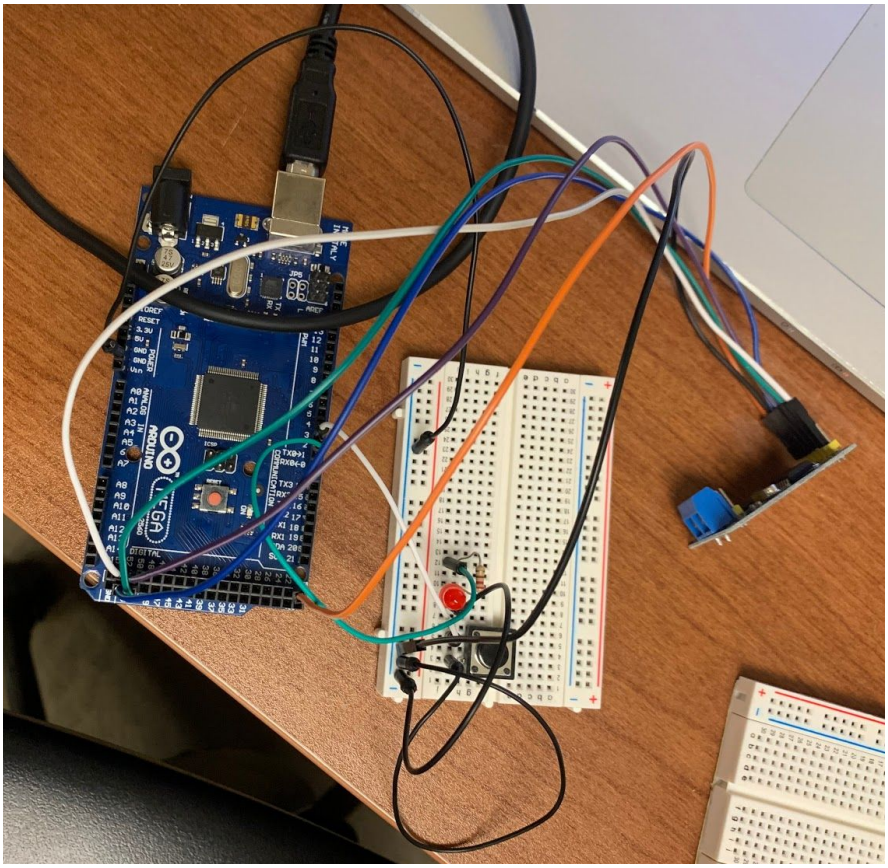

Figure 9. Arduino MEGA 2560 wired for CAN network testing

Testing CAN Protocol on Kelly Motor Controller

1. Read all relevant datasheets
2. To apply power to the controller (detached from the hybrid car), attach a 12 V DC power supply (GEMTECH in Hybrid Lab) to the Black GND (6) and Pink PWR (7) pins on connector. Determine which connector of the 3 possibilities by matching the shape of the plastic housing on the connector and the colors of the wires attached to the pins. Alternatively, the Low Voltage System in the car can be activated by attaching **the red and black leads** to a 12 V battery and flipping the ECU switch, LCD screen should turn on. If controllers are properly attached to the car, this should provide the 12 V to the controllers.

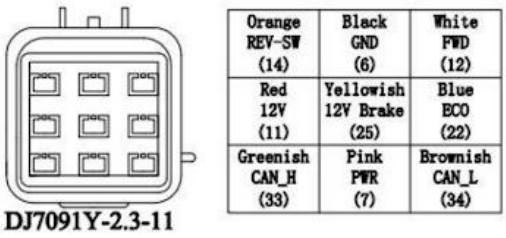

Figure 10. Wiring of Kelly Motor Controllers

3. For quick verification that a signal is being transmitted the Saelae Logic Analyzer will decode CAN messages. For more extensive debugging, the Keysight Infiniti Vision scope located in the Senior Design lab is preferred.
4. We used this tutorial to set up CAN on the oscilloscope:
   https://www.youtube.com/watch?v=dcs2QvJRsoA
5. Choose the CAN_H or CAN_L to model. Although they both add to the differential signal, when examining raw data only one is necessary.
6. We used the analog inputs on the Keysight scope, not the digital ones.
7. A photo reference of what we saw is included in Fig. 11.
8. To analyze the data, we compared the ID and the DATA fields to the Kelly CAN Protocol Description. Note that in the third data entry, the 0x1E signifies that this is one of the motor controllers and not the generator controller. It was experimentally determined by the first formula hybrid team that the generator controller will always have 0x41 in this position.

Figure 11. Keysight InfinitiVision oscilloscope analyzing CAN signal from Kelly Controllers

Using the Kelly Controls Software Application

Kelly includes an application called KMC User App.exe. This allows the user to view CAN data without implementing the CAN hardware and theoretically allows the modification of the device source address. However, changing the “J CAN Address” field did not impact the Message ID or source address. The application is fairly unreliable and usually gives error messages upon start. Although the software is unpredictable, it could be used as a last resort debugging tool. A screenshot of the software is shown in Fig. 12.

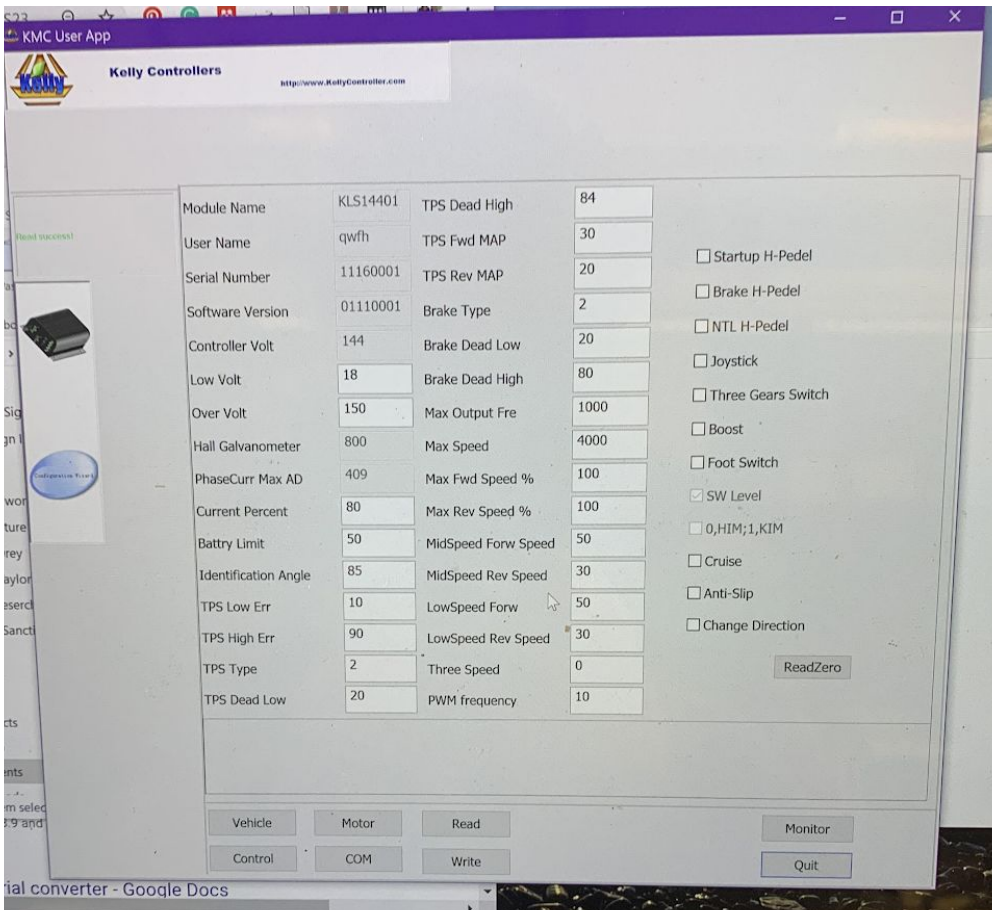

Figure 12. KMC User Application

Software Tips

(from Kelly Controls Support Team, the software was still pretty buggy no matter which OS we used after running going through these tips)

- Download the latest version of software from the website (linked above).

- Download the USB driver for the USB port in your computer. Update the USB driver if you are using Windows 10. [These update with regular Windows Updates]
- Do not install the user program in Local Disk (:C) or where anti-virus software could interfere. Close all the antivirus software before operating the user program.
- Do NOT connect the controller to the computer when the motors are running!
- You MUST turn the power supply on to use the software (and to read CAN messages).
- Please note only KDS/KDHD controllers need a Kelly SCI converter to support the application. Kelly SCI converter is useless for other controllers from Kelly.
- It is better to use Win XP or 2000 for the user program. Win7 or 8 may not be compatible with the GUI.
- Try to reset the power supply after applying settings

How to run the program in Windows XP SP3 compatibility mode:

- Right click on the User Program
- Properties > Compatibility > Windows XP Service Pack 3
- Reset power supply and open application in Administration mode.

Testing with the Kitboards in the Senior Design Lab

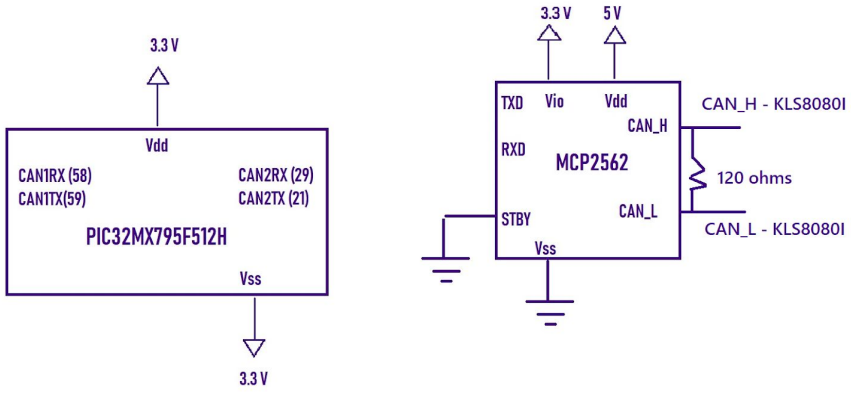

Figure 13. CAN node with SD demo boards and CAN transceiver

Using the CAN modules in the PIC32MX795F512H with MPLAB

After the CAN transceiver processes the CAN message, the CAN module must decide whether to accept the message or ignore it. In the module, there is the capability to have 32 acceptance filters and 4 masks. Initially, the receive message acceptance buffer (RMAB) stores the messages that the module will filter. When filtering the message, the mask determines which bits of the message the filter will pay attention to and which bits the filter will ignore. Then, the filters will

compare the selected bits with preset values. For example, Message 1 and Message 2 of the controller have different message IDs, 0x0x0CF11E05 for Message 1 and 0x0CF11F05 for Message 2. There is one mask and two filters to differentiate between the messages. The mask bits are all set to 1 to indicate that no bit of the message ID should be ignored. The filter bits encode the values of the message ID: 0b110011110000 for the standard identifier (SID) and 0b010001111000000101 for the extended identifier (EID) for Message 1. After filtering, the messages are stored in the CAN Message FIFOs which store a maximum of 1024 messages per module. Fig. 15 shows how the accepted messages move to the system bus where the CPU of the microprocessor stores them into the FIFOs in the RAM of the microcontroller.

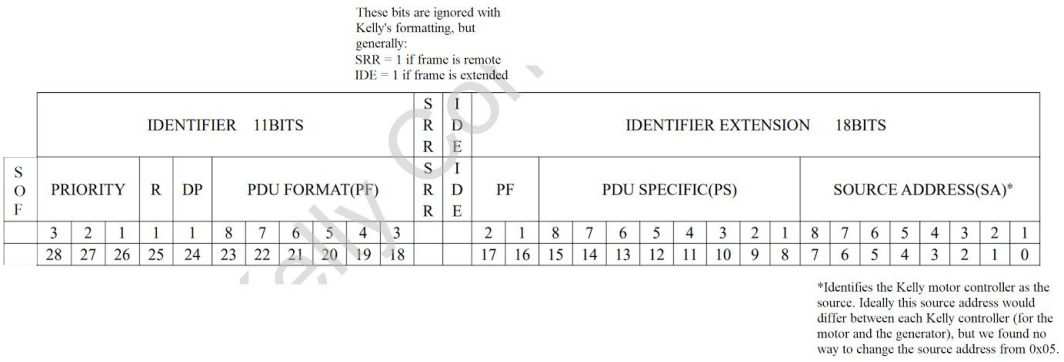

Figure 14. Message ID Format for Kelly controllers

Components of a CAN Message:
SOF - Start of Frame, dominant (0) bit
Arbitration and Control Fields - Determine priority of message and the address of the source, shown in Fig.
Data - 8 bytes of data
CRC - Cyclic Redundancy Check, 16-bit field that helps to find errors
ACK - recessive (1) sent by the transmitter and acknowledged with dominant (0) sent by receiver regardless of message acceptance
EOF - End of Frame, 7 recessive bits

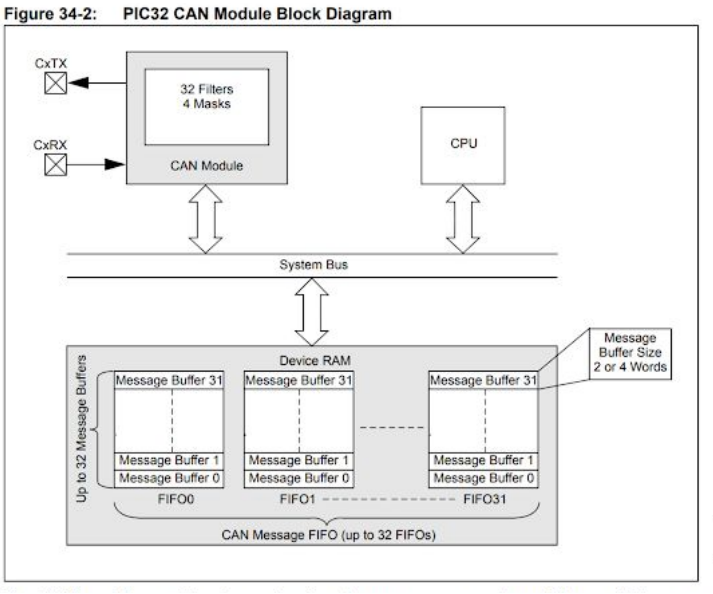

Figure 15. CAN Controller Module Block Diagram

Each message buffer has 4 bytes of data. Fig. 16 shows how the set-up of the message buffers would work in this application. Pay particular attention to the settings in the right side of the photo. TXEN denotes that it is a receive or transmit buffer, DONLY denotes whether the full message is stored or just the data bytes, and FSIZE denotes the number of words in a buffer. (note FSIZE = 3 denotes 4 words). Fig. 17 shows how the message ID and the data from each CAN message fits into the message buffers.

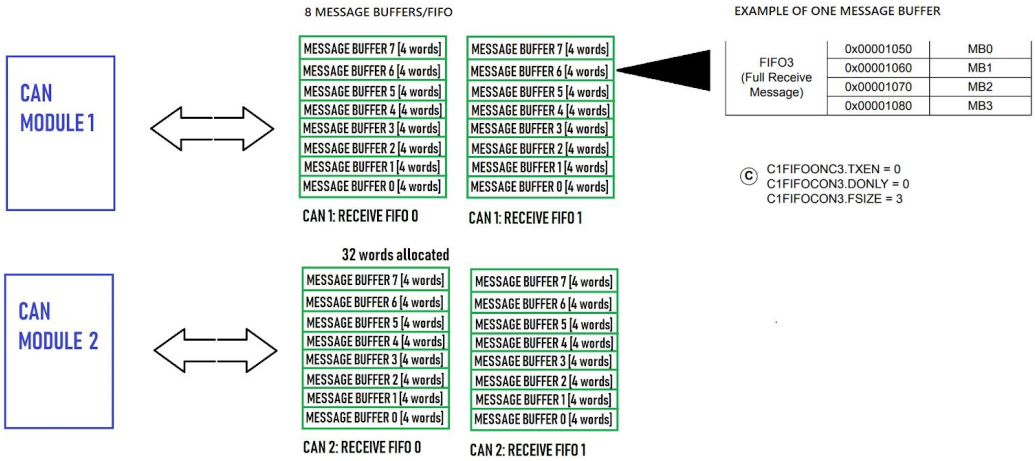

Figure 16. Individual Message Buffers: Structure and Sample Settings

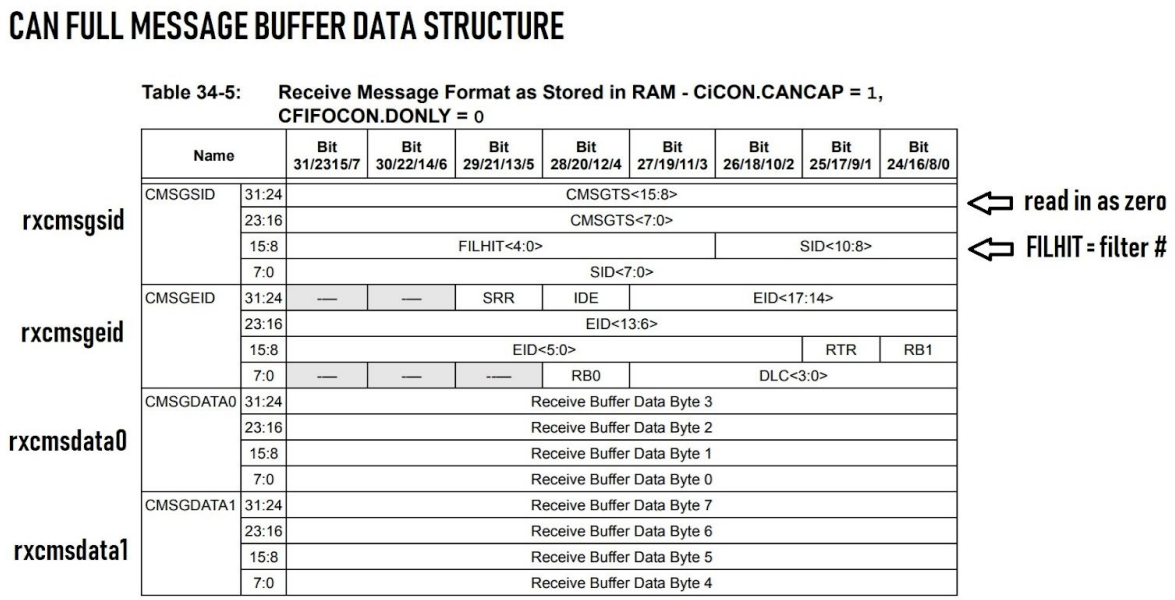

Figure 17. Contents of the receive buffer with variable names from this project

CAN Functions

Note: a lot of this code is very similar to the examples provided in the Microchip CAN reference sheet

The CAN messages are sampled once every 4 ms with the Timer 3 interrupt service routine.

initCAN2

Within the CAN configuration registers, there are different operation modes that the module has. The module itself can be turned on and off. The most prevalent operation modes are: configuration mode and normal operation mode. To detect if the module is processing messages, there is a CANBUSY bit. This function also configures the baud rate of the CAN.

set_Baud - The goal of this function is to program the module to operate the bus at 250 kbps. Since the Kelly Controllers define this bit rate and it will not adjust with new system requirements, modification of this function is unnecessary.

initFilter - Stores the Kelly Message ID so that the module can determine whether to accept or ignore a message. Selects FIFO0 and FIFO1 for storage. Requires the module to be in configuration mode for this.

initFIFO

The module must be in configuration mode for changes in FIFO set up. The KVA_to_PA function is defined by microchip and its operation is not detailed by the manual other than that it initializes the FIFO at the address given by the CanFifoMessageBuffers. The initFIFO function specifies the characteristics of the message buffer as seen in Fig. 16.

readFIFO

Process:

1. Flag enabled if FIFO0 is not empty
2. Turn the module on.
3. Flag checked to see if the FIFO is empty
4. If the FIFO0 is not empty, process the message
5. Increase the address increment pointer by 16 to indicate that the has been processed by setting the UINC bit.
6. Repeat until FIFO0 is empty
7. Repeat with Message 2 in FIFO1

ProcessMessage1

Data from the CAN message is transferred from the FIFO into variables: EID and D1-D8. The code checks to see if the data in the message is the proper length (8 bytes) in the buffer to ensure that the full message was stored. This extracts the rpm of the motors, the motor current, the battery voltage, and the two error bytes. It stores the message into the messageWord structure.

ProcessMessage2

Methods from ProcessMessage2 are repeated. This extracts the throttle level, the temperatures of the controllers and the motor, the controller status, and the switch status.

whichKelly

Differentiates between the CAN messages from the generator controller and the motor controllers. msg_array[5] will always be 0x41 for the generator controller and 0x1E for the motor controllers.

decodeErrors

This function looks at the error message seen from ProcessMessage1 and compares the set bits to the Kelly Controls CAN protocol data sheet to see which errors correspond to which bits. If there are error messages, it stores them in the Error_messages array

Relevant CAN code from the 2018-2019 Formula Hybrid team (excluding the timer interrupt functions):

```
//############################ CAN Variables ############################//

/* Data structure for CAN full receive message buffer. The following structs define
datatypes that are joined together to form the full buffer named
CANRxMessageBuffer. the rxcmsg… variables are named by user and the CMSG… types
are named by Microchip*/

/* Define the sub-components of the data structure as specified in Table 34-5 */
/* Create a C-MSG-SID data type. (see Fig. above in the operational CAN guide) */
    typedef struct
    {
        unsigned SID:11;
        unsigned FILHIT:5;
        unsigned CMSGTS:16;
    }rxcmsgsid;
/* Create a C-MSG-EID data type. */
    typedef struct
    {
        unsigned DLC:4;
        unsigned RB0:1;
        unsigned :3; // reserved bits
        unsigned RB1:1;
        unsigned RTR:1;
        unsigned EID:18;
        unsigned IDE:1;
        unsigned SRR:1;
        unsigned :2; // reserved bits
    }rxcmsgeid;
/* Create a C-MSG-DATA0 data type. */
    typedef struct
```

```
    {
        unsigned Byte0:8;
        unsigned Byte1:8;
        unsigned Byte2:8;
        unsigned Byte3:8;
    }rxcmsgdata0;
/* Create a C-MSG-DATA1 data type. */
    typedef struct
    {
        unsigned Byte4:8;
        unsigned Byte5:8;
        unsigned Byte6:8;
        unsigned Byte7:8;
    }rxcmsgdata1;

/* This is the main data structure. */
    typedef union uCANRxMessageBuffer {
    struct
        {
            rxcmsgsid CMSGSID;
            rxcmsgeid CMSGEID;
            rxcmsgdata0 CMSGDATA0;
            rxcmsgdata0 CMSGDATA1;
    };
        int messageWord[4];
    }CANRxMessageBuffer;

    volatile int EID = 0;
    volatile float rpm = 0;
    volatile float rpm_last = 0;
    volatile float rpm_new = 0;
    volatile float avg_rpm = 0;
    volatile float speed = 0;
    volatile float power = 0;
    volatile float motor_current = 0;
    volatile float mot_G = 0;
    volatile float mot_last = 0;
    volatile float mot_new = 0;
    volatile int bat_voltage = 0;
    volatile int error_lsb = 0;
    volatile int error_msb = 0;
    volatile int contr_temp = 0;
    volatile int contr_temp_G = 0;
    volatile int contr_temp_LR = 0;
    volatile int motor_temp = 0;
    volatile int mot_temp_G = 0;
    volatile int mot_temp_LR = 0;
```

```
    volatile int contr_status = 0;
    volatile int switch_status = 0;
    volatile float throttle = 0;

    volatile char* msg_array[8];
```

```
void dotheCAN(void){
    initCAN2();
    initFIFO();
    Read_FIFO();

}

void initCAN2(void){

    /* Turn CAN off to Reset it */
    C2CONbits.REQOP = 4;    /* Place the CAN module in Configuration mode. */
    while(C2CONbits.OPMOD != 4);

    C2CONCLR = 0x00008000;        /* Switch the CAN module off by clearing the ON
bit */
    while(C2CONbits.CANBUSY == 1);

    /* Switch the CAN module back ON and switch it to Configuration mode. Wait till
the switch is complete */
    C2CONbits.ON = 1;
    C2CONbits.REQOP = 4;
    while(C2CONbits.OPMOD != 4);

    /* Configure the CAN Module Clock */
    set_Baud();

    /* Set CAN2_RX pin as input */
    AD1PCFG = 0x00004100;       /* Disable Analog Mode */
    TRISBbits.TRISB14 = 1;

    /* Put it back in normal mode */
    C2CONbits.REQOP = 0;
    while(C2CONbits.OPMOD != 0);

    /* Configure filters and mask */
    initFilter();

}

void set_Baud(void){
```

```
/* Example 34-17: Configuring the CAN Module to Obtain a Specific Bit Rate */

/* This code example shows how to configure the CAN module to obtain a */
/* specific bit rate implementing the configuration shown in Example 34-16. */
/* Fsys = System Clock Frequency = 80MHz; */
/* Fbaud = CAN bit rate = 250000; */
/* N = Time Quanta (Tq) per bit = 16; */
/* Prop Segment = 1 Tq */
/* Phase Seg 1 = 7 Tq */
/* Phase Seg 2 = 7 Tq */
/* Sync Jump Width = 2 Tq */

/* Ensure the CAN module is in configuration mode.*/

    C2CONbits.REQOP = 4;
    while(C2CONbits.OPMOD != 4);

    C2CFGbits.SEG2PHTS = 1;     /* Phase seg 2 is freely programmable */
    C2CFGbits.SEG2PH = 6;       /* Phase seg 2 is 7 Tq.*/
    C2CFGbits.SEG1PH = 6;       /* Phase seg 1 is 7 Tq.*/
    C2CFGbits.PRSEG = 0;        /* Propagation seg 2 is 1 Tq. */
    C2CFGbits.SAM = 0;          /* Sample bit 1 time. */
    C2CFGbits.SJW = 2;          /* Sync jump width is 2 Tq */
    C2CFGbits.BRP = 9;         /* BRP is (Fsys/(2*Ftq))-1 = (80e6/(2*4e6))-1 = 9 */
}

void initFIFO(){
/* Allocate a total of 32 words */
    unsigned int CanFifoMessageBuffers[32];

/* Request CAN to switch to configuration mode and wait until it has switched */
    C2CONbits.REQOP = 4;
    while(C2CONbits.OPMOD != 4);

/* Initialize C2FIFOBA register with physical address of CAN message Buffer */
    C2FIFOBA = KVA_TO_PA(CanFifoMessageBuffers);

/* Configure FIFO0. This will be a receive FIFO for 4 full messages */
    C2FIFOCON0bits.TXEN = 0;    /* Clear the TXEN bit */
    C2FIFOCON0bits.DONLY = 0;   /* full message */
    C2FIFOCON0bits.FSIZE = 3;   /* size is 4 message buffers (MB0 to MB3) */

/* Configure FIFO1. This will be a receive FIFO for 4 full messages */
    C2FIFOCON1bits.TXEN = 0;    /* Clear the TXEN bit */
    C2FIFOCON1bits.DONLY = 0;   /* full message */
    C2FIFOCON1bits.FSIZE = 3;   /* size is 4 message buffers (MB0 to MB3) */
```

```
/* The CAN module can now be placed into normal mode if no further
 *  configuration is required. */
    C2CONbits.REQOP = 0;
    while(C2CONbits.OPMOD != 0);
}

void Read_FIFO(void){

   C2FIFOINT0bits.RXNEMPTYIE = 1;
/* From Example 34-4: Reading Received Messages from the FIFO */
/* FIFO0 size is 8 messages and each message is 4 words long. */
    unsigned int * currentMessageBuffer; /* Points to message buffer to be read */
    while(C2CONbits.ON == 1){
    /* MESSAGE 1 */
        /* Keep reading until the FIFO0 is empty. */
        while(C2FIFOINT0bits.RXNEMPTYIF == 1){
            /* Get the address of the message buffer to read from the C2FIFOUA0
             * register. Convert this physical address to virtual address. */
               currentMessageBuffer = PA_TO_KVA1(C2FIFOUA0);
               ProcessMessage1(currentMessageBuffer);
            /* Set the UINC bit to tell the CAN module that
             * a message has been read. */
               C2FIFOCON0bits.UINC = 1;
        }

    /* MESSAGE 2 */
    /* Keep reading until the FIFO1 is empty. */
          while(C2FIFOINT1bits.RXNEMPTYIF == 1){
            /* Get the address of the message buffer to read from the C2FIFOUA0
             * register. Convert this physical address to virtual address. */
               currentMessageBuffer = PA_TO_KVA1(C2FIFOUA1);
               ProcessMessage2(currentMessageBuffer);
            /* Set the UINC bit to tell the CAN module that
             * a message has been read. */
            C2FIFOCON1bits.UINC = 1;
        }
    }
}

void ProcessMessage1(int currentMessageBuffer){
    CANRxMessageBuffer *buffer;

/* When a message have been received and read, the individual fields of */
/* the received message can be queried as such. */
    buffer = (CANRxMessageBuffer *) (PA_TO_KVA1(C2FIFOUA0));

    if(buffer->CMSGEID.DLC == 8){
```

```
    /* If the length of the received message is 8 then process the message. */
        unsigned int * bufferToRead;
    /* Check if there is a message available to read. */
        if(C2FIFOINT0bits.RXNEMPTYIF == 1){
        /* Get the address of the buffer to read */
            bufferToRead = PA_TO_KVA1(C2FIFOUA0);

            EID = buffer->CMSGEID.EID;
            int D1 = buffer->CMSGDATA0.Byte0;
            int D2 = buffer->CMSGDATA0.Byte1;
            float D3 = buffer->CMSGDATA0.Byte2;
            float D4 = buffer->CMSGDATA0.Byte3;

            int D5 = buffer->CMSGDATA1.Byte0;
            int D6 = buffer->CMSGDATA1.Byte1;
            int D7 = buffer->CMSGDATA1.Byte2;
            int D8 = buffer->CMSGDATA1.Byte3;

            rpm = (D2*256)+D1;
            motor_current = ((D4*256)+D3)/10;
            bat_voltage = ((D6*256)+D5)/10;
            error_lsb = D7;
            error_msb = D8;

            // Store Full message 1 //
            msg_array[0] = buffer->messageWord[0];
            msg_array[1] = buffer->messageWord[1];
            msg_array[2] = buffer->messageWord[2];
            msg_array[3] = buffer->messageWord[3];

        /* Update the message buffer pointer. */
            C2FIFOCON0bits.UINC = 1;

        /* Turn CAN off */
            C2CONbits.REQOP = 4;    /* Place the CAN module in Configuration mode.
*/
            while(C2CONbits.OPMOD != 4);
            C2CONbits.ON = 0;
            while(C2CONbits.ON != 0);
        }
    }
}

void ProcessMessage2(int currentMessageBuffer){
    CANRxMessageBuffer *buffer;

/* When a message have been received and read, the individual fields of */
```

```
/* the received message can be queried as such. */
    buffer = (CANRxMessageBuffer *) (PA_TO_KVA1(C2FIFOUA1));

    if(buffer->CMSGEID.DLC == 8){
    /* If the length of the received message is 8 then process the message. */
        unsigned int * bufferToRead;
    /* Check if there is a message available to read. */
        if(C2FIFOINT1bits.RXNEMPTYIF == 1){
        /* Get the address of the buffer to read */
            bufferToRead = PA_TO_KVA1(C2FIFOUA1);

            EID = buffer->CMSGEID.EID;
            float D1 = buffer->CMSGDATA0.Byte0;
            int D2 = buffer->CMSGDATA0.Byte1;
            int D3 = buffer->CMSGDATA0.Byte2;
            int D4 = buffer->CMSGDATA0.Byte3;

            int D5 = buffer->CMSGDATA1.Byte0;
            int D6 = buffer->CMSGDATA1.Byte1;
            int D7 = buffer->CMSGDATA1.Byte2;
            int D8 = buffer->CMSGDATA1.Byte3;

            throttle = (D1/255)*100;
            contr_temp = D2 - 40;
            motor_temp = D3 - 30;
            contr_status = D5;
            switch_status = D6;

            // Store Full message 2 //
            msg_array[4] = buffer->messageWord[0];
            msg_array[5] = buffer->messageWord[1];
            msg_array[6] = buffer->messageWord[2];
            msg_array[7] = buffer->messageWord[3];

        /* Update the message buffer pointer. */
            C2FIFOCON1bits.UINC = 1;

        /* Turn CAN off */
            C2CONbits.REQOP = 4;    /* Place the CAN module in Configuration mode.
*/
            while(C2CONbits.OPMOD != 4);
            C2CONbits.ON = 0;
            while(C2CONbits.ON != 0);
        }
    }
}
```

```
void initFilter(void){
    /* Filter 0 is set up to accept messages with Kelly controller EID.
     * Accepted messages will be stored in FIFO0. Mask 0 is
     * used to implement the filter address range. */

    /* Check that it's in in configuration mode */
    C2CONbits.ON = 1;
    while(C2CONbits.ON != 1);

    C2CONbits.REQOP = 4;
    while(C2CONbits.OPMOD != 4);


    /* Kelly Message 1 Filter */

    C2FLTCON0bits.FSEL0 = 0x0000;           /* Store messages in FIFO0 */
    C2FLTCON0bits.MSEL0 = 0x00;             /* Use Mask 0 */

   C2RXF0bits.SID = 0b110011110000;         /* Kelly SID 0x0CF0*/
   C2RXF0bits.EID = 0b010001111000000101;  /* Kelly message 1 EID 0x11E05*/


    C2RXF0bits.EXID = 1;                    /* Match only messages with extended
identifier addresses */
    C2RXM0bits.SID = 1;                     /* Allow only Kelly's SID bits */
    C2RXM0bits.EID = 0b111111111111111111;  /* Allow only 0x11E05 EID bits */
    C2RXM0bits.MIDE = 1;                    /* Match only EID message types. */

    C2FLTCON0bits.FLTEN0 = 1;               /* Enable the filter */

    /* Message 2 Filter */

    C2FLTCON0bits.FSEL1 = 0x0001;           /* Store messages in FIFO1 */
    C2FLTCON0bits.MSEL1 = 0x00;             /* Use Mask 0 (already configured) */

    C2RXF1bits.SID = 0b10011110000;         /* Kelly SID 0x0CF0*/
    C2RXF1bits.EID = 0b010001111100000101;  /* Kelly message 2 EID 0x11F05*/

    C2RXF1bits.EXID = 1;                    /* Match only messages with extended
identifier addresses */

    C2INTbits.IVRIE = 1;

    C2FLTCON0bits.FLTEN1 = 1;               /* Enable the filter */
}

int last_Kelly_msg = 0;
```

```
void whichKelly(void){
    if (msg_array[5] == 0x41){
        // message from G Kelly
        mot_G = motor_current;
        contr_temp_G = contr_temp;
        mot_temp_G = motor_temp;
    }
    else if(msg_array[6] == 0x1E){
        // message from one of the motor Kellys
        mot_last = mot_new;
        mot_new = motor_current;

        rpm_last = rpm_new;
        rpm_new = rpm;

        contr_temp_LR = contr_temp;
        mot_temp_LR = motor_temp;
    }
}

void computeDynamics(void){
    whichKelly();
    avg_rpm = (rpm_last + rpm_new)/2;
    speed = avg_rpm/16.4;       // speed in mph with wheel radius of 10.25 inches

    if(BRAKE > BRAKE_OUT_MIN){
        power = mot_last + mot_new - mot_G;
    }
    else{
        power = -mot_last-mot_new-mot_G;
    }
}

void decodeErrors(void) {

    uint16_t error_msg = (error_msb << 8) | (error_lsb & 0xff);
    int i=0;
    int comparison = 0b00000001;
    char *Error_messages[] = {"ID Angle",
                              "Over Voltage",
                              "Low Voltage",
                              "",
                              "Stall",
                              "Internal Volts Fault",
                              "Controller temp",
                              "H Pedal",
                              "",
```

```
                                "Reset",
                                "Throttle Fault",
                                "Angle sensor",
                                "",
                                "",
                                "Motor Temp",
                                "Hall Galvanometer"};


    char* messages = " ";
    new_CAN_Error_MSG = 0;
    CAN_COUNT=0;
    for(i=0;i<16; i++){
        if((error_msg & comparison) > 0){
            errorPointer[CAN_COUNT] = Error_messages[i];
            CAN_COUNT++;
            new_CAN_Error_MSG = 1;
        }
        comparison = comparison << 1;
    }


}
```

Relevant Datasheets and Software Downloads

Kelly Controllers (motors: KLS 14401-8080I, generator: KLS 14401-8080IPS)
https://kellycontroller.com/wp-content/uploads/kls-8080i-ips/KLS8080I-IPS-Opto-isolated-Sinusoidal-BLDC-V1.10.pdf
Kelly Controller Specific CAN Protocol Description
https://kellycontroller.com/wp-content/uploads/kls-8080i-ips/Sinusoidal-Wave-Controller-KLS-D-8080I-8080IPS-Broadcast-CAN-Protocol.pdf
Kelly Controller Software Download
https://www.kellycontroller.com/support/ (we used PC version KMC User App.exe)
FAQ (generally useful)
https://www.kellycontroller.com/faqs/
PIC32MX795F512H Datasheet
http://ww1.microchip.com/downloads/en/DeviceDoc/PIC32MX5XX6XX7XX_Family)Datasheet_DS60001156K.pdf
Standalone CAN Controller with SPI (MCP2515)

http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf
High Speed CAN Transceiver (MCP2562)
https://www.mouser.com/datasheet/2/268/20005167C-1512552.pdf
2018-2019 Formula Hybrid Senior Design Team Documentation
http://seniordesign.ee.nd.edu/2019/Design%20Teams/ecar/index_code.html#
Arduino CAN library
https://github.com/coryjfowler/MCP_CAN_lib
Arduino CAN tutorial used:
https://www.electronicshub.org/arduino-mcp2515-can-bus-tutorial/

CAN overview pictures from:
https://www.ni.com/en-us/innovations/white-papers/06/controller-area-network--can--overview.html

https://e2e.ti.com/blogs_/b/industrial_strength/archive/2015/06/04/what-do-can-bus-signals-look-like

Code formatting:
Used Code Blocks add on in Google docs for code formatting
https://gsuite.google.com/marketplace/app/code_blocks/100740430168