Ashley Sharp

Implementing Controller Area Network

Hybrid Drivetrain ECU

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Inplementing Controller Area Network

Date 9.1.2017

Helsinki
Metropolia
University of Applied Sciences

Abstract

Author(s) Title Number of Pages Date	Ashley Sharp Implementing Controller Area Network 30 pages + 2 appendices 27 December 2017
Degree	Bachelor of Engineering
Degree Programme	Engineering
Specialisation option	Electronics
Instructor(s)	Matti Fischer

This document will discuss the implementation of Controller Area Network (CAN) communication protocol as a substitute for general analog and Digital (I/O) signals as a primary means to control an automotive hybrid drivetrain as to withstand high electrical signal interference produced and associated with electric vehicle propulsion elements.

This work was performed within the scope of a project to develop an All-Terrain Vehicle electronic control system.

Keywords	CAN Network, ECU, CAN, Automotive, Microcontroller

Helsinki Metropolia University of Applied Sciences

Contents

1 Introduction 1

2 Controller Area Network 1

2.1 Description 1
2.2 The CAN Standard 2
2.2.1 Standard and Extended CAN Format 2
2.2.2 Message Field Bit Description 3
2.3 Message Arbitration 4
2.4 Bit Timing 5
2.4.1 Bit Structure 5
2.4.2 Synchronizing and Propagation 8
2.5 CAN Message Frames 9

3 CAN Physical Layer 11

3.1 CAN Transceiver 11
3.2 Physical Line Topology 12
3.3 Line Filtering 12
3.4 Transceiver Operation 14

4 CAN Controller Protocol 15

4.1 ARM NXP Mbed Microcontroller 15
4.2 CANopen 16
4.2.1 CANopen Object Dictionary 17
4.2.2 Communication Configurations 17
4.2.3 Network Management Protocols (NMT) 18
4.2.4 Service Data Objects (SDO) 19
4.2.5 Process Data Objects (PDO) 20
4.2.6 Synchronization Object Protocol (SYNC) 21

5 Hybrid Vehicle Control 21

5.1 System Configuration 21
5.2 Control Code Block Diagram 24
5.3 Initialization 24
5.4 Code Operation Sequence 28

6 Conclusion 30

References 31

Appendices
Appendix 1. Main program control code
Appendix 2. Main ECU PCB for Hybrid Vehicle

Abbreviations

ACK	Error Free Return Bit
AMP	Arbitration Message Priority
API	Application Programming Interface
ARG	Code Argument
ATV	All-Terrain Vehicle
CAN	Controller Area Network
CD	Collision Detection
CPU	Central Processing Unit
CRC	Checksum
CSMA	Carrier Sensed Message
DCF	Device Configuration File
DLC	Byte Data Length
DSP	Digital Signal Processer
ECU	Electronic Control Unit
EMCY	Emergency CAN Message
EMI	Electromagnetic Interference
EOF	End of Frame
EEPROM	Electrically Erasable Programmable Read Only Memory
ESD	Electric Spark Discharge
IC	Integrated Circuit
ICE	Internal Combustion Engine
IDE	Identifier Extension
IFS	Interface Space
I/O	Input/output
ISO	International Standard
MCU	Microcontroller Unit
NMT	Network Management Task
NRZ	Non-Return to Zero
PDO	Process Data Object
PWM	Pulse Width Modulation
RPDO	Receive Process Data Object
RPM	Revolutions per Minute
RTR	Remote Transmission Request

SDO	Service Data Object
SMD	Surface Mounted Device
SOF	Start of Frame
SYNC	Synchronized Object Protocol
TPDO	Transmit Process Data Object
TTL	Transistor-Transistor Logic
TVS	Transient Voltage Suppressor
USB	Universal Serial Bus

1 Introduction

Controlling an automotive drivetrain, let alone a hybrid (electrical + internal combustion) system is a complex issue. With the advance of sensor technology, we have a multitude of information being passed around the vehicle that in some cases is critical to normal safe operation. Different methods are used to deliver these information signals, one being the use of physical analogue or simple digital Boolean signals. But as system information load and signal interference increases there becomes a need for a robust high-speed signal information platform.

We will now discuss the incorporation of the controller area network communication medium including the protocol and necessary hardware to make this system operate within an automotive vehicles control system. This modular system is comprised of a master node being an All-terrain vehicles (ATV) main electronic control unit (ECU) that incorporates two NXPLPC1768 microcontrollers connected to two Microchip MCP2551 controller area network (CAN) transceivers that drive the physical CAN BUS signal. This Master ECU is connected in a modular format to the two 3 Phase electric drive inverters as well as a brake control ECU and an Instrumentation cluster ECU.

2 Controller Area Network

2.1 Description

Controller Area Network (CAN) is a Serial, Asynchronous, broadcast type communications system developed by Bosch GmbH in the 1980s. It was originally developed for the automotive industry to replace the standard wiring harness with a simple two-wire bus. The specifications of the system allow for robust EMI tolerances and the ability to self-check and correct its own errors. It can be referred to as a multi master bus connecting various control units which are referred to as nodes.

CAN is a system where all nodes connected to the CAN bus can hear every message being sent at the same time simultaneously. This method allows microcontrollers and other CPUs to communicate with each other without the need of a host. In a CAN network, many small messages are sent around the whole bus allowing message consistency as opposed to other network methods like USB which send large message blocks from point to point under a host supervisor.

2.2 The CAN Standard

CAN is an International standard (ISO) which defines everything about the CAN Network. The communication protocol ISO-11898 describes how the information on the network is handled and how the system conforms to the Open Systems Interconnection model (OSI). In Figure 1, the application layer allows for the link to upper level application specific protocol such as CANopen as we are using in this scenario.

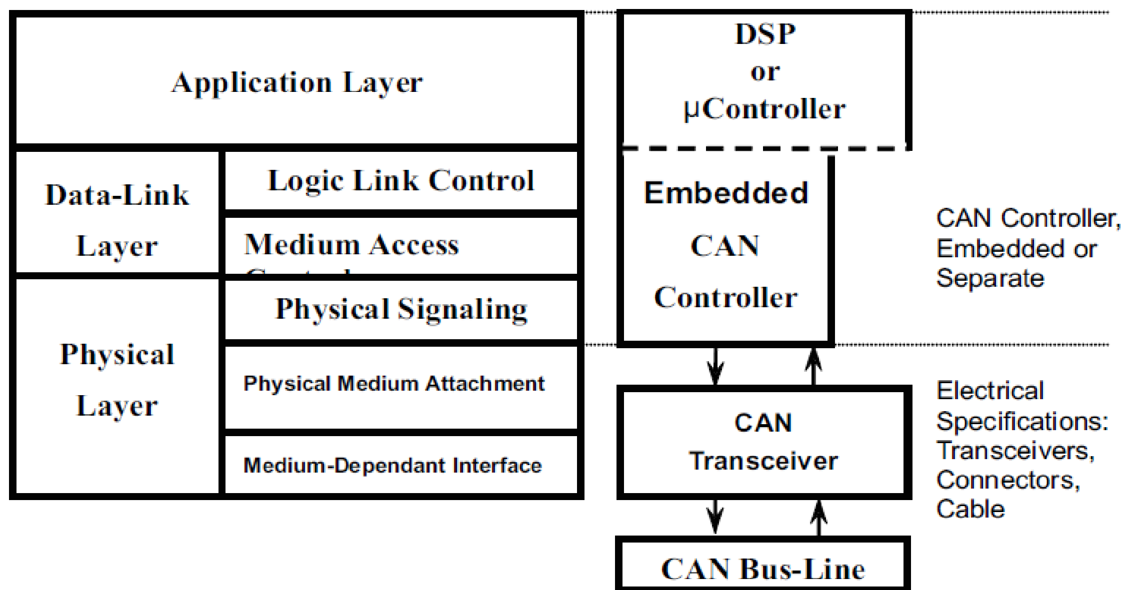

Figure 1. ISO 11898 Standard Architecture

The data layer link here is the can controller which in our case is embedded within the NXP LPC1768 microcontroller.

2.2.1 Standard and Extended CAN Format

The CAN standard protocol ISO 11898-1 and ISO 11898-2 or also known as CAN 2.0A or CAN 2.0B define 2 types of message formats. 2.0A defines the CAN standard message which has an 11-bit standard identifier field (or arbitration field) as opposed to the CAN Extended format which has an extra 18 bit identifier field.

The communication method is carrier sensed (CSMA) with collision detection (CD) which works by arbitration message priority (AMP). What (CSMA) means is that each node on the bus must wait for a specific bit length of inactivity before sending a new

message (CD+AMP) means that colliding messages are handled via bitwise arbitration. Higher priority message will always win arbitration.

Arbitration will be explained later in this document. The standard 11-bit identifier can provide signal frequency up to 1Mbps. This was later updated to the 29-bit identifier or (Extended) identifier.

Table 1. Maximum number of message Identifiers

Identifier Field	Message Identifier Number (Maximum)
11 Bit Standard	$2^{11} = 2048$
29 Bit Standard	$2^{29} = 537 \times 10^6 = 537\ Million$

These value state the maximum number of identifiers allowed for message types A and B.

2.2.2 Message Field Bit Description

Below is a reference to the CAN field identification for CAN 2.0A CAN Standard.

Figure 2. Complete CAN Message Field

1. **SOF** – The Start of frame bit when held dominant is used to start the message and synchronizes all the nodes once the bus becomes idle.
2. **RTR** – remote transmission request is used when information is needed from another node. All nodes receive this request but only the nodes required will respond.
3. **IDE** – Identifier extension bit tells the system what ID format is being sent, for example, Standard or extended.
4. **r0** – is a reserved bit.
5. **DLC** – Is the Byte Data length of the message.
6. **Data Field** - Up to 64 bits (8 Bytes) of information data can be sent controlled by DLC value.
7. **CRC** – Contains the checksum of all bits transmitted for all preceding data for error detection.

8. **ACK –** All nodes who receive an accurate message will overwrite this recessive bit with a dominant bit to show that the message was received error free. If the node detects an error, then it will leave this field recessive and the sending node will then repeat the message again after arbitration.
9. **EOF** – This end of frame 7-bit field signals the end of the CAN message and will disable bit stuffing.
10. **IFS** – Interface space contains 7 bits which is the time required by the controller hardware to move a correct message frame into the registers of the message buff-er.

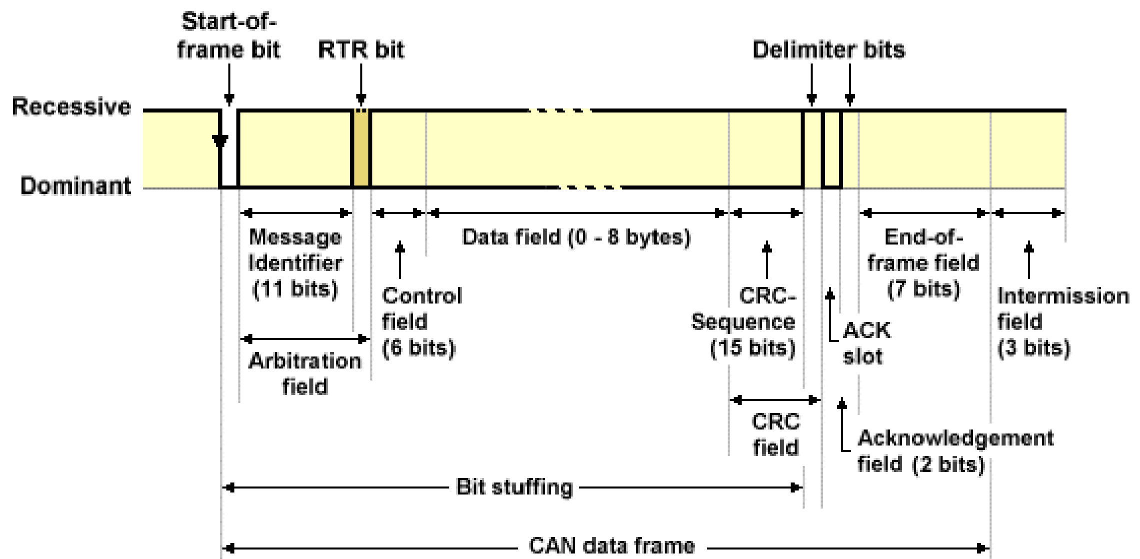

Figure 3. CAN standard format expanded

For our application, we will be using only the 11-bit identifier. There are further additions to the message field when dealing with the 29-bit identifier that we will not go into detail with here.

2.3 Message Arbitration

The CAN bus uses non-return to zero (NRZ), and each node is wired to the bus (CAN H/CAN L). If one node drives the bus to Logic 0 then the whole bus is driven to that state regardless of any node driving it to Logic 1. Logic 0 in CAN implementation is the dominant state, Logic 1 is the recessive state. This is where the arbitration process comes into play.

Notice the 11-bit message identifier and the RTR bit making up the Arbitration field in Figure 3. The node sending the lowest 11-bit binary identification message wins arbitration and is allowed to continue (See figure 5).

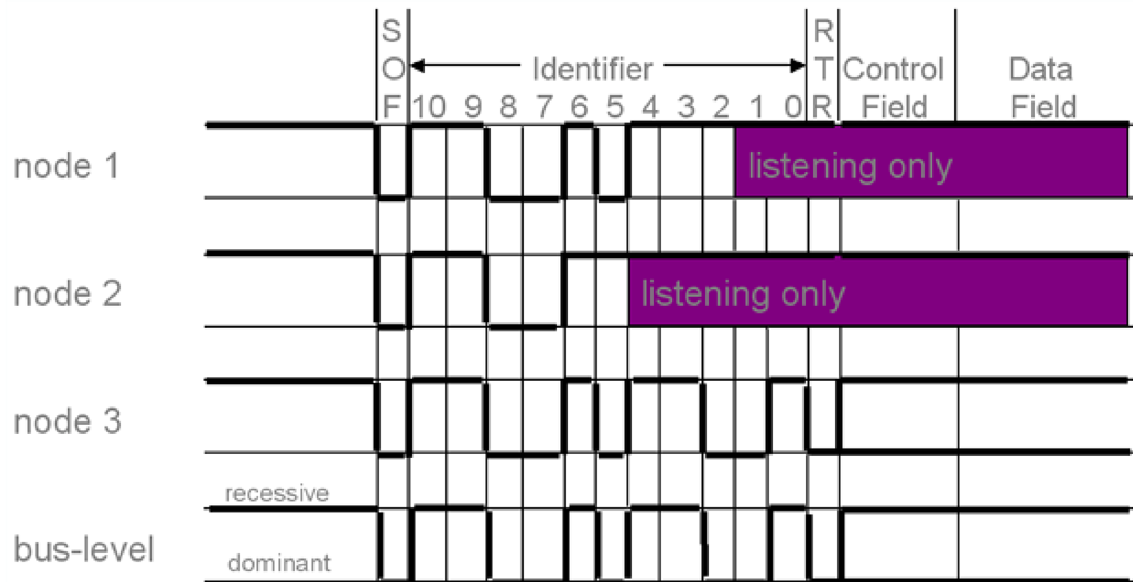

Figure 4. Three Node message arbitration

By using this method of Arbitration all messages can operate in a non-destructive manner.

The dominant state will overwrite the recessive state always. The Non-destructive and transparent nature of this means that valuable information (Bus Data) will not be corrupted or interrupted by the arbitration process (This is especially important for high speed real time environments needing Real Time control data). Once the message is sent and the 7-bit message EOF (End of frame) field and intermission field has passed then the BUS is free, and the next message is able to send.

2.4 Bit Timing

2.4.1 Bit Structure

Setting up the CAN bit timing is an important process. When you calculate and implement specific bit rate parameters in the Laboratory and then move to the real operating environment, sometimes the system does not operate in the correct manner. So, selection of oscillator components are critical to stable and fault free operation as all nodes must read a bit state correctly at the same time throughout the network while dealing with propagation delay times, as well as component tolerances.

Looking at the structure of one bit within the CAN network, the bit rate must be uniform throughout for communication between asynchronous nodes to be possible.

System nominal bit time is given by:

$$f_{NBT} = \frac{1}{t_{NBT}} \tag{1}$$

Where t_{NBT} is the nominal bit time.

The structure of each bit is made up of 4 separate segments which are as follows:

Table 2. Bit segments

Segment Function	Register ID
Synchronizing	SYNC_SEG
Propagation	$PROP_SEG$
Phase 1	PHASE_SEG1
Phase 2	PHASE_SEG2

The segments are arranged in the order as seen below,

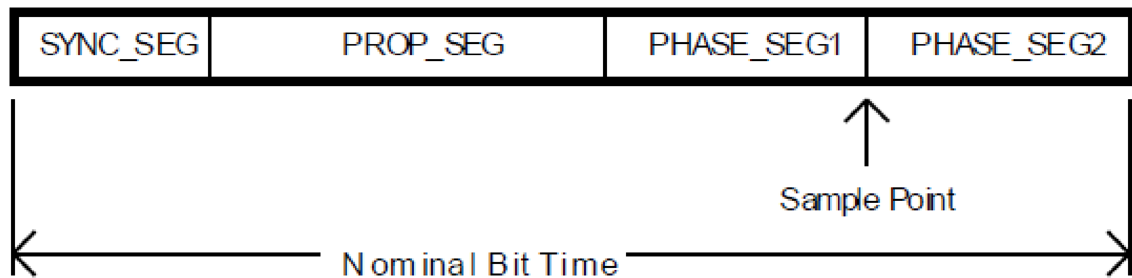

Figure 5. Bit segment structure

The sample point shown in figure 5 is the nominal bit sampling point for CAN systems. Sample points from 80-90% of total nominal bit time are recommended. The total period of the nominal bit time can then be calculated as,

$$t_{NBT} = t_{SYNC_SEG} + t_{PROP_SEG} + t_{PHASE_SEG1} + t_{PHASE_SEG2} \tag{2}$$

Each bit time segment is and integer multiple of the Unit time-quanta t_Q.

The unit t_Q represents the CAN clock period which is in turn derived from the MCU system clock frequency by calculating it in the manner show below in Equation (3)

$$\frac{(MCU\ Clock\ frequency)\ Hz}{Baud\ Rate\ Prescaler(Programmable\ Integer)} = CAN\ System\ Clock\ Hz \quad (3)$$

Below shows the relationship between main oscillator frequency and CAN clock by the way of an integer divisors which is referred to as the Baud rate pre-scaler.

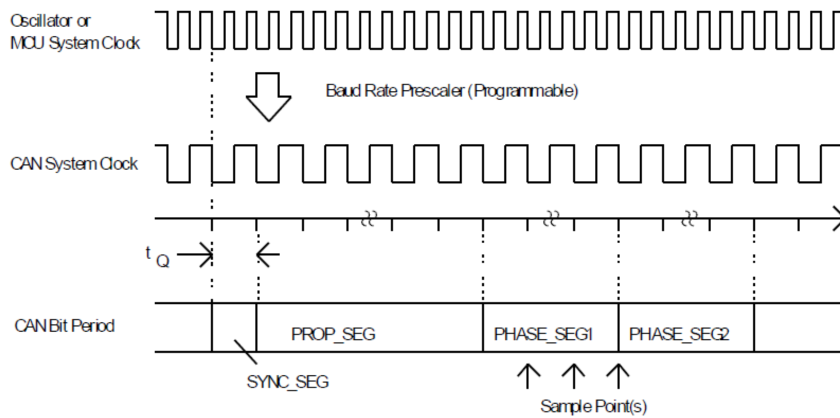

Figure 8. Baud rate prescaler

The Synchronization segment of bit timing is set to a default value of 1 time-quanta and cannot be reprogrammed, but the quanta for the remaining segments can be program adjusted.

Table 9. Time quanta per Bit

Segment	Q Value
Sync_Seg	$t_{SYNC_SEG} = 1t_Q$
Prop_Seg	$t_{PROP_SEG} = 1,2 \ldots 8t_Q$
Phase_Seg1	$t_{PHASE_SEG1} = 1,2 \ldots .8t_Q$
Phase_Seg2	$t_{PHASE_SEG2} = MAX(IPT, t_{PHASE_SEG1})$

Many CAN controllers require a minimum of 8 time-quanta per bit and a maximum of 25 time-quanta per bit.

2.4.2 Synchronizing and Propagation

For each Node connected to the CAN bus, the start of their respective bit begins with the SYNC_SEG segment. Now theoretically all node bit times begin at the same segment, but in the case of a node that is transmitting as opposed to a Node that is receiving, we can begin to see where we could have problems as there is no synchronized timing structure.

If a transmitting node begins its bit transfer which occurs at the beginning of the SYNC_SEG segment, the receiving node would then expect to receive this bit in the sync segment. Due to propagation of the signal down the length of the BUS as well as switching delays at the Nodes transceivers, there will be fractional delay in the received signal compared to the sent signal, and in the reciprocating signal from the receiving node back to the transmitting node. This is where the propagation segment comes into play.

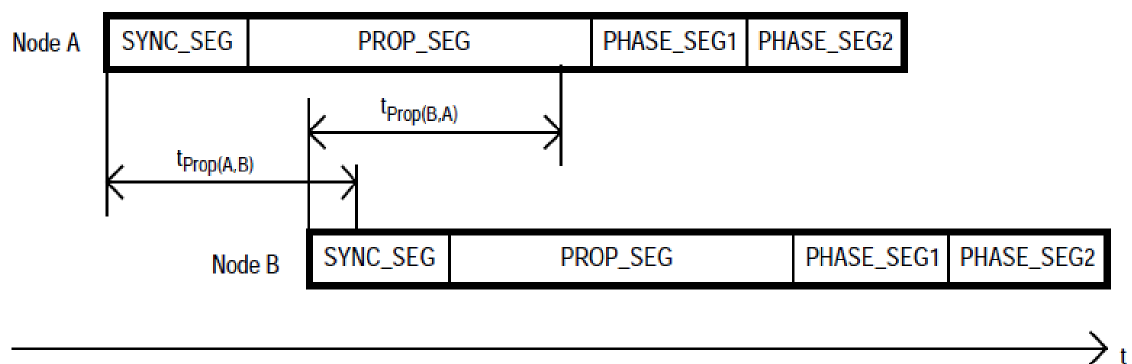

Figure 9. Bit timing delay

CAN bit timing calculations have five parameter requirements, as shown below.

1. Bit Rate
2. Bit Rate Accuracy
3. Sample Point
4. Sample Mode
5. Re-synchronization Jump Width

1. Bit Rate -

Bit rate is usually determined by the manufacturer of the device or from an industry standard protocol.

2. Bit Rate Accuracy -

Accuracy in the system is usually good when an external crystal is implemented. Ceramic resonators can be used for lower bit rate protocols.

3. Sample Point -

The sample point of the timed bit is usually determined by the manufacturer or by the industry standard of the devices application. Unlike standard forms of communication protocol like UART which have 50% sample points, the CAN protocol places its sample point in the optimal area of 80-90% of the bit period. Industry standards use this range for bit time in the 500K BPS.

4. Sample Mode -

Sample modes can be preset to single sample mode and 2of 3 majority wins sampling. These are the two modes that can be handled by the chips themselves.

5. Re-synchronization Jump Width **-**

This parameter is very important as it is the time allocated to allow for compensation of timing variation between various nodes.

2.5 CAN Message Frames

Within the CAN message there can be 4 types of message frames,

The Data frame, Remote frame, Error frame and the Overload frame. The data frame is the most common, sending the 8 Byte data. The remote frame is used to get information from another node and is similar to the data frame other than the RTR bit in the arbitration is recessive and there is no data frame.

The error frame is special in that it is transmitted when a node detects an error causing all other nodes to send an error frame. The original transmitter will then resend the message, error counters are in place to ensure that a node cannot tie up the Bus by continually sending error frames. The Overload frame is used when a node become too busy and allows for some delay between messages.

Below in figure 10 is an oscilloscope screen grab of a single CAN message. The message is of Extended format sending a three byte data frame with the data consisting of 3 Bytes (A6 00 00).

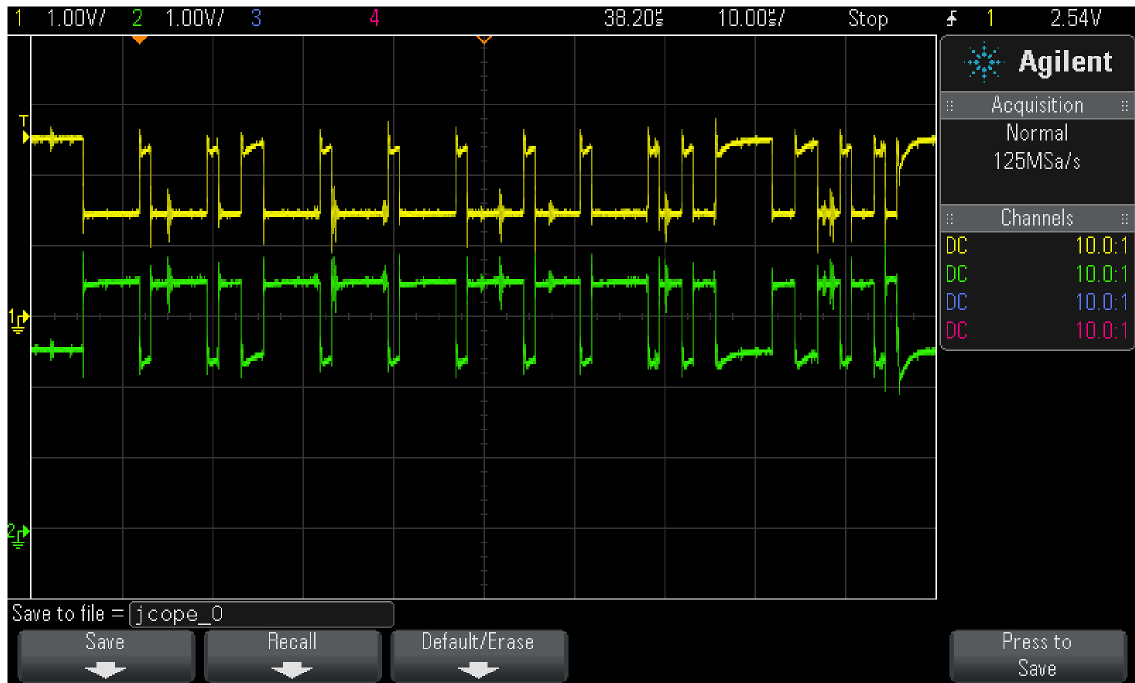

Figure 10. Oscilloscope output of CAN message frame

Below in figure 11 you can see the CAN H and CAN L transmission line signals. The voltage range is a center line 2.5V signal with one line going high to 3.5V and the other going low to 1.5V showing the 2V differential signal of the bus to each node.

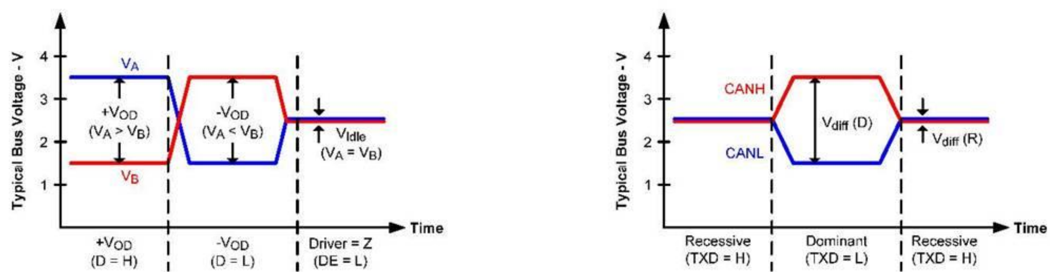

Figure 11. CAN bus logic levels

The transmitting node always internally monitors each bit of its own transmission. As you will see from Figure 17 the CAN H and L bus lines are internally connected to the receivers input line from the CAN controller.

3 CAN Physical Layer

3.1 CAN Transceiver

The CAN transceiver is the physical layer driver for the network implemented by driving the twisted pair Bus high or low. Below in Figure 12, it shows the IC connection pin out.

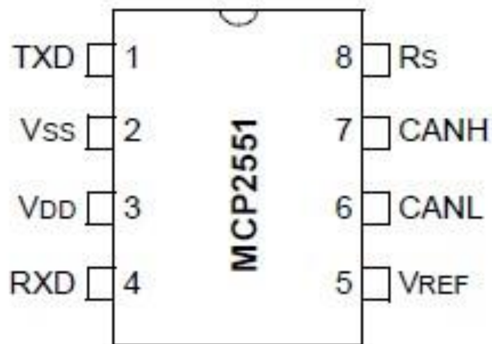

Figure 12. MCP2551 Topography

The CAN bus connection will connect to each node on the CAN H and CAN L pins of the transceiver, and the line must also be terminated by a 120 Ohm resistor at each end to assist with limiting signal wave reflections. Only two termination resistors should be used at each end of the can bus as more resistors will increase the load upon the drivers.

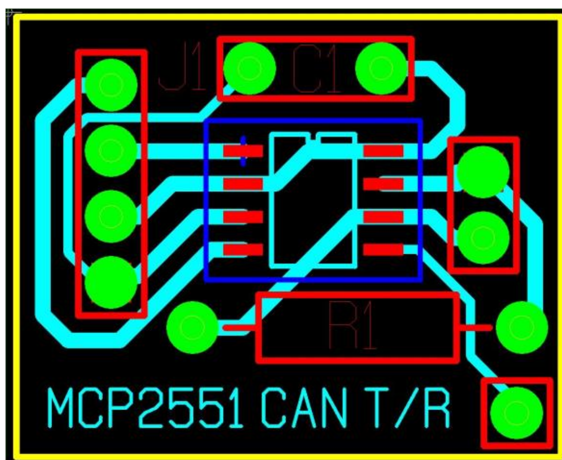

Figure 13. Breakout board for SOT-8 SMD in Mentor PADS Software.

This device can handle up to 112 nodes, but the ISO standard recommends a 45 Ohm minimum load impedance at a maximum line distance of 40 meters for speeds of 1MHz. Much longer distances are possible for lower speed transmissions.

3.2 Physical Line Topology

ISO 11898-2 gives the line topology recommendations for different data transmission rates.

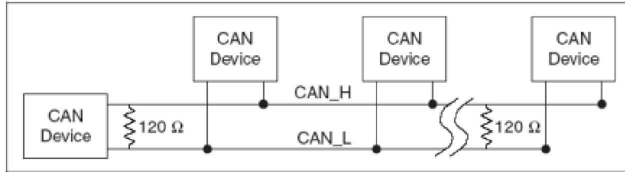

Figure 14. CAN Line Topology

Line termination is important when dealing with high speed TTL signals as a means to limit or remove signal wave reflections. If the bus lines Zo = Z_{term} (in a finite line distance) is matched, then when the logic signal voltage is placed across the Source, current flow is the same as in the case of an infinite length cable and transmission line effects no longer need to be considered.

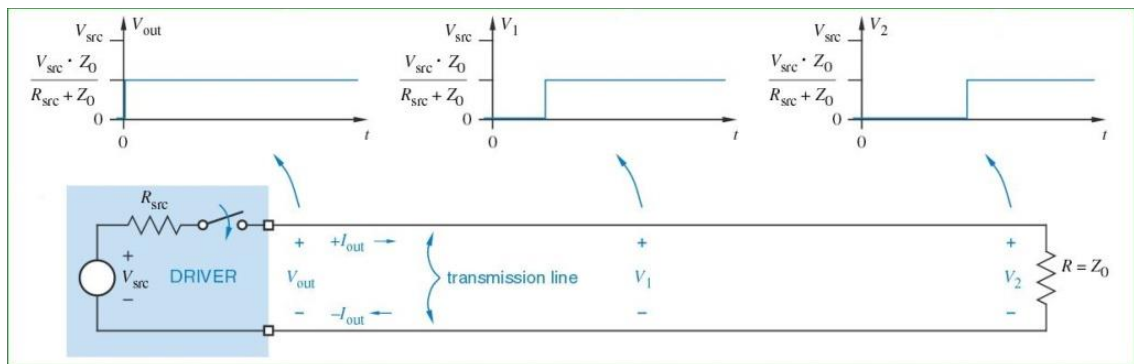

Figure 15. Line Termination

Zo characteristic impedance of the can transceiver can be found in the datasheet of any IC and in our case, it is 60 Ohms. The lines are terminated by a 120ohm resistor on each end of the bus.

3.3 Line Filtering

The CAN bus can also be filtered for interference by not only parallel termination resistors but also using a parallel common mode filter (See Figure 16). A typical C_T value of 4.7nF gives a -3dB corner frequency of 1.1MHz for high speed signals but this value is completely signal rate dependent.

The type of filtering is selected based on the signal frequency and the type of electrical interference around the CAN Bus.

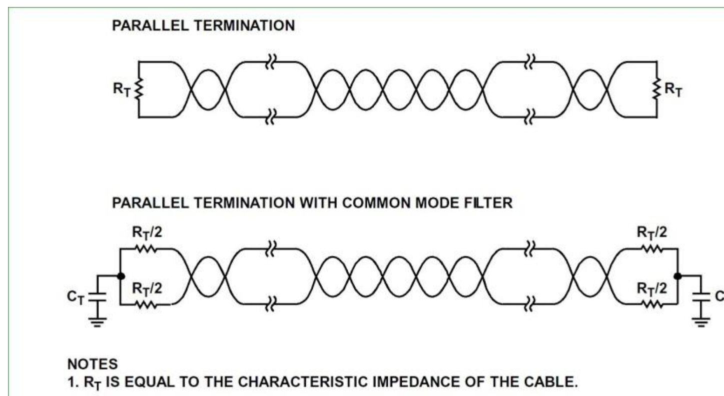

Figure 16. Common Mode Filter

The CAN bus lines can also be protected from over voltage spiking, by using TVS Zener diodes to limit damage to the transceiver drivers. Below (Figure 17) shows a typical application of the line protection connections.

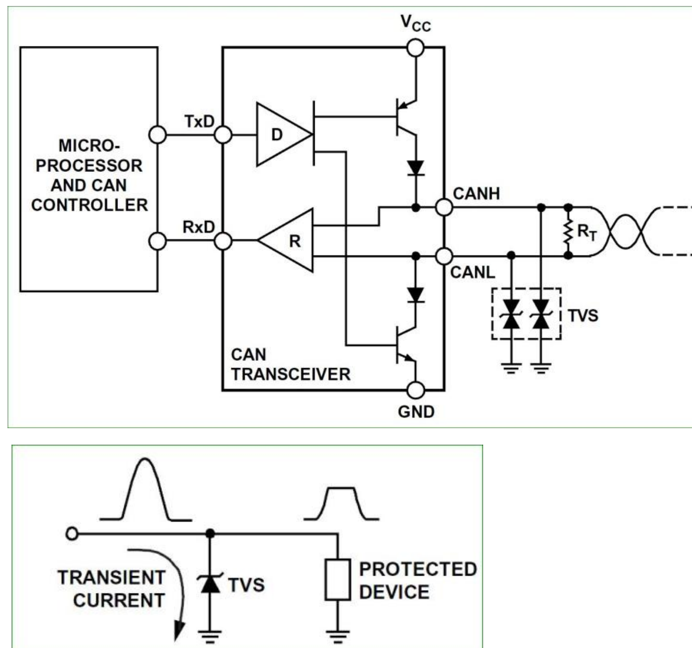

Figure 17. TVS diode Voltage protection

The transceiver works by providing a differential input and output line drive capability. This utility provides high powered line protection to the CAN protocol controller and protection from large high voltage ESD and EMI. The chip has two states, a Recessive and a Dominant state. When transmitting a message, a dominant state occurs when

the differential voltage between the CAN H and CAN L lines are above a specific voltage. A recessive state occurs when the differential voltage is below a specific voltage (e.g. 0 volts).

3.4 Transceiver Operation

Below in Figure 18, you can see the internals of the MCP2551 IC and some of its operating functions and how they are connected to the external pins.

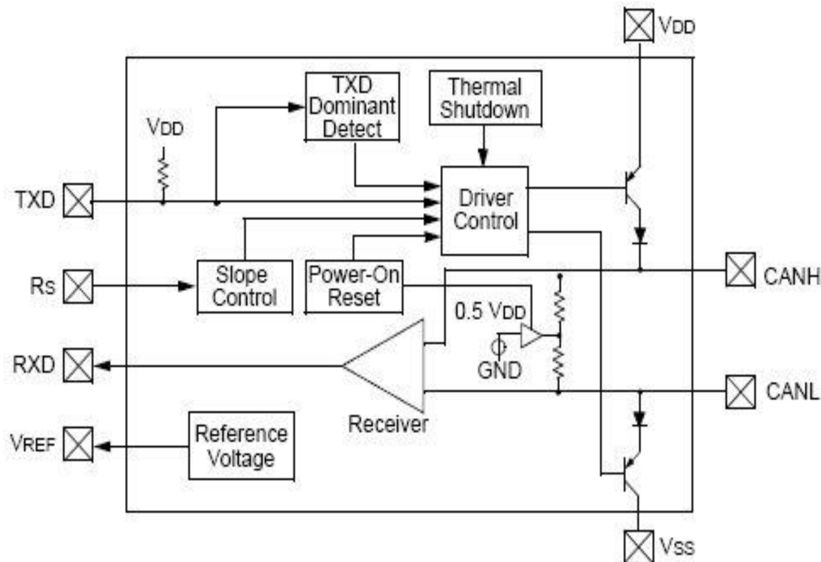

Figure 18. Internal Hierarchy

TXD and RXD connect to the CAN controller (Which can be embedded inside a microcontroller).
VDD input voltage 5v+
VSS Ground
CAN H and CAN L are the Message bus wires. (Twisted Pairs)
Rs input controls the slope or Slew rate of signal by limiting the rise and fall times of the CAN H and CAN L lines.

In the case of Mode applications (There are 3, High Speed, slope control and sleep), Rs can be driven high and will therefore put the IC into sleep mode. In sleep mode, the unit will switch off the transmitting side and only receive incoming signals (albeit at a much slower rate). The host controller can monitor the TX line and control the Rs pin to wake up on transmission signal detection. At higher data rates this wake-up period (requiring 5 micro seconds) may lose or corrupt the first CAN message sent due to

power up stabilization time requirements (It is possible to send 1 bit in a 1 microsecond period so 5 bits can be lost).

When the Rs pin is taken to ground, the chip will go to high speed mode as the slope or slew rate of the CAN bus is controlled by the level of current I_{rs} detected at the Rs pin. This is then determined by the R_{ext} external control resistor. This high-speed mode will also be the most susceptible to EMI noise and interference but allows the bus frequency to reach speeds up to 1MHz.

4 CAN Controller Protocol

4.1 ARM NXP Mbed Microcontroller

In this application we are using the Mbed NXP1768 Microcontroller with an embedded CAN controller that can support the CAN 2.0B and 2.0A formats. The NXP LPC1768 supports 2 physical CAN BUS Lines even though only one is shown on the Mbed breakout board pinout.

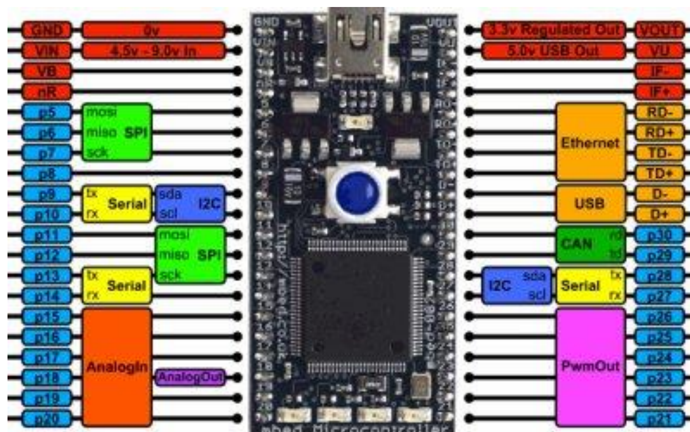

Figure 19. Mbed NXP LPC1768 Microcontroller Development Board.

The Mbed development platform in conjunction with ARM have created a specialized compiler that enables fast development. This embedded software environment centers on the online compiler that has a wealth of libraries for various functions within the NXP MCU. The API that we are implementing in this case is CAN. The process of developing the embedded software is as shown in Figure 20.

Below we see how the User code is generated in the online complier and then a compiled (.BIN) file is flashed to the onboard EEPROM via USB.

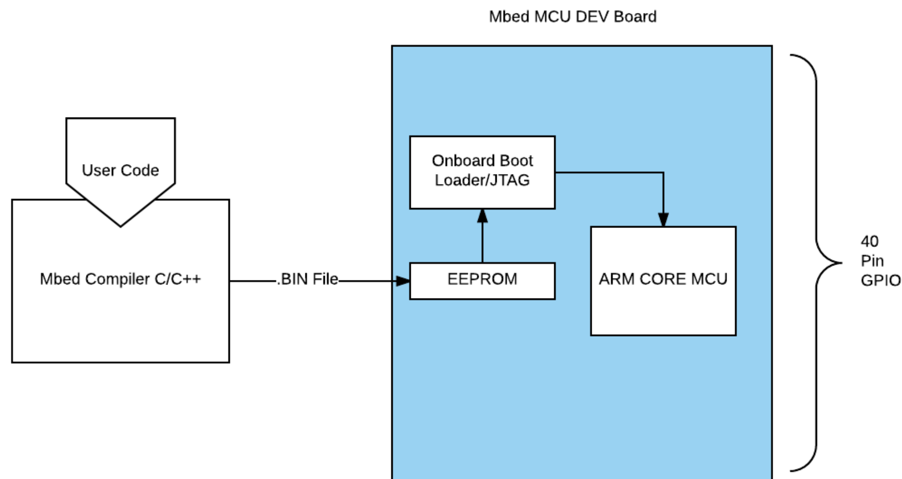

Figure 20. Mbed Software Development chain

When the system is rebooted, the bootloader creates the (.HEX) file for the MCU which in turn controls the physical I/O registers.

4.2 CANopen

CANopen protocol was developed as an industry standardization for device profiles used in automation to simplify interconnectivity between different systems. CANopen implements the OSI layer model, including a network layer. The system uses an addressing system using communication protocols and an application layer. The communication protocols allow for network control and management, device monitoring and node communication which includes the transport layer for partitioning of messages. Below this layer are the data link and physical layers which are the standard Controller Area Network (CAN).

The simple CANopen communication profiles are given in the CiA301 specifications issued by CAN in Automation. There are more specialized types of the CiA standards which are built again on top of the CANopen standard which can be found in the CAN in Automation documentation. By adhering to this standard, any new nodes can be introduced to an existing CAN bus and there would be no conflicting signals. Inverters A and B in our system were pre-configured from the manufacturer as CANopen nodes so the system nodes (1)(4) and (5) were developed to adhere to these rules of standardization.

4.2.1 CANopen Object Dictionary

CANopen nodes must contain an object dictionary holding all the standard addresses which are used to configure and communicate with the devices. Using CANopen software or software from the device manufacture being used, we can then customize the object configuration as desired and save these new settings to a Device Configuration File (DCF) which can then be uploaded via the CAN bus to apply these customized settings.

The CAN bus, which is the data link layer of the system can only transmit short packages of data. In CANopen as opposed to the Standard CAN, it uses the 11-bit ID frame by assigning 4 bits to the function code and 7 bits to the CANopen Node ID. This ID is referred to as the Communication Object Identifier (COB-ID). Standard CAN-ID mapping associates the function code (NMT, SYNC, EMCY, PDO, SDO) to the first 4 bits of the ID so all critical functions of communication have priority on the network. NMT and SDO messages are fixed and cannot be reconfigured.

4.2.2 Communication Configurations

The communication methods that can be configured in CANopen-

- Master/Slave relationship
  This method allows a master node to send or request messages to/from the slave nodes. The NMT command is an example of this type of communication method.
- Client/Server relationship
  This is achieved in the SDO messaging. The client sends an SDO message and the server responds with the data requested from a specific object index/Sub index.
- Producer/Consumer
  This method is used in the Heartbeat protocol. The producer sends data to the consumer without any specific request.

The CANopen main elements that are used in this system are listed as follows -

- Network Management (NMT) Command
- Heartbeat
- Process Data Objects (PDO's (RPDO/TPDO))
- Service Data Objects (SDO's)

4.2.3 Network Management Protocols (NMT)

This form or communication is to issue changes to the devices state machine. It can start or stop devices and detect node boot up and errors upon the bus.

The Module Control Unit is used by NMT commands to change the state of the devices. The CAN message frame ID of this protocol (COD-ID) is always zero, this means the function code bits are Zero, and the ID is Zero. This means that it is general broadcast at highest priority. The ID of the recipient node of this command is embedded in the data frame of the message at the location of Data byte 1 (second byte in the data frame). If data byte 1 is set to Zero, then the message with be received by all nodes on the network.

Upon power up of a node, the node will enter a state of initialization. After this process the node will then enter a pre-operational state.

The NMT command is a message that can control a nodes state of operation. From this point three states are achievable with this method being OPERATIONAL_STATE, PRE_OPERATIONAL, STOPPED.

COB-ID	Data Byte 0	Data Byte 1
0x000	Requested state	Addressed node

NMT command code	Meaning
0x01	Go to 'operational'
0x02	Go to 'stopped'
0x80	Go to 'pre-operational'
0x81	Go to 'reset node'
0x82	Go to 'reset communication'

Figure 21. NMT Command values.

Above in (Figure 21) we see the COB-ID and data byte values for issuing a state machine change of the devices as a general broadcast.

When the state machine is commanded to change its state then the system follows a specific sequence guideline.

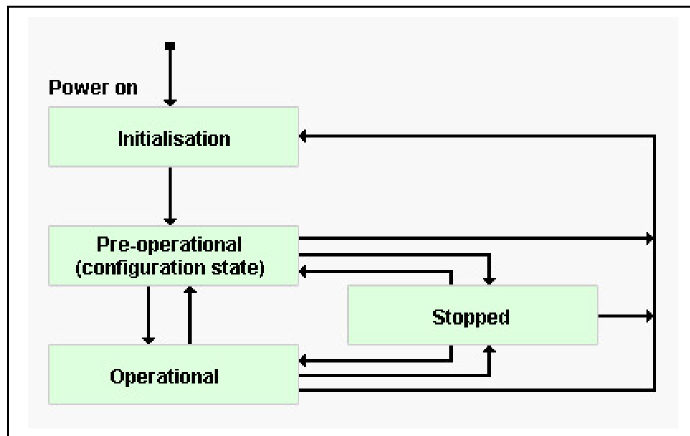

Figure 22. Node States of operation.

Different NMT commands can drive the node into various states of operation in the sequence shown above in Figure 22. After initialization when the node moves to the pre-operational state, the node can then begin to receive PDO's and SDO's.

4.2.4 Service Data Objects (SDO)

SDO protocol is used for setting or reading values from the device object libraries. These messages can be used to read what hard values are set within each device. For example, If the maximum current cutout limit for the 3-phase bridge of the inverter (A) is set to 600 Amps, then this value can be requested by sending an SDO to that device by specifying the Address index and sub-index that that information is contained within.

Byte 1					Byte 2-3	Byte 4	Byte 5-8
3 bits	1 bit	2 bits	1 bit	1 bit	2 bytes	1 byte	4 bytes
ccs=1	reserved(=0)	n	e	s	index	subindex	data

Figure 23. SDO Message Structure

The block SDO form is a newer form and allows for message segregation and desegregation to allow for larger data packets per individual message request.

In the figure above (Figure 23), we see an example of the structure of a SDO message. To initiate a download, the client sends a message containing the following data in a CAN message frame. We will now examine the frame contents –

- CCS is the command specifier for the SDO transfer, in this case it is set to 0 enabling segment download, but it can also be 1 for initiating download, 2 for initializing upload, 3 for SDO segment upload, 4 for aborting SDO transmission, 5 for SDO block upload and 6 for SDO block download.
- N is the number of bytes contained within the data frame of the message that do not contain data.
- E is set if the message is to be expedited. This is the case if all the data contents fit within the data bytes of the message.
- S if set, indicates that the data size of the message is specified in N
- Index is the object dictionary index of the data to be accessed.
- Sub index is the sub index of the object dictionary variable.

Data contains the data to be uploaded.

4.2.5 Process Data Objects (PDO)

PDO or Process Data Object protocol are what is used to send or receive real-time data to BUS nodes.

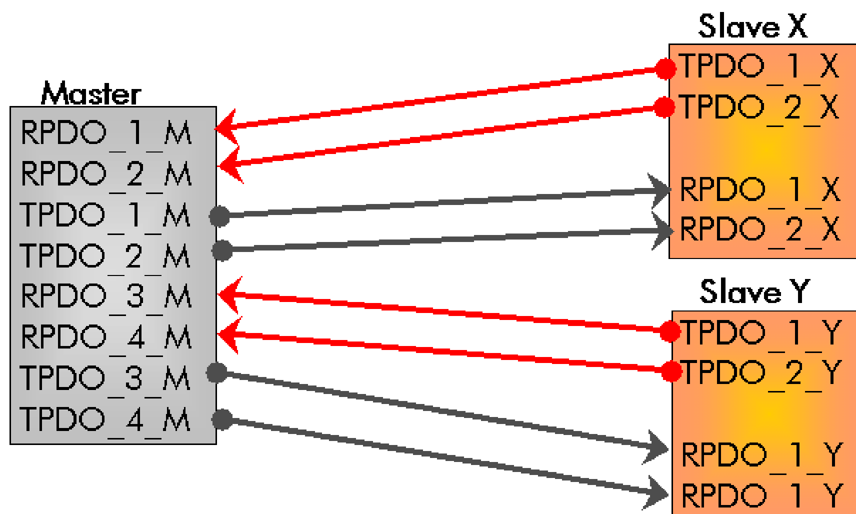

Figure 24. PDO Identifiers.

8 bytes of data can be sent per PDO message and multiple object dictionary entries can be contained within one PDO.

This can be achieved by mapping the PDO data fields when configuring the individual PDOs. There are two types of PDOs called Transmit PDO (TPDO) and Receive PDO (RPDO).

PDO object identifiers are always seen in terms of the transmitting node. The master node will transmit a receive PDO (RPDO) to the slave unit that is targeted, the (Slave) unit will then treat this message and a command to update some parameter that the message has addressed, per its mapping. With a RPDO you can send data to a device and with a TPDO you can read data from the device. This method works reciprocal to any node that sends a message to the network bus.

PDOs can be sent synchronously and asynchronously. Synchronous messages are sent after the SYNC message request and the asynchronous messages are sent when there is a message request from a node. If the TPDOs have been mapped to react to a specific trigger upon a change in value or a timer for message delivery they will behave according to their mapping.

4.2.6 Synchronization Object Protocol (SYNC)

A SYNC message from a master or producer provides a message to recipient nodes to begin carrying out their assigned synchronous tasks. Using periodic SYNC messages and standard PDOs will guarantee that sensor information and commands to system controls will work in a coordinated pattern.

5 Hybrid Vehicle Control

5.1 System Configuration

Since the purpose of this system is to control a vehicles traction drives, we will now look to the control system for the hybrid drivetrain and the mechanical configuration of the system. Except for the Internal Combustion Engine (ICE), the remaining components of the system are electromechanical and electrical components and behave as CAN nodes in respect to control.

The main ECU for controlling the vehicle was designed with the CAN hardware (Figure 12) incorporated into the PCB as shown below in Figure 25.

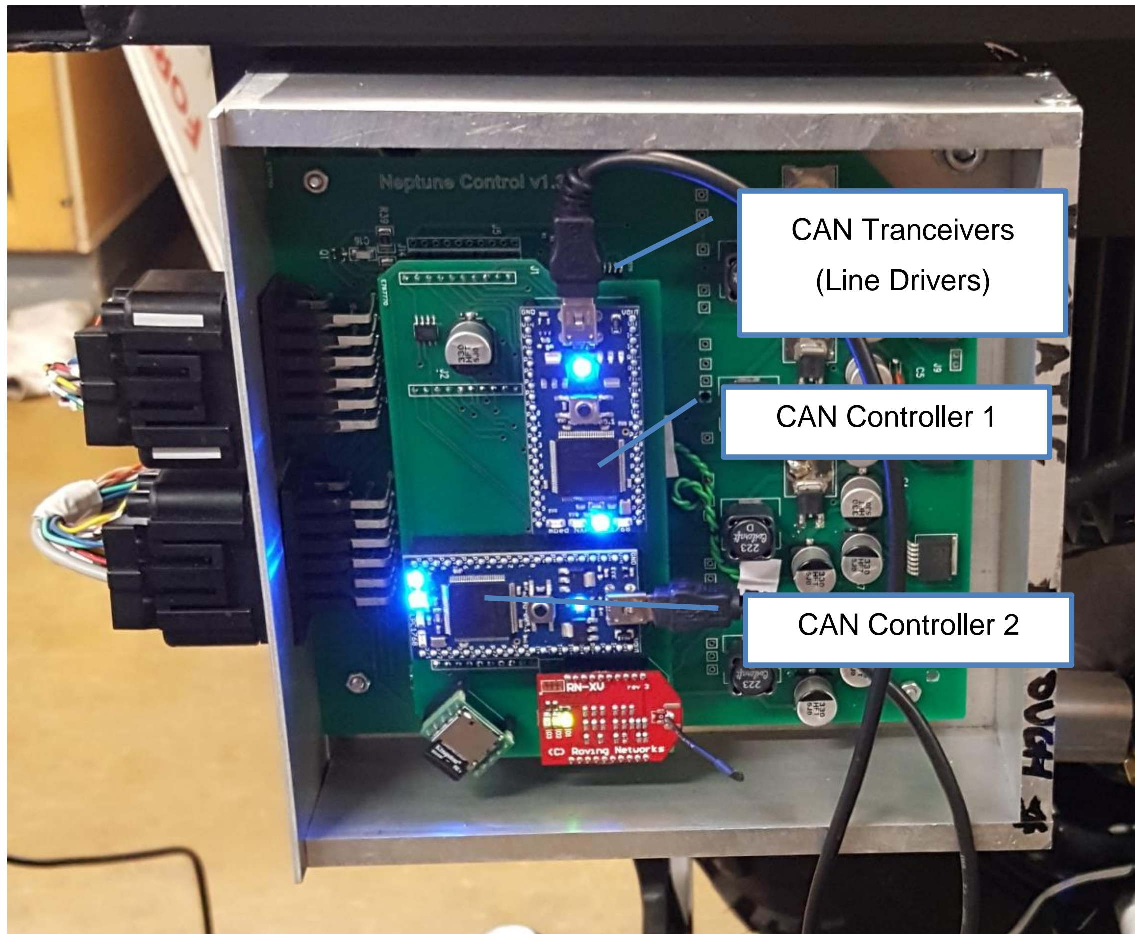

Figure 25. Master ECU for Hybrid control

Controllers for node 1 and node 2 were situated inside the same ECU as to handle critical and non-critical functions. The CAN BUS wiring consists of a twisted pair with earth shielding. Total BUS length totals 3.3 meters.

A PCB schematic can be found in Appendix (2) at the end of this document.

The CAN messaging for control will now implement the CANopen protocol (Chapter 4) for all object identification from this point forward.

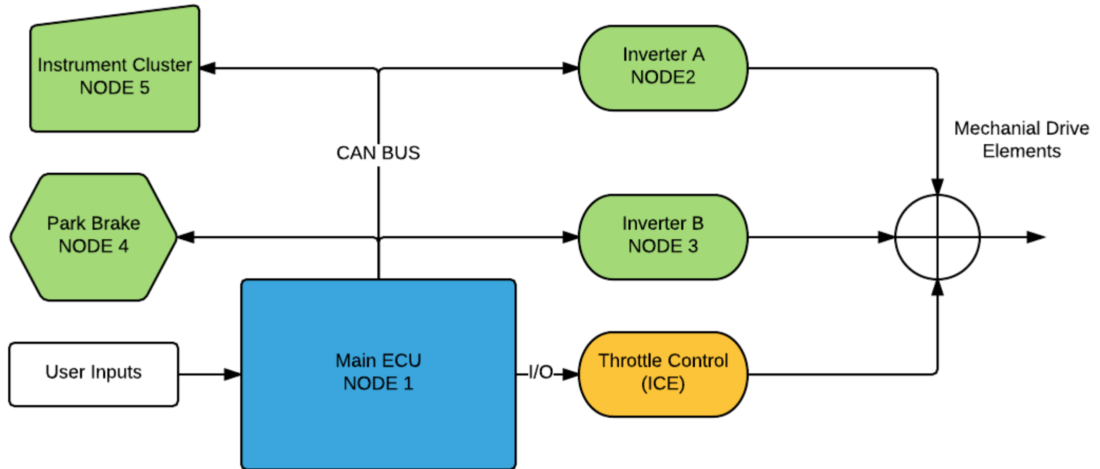

Figure 26. Electromechanical System overview

Below we will go through the necessary steps of the software development to enable a fully working CAN BUS messaging protocol for our purposes by creating a program file with the following structure:

- main.cpp (User created program)
  Included in program are header files:
- mbed.h main library inclusion (mbed)
- can.h (User created)

The mbed.h library is a purpose-built header file that is included in all programs when using the ARM Mbed IDE that will port the controller and enable all the methods and objects that are available. The (CAN.h) header file which is included allows access to the methods of the CAN class API that will handle the CAN controller 2.0A and 2.0B type protocols.

5.2 Control Code Block Diagram

Below we see how the code sequence is structured beginning from power up. The Main ECU Node is comprised of two separate MCU communicating critical data between themselves on the CAN Bus.

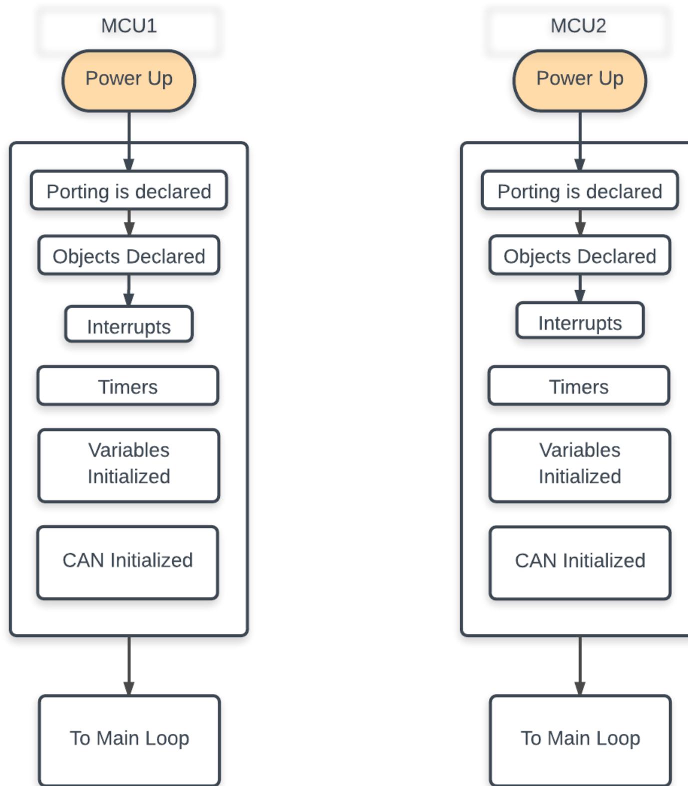

Figure 27. Code Sequence

Above in (Figure 27) the power up sequence is shown. Both MCUs operate independently as they are not synchronized.

5.3 Initialization

The first part of any MCU code design is to Identify I/O requirements of the system and to port all external I/O to their relevant objects. In our case we will be using the CAN Mbed Interface plus any User defined I/O's.

In the Code fragment below, we are creating a user defined library that we will call (e_can1.h) that has a class that we will name (e_can) to handle all the CAN protocol. Before we create the class structure we firstly include the mbed.h library which holds the general objects for the system, also we will now include the header (CAN.h) which is the CAN object library which we need to access the CAN API.

We will then initialize the CAN object, and name it can1, this is a user defined name and enter the arguments for this function which in this case requires two arguments that represent the pin numbers that the CAN interface will be using.

Note* On the LPC1768 the CAN bus designation is on pins (p30, p29) which is CAN bus 1 (There are two CAN buses available for use on this MCU).

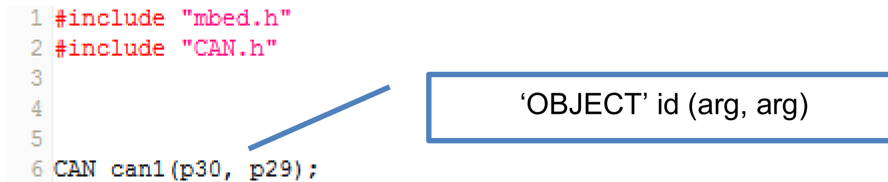

Figure 28. CAN Object Initialization.

Once this is set up we will then begin with the class structure. Here we state any public, private and protected variables that can be accessed from this class followed by the functions that are available to that class. The function (CANMessage()) is the method associated with (CAN.h) that will perform the physical sending or receiving of the CAN message frame stated in the previous chapters. The three following functions (can_send(arg,arg)), (can_receive()), (can_setup( )) are user defined functions on how to handle the various variable data from the external system and place it into and pull it out of the <CANMessage()> method.

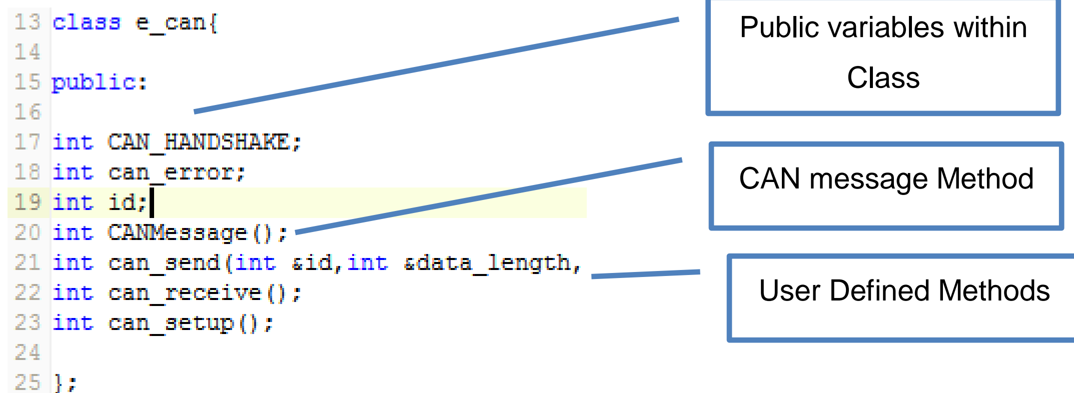

Figure 29. CAN class structure, variables and methods.

Now that we have our class and methods declared then we can begin our user defined CAN handling protocol. For any system a handling protocol can be completely designed to their own standards and wishes. If the system is going to be introduced to a network of an existing system, then the protocol must adhere to that systems parameters. For example, there are many standards that are used for industry areas such as Automotive, Aerospace, Industrial and Marine environments that these industries ad-

here to as a common mode of interfacing. CANopen (CiA) 301 specification released by (CAN in Automation) is a standard that is used in a wide array of applications relating to Automotive use, and this is the one that we will be using in this case as it is the standard for the inverters that will be implemented to control the 3 phase drives in this hybrid drivetrain. Any customized protocol will need to stay within the areas left free by the CANopen protocol as to not interfere with the system and send messages that conflict with pre-existing object names.

Note* from this point forward Hexadecimal values will be used to refer to object identification.

Before the CANopen nodes, which in our case are the inverter nodes numbered 3 (0x03h) and 4(0x04h) are enabled we will create a function for the Main ECU MCUs node1 and node 2 to perform a handshake to establish a confirmed communication connection and after this initial handshake, both main nodes will then follow a PDO structure in accordance with the CANopen standard.

In this process shown in Figure (31), the main node sends out a message on the CAN bus while all other nodes are only listening for this message. When the node with ID 0x02h hears this it then receives the 3-byte Data field (0xA6, 1, 1). The 0xA6 in the message informs the node that this is a handshake procedure, so it will then return a ACK confirmation message by sending back the data bytes {1,1} which was the data contained in the handshake message to tell main node 0x01h that it is connected and listening, and this was the data it received.

When the data returned is received correctly by the main node and compared, then the program moves on to the next node and repeats the process with a different ID for that node. Once each node has hand shaken then the program moves on to normal operation. If within this hand shake process the node fails to respond, then the program will count errors of no response. Once the error count exceeds 50 then the program will FLAG a disconnection and reset the error count again and continue to establish connection, if the error count exceeds 50 again and the FLAG has been activated then in the instance of a critical node i.e. Inverter, then the program terminates with a critical error warning.

Once the system has performed the handshake initialization and operation has been confirmed, the system then moves on to normal operation.

In the diagram below (figure 31) the handshake process sequence is explained.

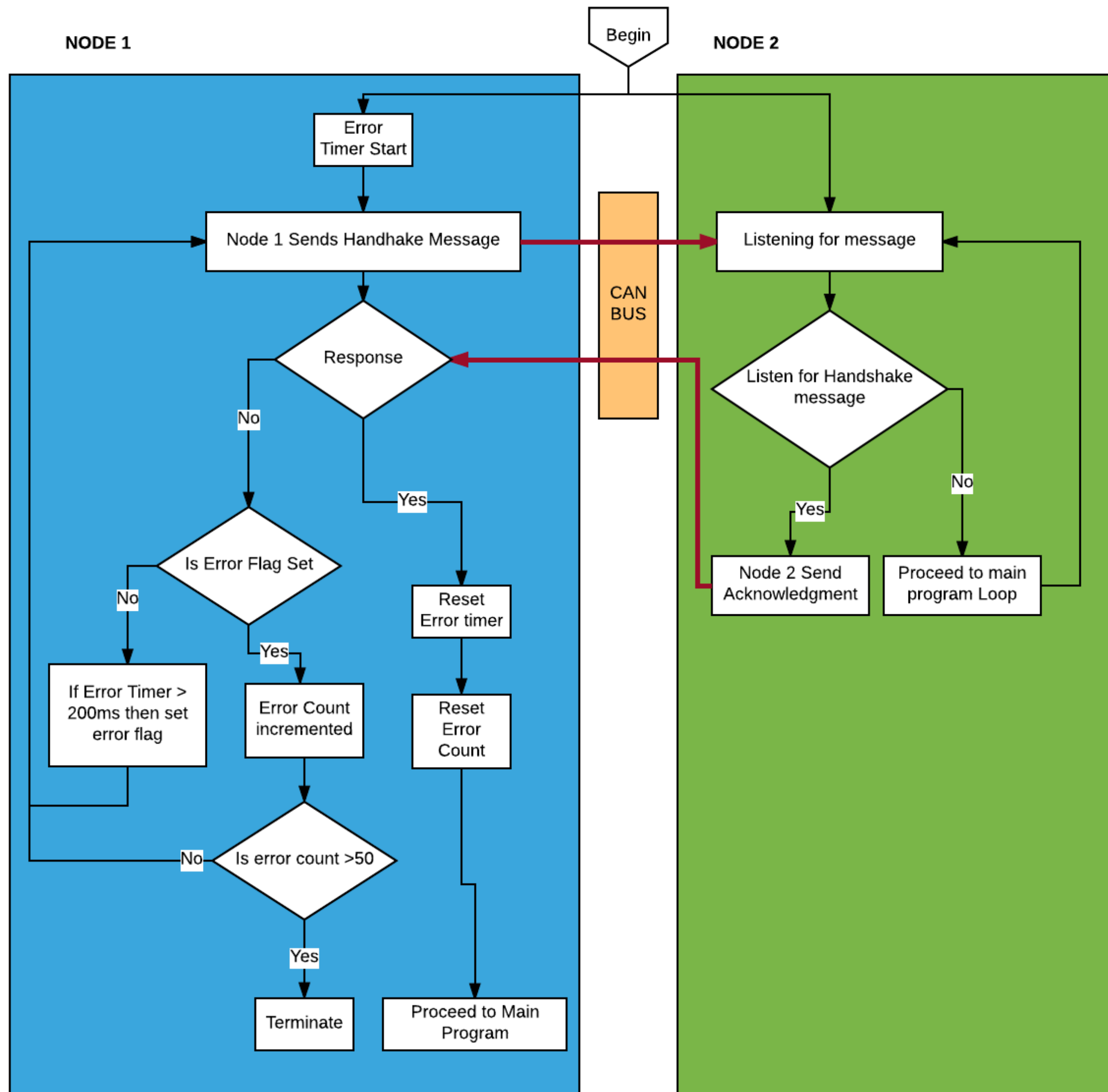

Figure 30. CAN Handshake sequence

Once the sequence is complete then the program will move on to the normal traction control code block.

5.4 Code Operation Sequence

In the code below, we send a NMT message in the CAN message format explained in the previous chapter to enable all nodes on the network. This is achieved by sending an open broadcast message ID = 0. Individual nodes can be addressed only by using the node id of that specific node if desired.

```
387    // CAN NMT (Operational) Initialise //
388    wait(2);
389    E_CAN.CANopen_send(COB_ADR0,id0,NMT_dataL,NMT_D1,NMT_D2,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
```

Figure 31. NMT Command

In this command the (*args)* passed to the message are the COB address set as 0x00h, Id = 0x00h, the length of the data field for this command is 2 Bytes and the data fields being NMT_D1=0x01h and NMT_D2=0x00h. This message will put all nodes on the network into operational state.

Following this message, the manufacturers setting require that 3 messages are sent to enable the PWM of the 3 phases of the inverters.

```
392    // Enable PWM Output Nodes 3/4  Inverters //
393    // 207
394    E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
395    wait(0.05);
396    E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E2_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
397    wait(0.05);
398    E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E3_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
399    wait(0.05);
400    // 228
401    E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
402    wait(0.05);
403    E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E2_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
404    wait(0.05);
405    E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E3_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH);
406    wait(0.05);
```

Figure 32. PWM Enabling

This messages above in Figure 32 shows the 3 messages being sent to Inverter A (Node 3) and Inverter B (Node 4). The COB_ADR3 for this message is stating that it is a RPDO message of the value 0x200h having the control bit value of 0100b, this defines the message type and how the node should respond to it. The PWM command values for this device in the data field (also being 2 Bytes as in the NMT message) are 0x06h, 0x07h and 0x0Fh, these values will enable all 3 phase lines of the H-bridge.

At this point the Inverters are then operational and active and ready to receive or send commands as required. CANopen can have many R/TPDOs, but this system has 4 PDO of both types that are pre-mapped to specific parameters inside its system. We

are using the RPDO1 message which has been mapped to the desired torque command of the inverter. The message will entail the COBID which is made up of the RPDO1 address = 0x200h and the node ID being 0x03h or 0x04h depending upon which drive that is being controlled. The message for controlling the torque of the drive is therefore shown below in Figure 33

```
68 void CANOpen_CALL_SEND(){
69
70     // Nb Torque Command
71     E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NB_torque_LSB,NB_torque_MSB);
72
73     // Ns Torque Command
74     E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NS_torque_LSB,NS_torque_MSB);
75 };
```

Figure 33. Torque Command PDO

In this message being sent to both inverters for a desired torque demand, the data field is made up of 8 data Bytes. The first Byte in the field being the Control Word (CW) 0x0Fh followed by five Zero value bytes, then the final two values are the torque desired values in the form of least significant bit and most significant bit respectively.
The value range for the torque command in this manufacturer’s case is -1000d to +1000d for forward and reverse drive so with one single byte (8 bit) data field we will have truncation so 2 bytes are necessary.

```
245         if(CON.NB_torque_direction == 1){
246                 NB_torqueP =  (CON.nb_sp_sig*8); // Torque Demand positive value
247                 NB_torqueP_MSB = NB_torqueP >> 8;
248                 NB_torqueP_LSB = NB_torqueP;
249                 NB_torque_MSB = NB_torqueP_MSB;
250                 NB_torque_LSB = NB_torqueP_LSB;
251         }
252         else{
253                 NB_torqueN = (0x8000 + (0x7FFF - ((CON.nb_sp_sig*8)-1))); // Torque demand Negative value
254                 NB_torqueN_MSB = NB_torqueN >> 8;
255                 NB_torqueN_LSB = NB_torqueN;
256                 NB_torque_LSB = NB_torqueN_LSB;
257                 NB_torque_MSB = NB_torqueN_MSB;
258         };
```

Figure 34. Bit shifting

This works as 2 bits compliment, so to create the forward and reverse control signal we implement bit-shifting. This now takes the throttle control signal of 0-100 with a forward or reverse input and formats it to a 2 byte value of (1000d) or (-1000d) in 2 bits compliment. This command is all that is needed to control the system drives for traction control.

Aside from transmitting drive commands to the inverters, we can also receive information from them as well using the TPDO. In the instance of this case we would like to receive Rpm data as well as temperatures and current usage.

Individual data can be requested at any time by sending commands to one or all nodes at the same time. In our case we are sending a SYNC message to all the nodes and have them respond with the data that they have that is mapped to that message. The message that will respond from this SYNC request contains the relevant Rpm data of that node.

By receiving the Rpm feedback data of the working system, we then have a complete cycle of critical information that is required to control the traction system of the hybrid drivetrain contained within the ATV.

6 Conclusion

It was found that using standard Analogue/Digital signals for critical function in this vehicles environment was not possible due to the electrical interference caused by the 3 phase drives. All digital signals were corrupted upon activation of the 3 phase inverter bridges. Therefore, control and feedback signals failed and created an uncontrolled and unsafe automotive control system. The CAN bus implemented allowed all signals to be transmitted around the vehicle in real-time and arrive uncorrupted.

It was found during the design process of the physical line system that the CAN Transceivers were extremely sensitive to voltage supply stability. Common mode filtering had a very good effect on the High and Low lines under heavy three phase motor interference. The next steps of this system would be to continue the code development to adhere the base code totally to the CANopen standard and enable all objects to be operational. This would enable interoperability with other hardware manufacturers that adhere to this standard.

References

1 CAN in Automation. [Online] URL: https://www.can-cia.org

2 Fast and effective embedded systems design - Applying the ARM mbed. Authors: Rob Toulson and Tim Wilmshurst; Year 2016

3 CANopen solutions. [Online] URL: https://www.canopensolutions.com

4 Embedded Networking with CAN and CANopen. Authors: Olaf Pfeiffer, Andrew Ayre and Christian Keydel; Year 2016

5 C++ Programming. [Online] URL: https://www.tutorialspoint.com/cplusplus/

6 ARM mbed Handbook. [Online] URL: https://www.mbed.com

7 Mentor Graphics PCB Design Software. [Online] URL: https://www.pads.com

8 [Online] URL: https://www.wikipedia.org/wiki/CAN_bus

9 [Online] URL: https://www.microchip.com

10 [Online] URL: https://www.microchip.com/MCP2551

11 [Online] URL: https://www.ni.com/white-paper/2732/en/

Main Program Control Code

```
#include "mbed.h"
#include "Init.h"
#include "e_can1.h"
#include "Throttle.h"
#include "main.h"
Timer SYS_LOOP;
Ticker CAN_WRITE_T1;
Ticker UPDATE_T2;
Ticker CAN_READ_T3;
Ticker TPS_CHECK_T4;
Ticker CAN_WRITE_T5;
Ticker CAN_WRITE_T6;
Ticker CAN_WRITE_T7;
Ticker CAN_SEND_T8;
/// General (I/O) //
DigitalOut CAN_KILL(p12);
DigitalOut REG_KILL_O(p8);
DigitalOut NS_FWD_O(p20);
DigitalOut NB_FWD_O(p10);
DigitalOut NS_INVERTER_O(p15);
DigitalOut NB_INVERTER_O(p16);
DigitalOut NS_REGEN_O(p19);
PwmOut NB_REGEN_O(p26);
DigitalOut NB_REV_O(p24);
DigitalOut NS_REV_O(p25);
InterruptIn FWD_REV_I(p9);
DigitalIn BRAKE_I(p11);
DigitalIn USER_MODE_I(p7);
AnalogIn TPS_I(p17);
AnalogIn THROTTLE_I(p18);
PwmOut NB_SPD_SIG_O(p21);
PwmOut NS_SPD_SIG_O(p22);
PwmOut SERVO_SIG_O(p23);
/// Lib Object Definitions //
ecu_init ECU;
e_can E_CAN;
t_con CON;
void CAN_CALL_PARK_LOCK(){
//park_direction = !park_direction;
for(int k=0;k<2;k++){
E_CAN.can_send(COB_ADR3,id5,data_length,CON.park_COMMAND,CON.park_direction,CON.ns_rpm,d
ata4,data5,data6,data7,data8); //
};
};
void CAN_CALL_RECEIVE(){
E_CAN.can_recieve(id2,data_length,data1,data2,data3,data4,data5,data6,data7,data8);
};
void CAN_CALL_SEND(){
E_CAN.can_send(COB_ADR1,id2,data_length,data1,data2,data3,data4,data5,data6,data7,data8)
;
};
void CAN_CALL_SEND2(){
E_CAN.can_send(COB_ADR2,id2,data_length,data9,data10,data11,data12,data13,data14,data15,
data16);
};
// CANopen Master control TPDO Message
void CANOpen_CALL_SEND(){
// Nb Torque Command
E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NB_torque_LS
B,NB_torque_MSB);
// Ns Torque Command
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NS_torque_LS
B,NS_torque_MSB);
};
void CANOpen_SYNC(){
E_CAN.CANopen_send(COB_SYNC,id0,SYNC_DL,SYNC_COM,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,Zer
oH);
};
void CAN_CALL_INIT(){
E_CAN.can_init_send(id2,init_data_length,data_ID);
```

```
E_CAN.can_init_send(id3,init_data_length,data_ID);
E_CAN.can_init_send(id4,init_data_length,data_ID);
};
void fwd_rev_call(){
ECU.t_fwd_rev_in();
};
void IO_UPDATE(){
//CAN_CALL_RECEIVE(); // Recieve Variable Information from Node 2 //
REG_KILL_O = CON.reg_kill;
NS_FWD_O = CON.ns_fwd;
NB_FWD_O = CON.nb_fwd;
NS_REV_O = CON.ns_rev;
NB_REV_O = CON.nb_rev;
NS_INVERTER_O = CON.ns_inverter;
NB_INVERTER_O = CON.nb_inverter;
NS_REGEN_O = CON.ns_regen;
NB_REGEN_O = CON.nb_regen;
CON.user_mode = USER_MODE_I.read();
CON.fwd_rev_in = FWD_REV_I.read();
CON.brake_in = BRAKE_I.read();
CON.tps_in = TPS_I.read();
// Filter Throttle Input //
t_sample_v = THROTTLE_I.read()+0.005;
if ((t_sample_v) > (CON.throttle_in + 0.5)){
//pc.printf("\n\n\rHIGH NOISE Value: %.4f Previous Value:
%.4f\n\r",t_sample_v,CON.throttle_in+0.005);
wait(0.001);
CON.throttle_in = (((THROT-
TLE_I.read()+0.005)+(THROTTLE_I.read()+0.005))/2.00);//(t_sample_v);
//pc.printf("\n\n\rDriven Throttle Value: %.4f\n\r",CON.throttle_in);
CON.throttle_filter = ((CON.throttle_in * 100) - 21)*1.3; // Throttle noise filtering
CON.thr_err_c_high++;
CON.fault_v = t_sample_v;
t_sample_v = 0.0;
}
else if ((t_sample_v) > (CON.throttle_in + 0.05)&&(t_sample_v) <= (CON.throttle_in +
0.5)){
//pc.printf("\n\n\rLOW NOISE Value: %.4f Previous Value:
%.4f\n\r",t_sample_v,CON.throttle_in+0.005);
wait(0.001);
CON.throttle_in = (((THROT-
TLE_I.read()+0.005)+(THROTTLE_I.read()+0.005))/2.00);//(t_sample_v);
//pc.printf("\n\n\rDriven Throttle Value: %.4f\n\r",CON.throttle_in);
CON.throttle_filter = ((CON.throttle_in * 100) - 21)*1.3; // Throttle noise filtering
CON.thr_err_c_low++;
CON.fault_v = t_sample_v;
t_sample_v = 0.0;
}
else if(t_sample_v < 0.21){
CON.throttle_filter = 0;
t_sample_v = 0.0;
}
else{
CON.throttle_in = (t_sample_v);
CON.throttle_filter = ((CON.throttle_in * 100)- 21)*1.3; // Throttle noise filtering
t_sample_v = 0.0;
};
// End of Throttle Filter
SERVO_SIG_O = CON.servo_sig;
NS_SPD_SIG_O = CON.ns_sp_sig;
NB_SPD_SIG_O = CON.nb_sp_sig;
// Rpm Data from CAN (207/228) *** Remember Direction of drive is Hex inverted ****
if(E_CAN.value30 == 255){ // value 30 is MSB True for inverter 3rd byte
NB_rpm_MSB = ((255 - (E_CAN.value29))*255); // HALL SENSOR: E_CAN.value3, INVERTER:
E_CAN.value25-32
NB_rpm_LSB = (255 - E_CAN.value28); // HALL SENSOR: E_CAN.value4, INVERTER:
E_CAN.value25-32
NB_rpm = (NB_rpm_MSB + NB_rpm_LSB);
if(NB_rpm < 5000){
CON.nb_rpm = NB_rpm; // Torque demand Negative value
}
else{
pc.printf("NB Rpm Out of range\n\r");
};
```

```
}
else{
NB_rpm_MSB = ((E_CAN.value29)*255); // HALL SENSOR: E_CAN.value3, INVERTER:
E_CAN.value25-32
NB_rpm_LSB = E_CAN.value28; // HALL SENSOR: E_CAN.value4, INVERTER: E_CAN.value25-32
NB_rpm = (NB_rpm_MSB + NB_rpm_LSB);
if(NB_rpm < 5000){
CON.nb_rpm = NB_rpm; // Torque demand Negative value
}
else{
pc.printf("NB Rpm Out of range\n\r");
};
};
if(E_CAN.value22 == 255){ // value 22 is MSB True for inverter 3rd byte
NS_rpm_MSB = ((255 - (E_CAN.value21))*255); // HALL SENSOR: E_CAN.value1, INVERTER:
E_CAN.value17-24
NS_rpm_LSB = (255 - (E_CAN.value20)); // HALL SENSOR: E_CAN.value2, INVERTER:
E_CAN.value17-24
NS_rpm = (NS_rpm_MSB + NS_rpm_LSB);
if(NS_rpm < 5000){
CON.ns_rpm = NS_rpm; // Torque demand Negative value
}
else{
pc.printf("NS Rpm Out of range\n\r");
};
}
else{
NS_rpm_MSB = ((E_CAN.value21) * 255); // HALL SENSOR: E_CAN.value1, INVERTER:
E_CAN.value17-24
NS_rpm_LSB = (E_CAN.value20); // HALL SENSOR: E_CAN.value2, INVERTER: E_CAN.value17-24
NS_rpm = (NS_rpm_MSB + NS_rpm_LSB);
if(NS_rpm < 5000){
CON.ns_rpm = NS_rpm; // Torque demand Negative value
}
else{
pc.printf("NS Rpm Out of range\n\r");
};
};
sc++;
// Calculate Na with algorithm //
if(sc == 1){
NA_rpm = ((((CON.nb_rpm*1.14)*3.24)-(CON.ns_rpm*2.24))*1.194);
if(NA_rpm < 0.0){
CON.na_rpm = (NA_rpm *(-1));
}
else{
CON.na_rpm = NA_rpm;
};
sc=0;
};
pc.printf("Ns Rpm: %d (%d,%d,%d) Nb Rpm: %d (%d,%d,%d) Na Rpm: %d\n\r",CON.ns_rpm,
E_CAN.value20,E_CAN.value21,E_CAN.value22,CON.nb_rpm,E_CAN.value28,E_CAN.value29,E_CAN.v
alue30,CON.na_rpm);
//}
//else{
// pc.printf("Na Rpm NOT UPDATED******* Messages out of sync *************** Na Rpm:
%d\n\r",CON.na_rpm);
//};
// CAN Variable Update //
data1 = CON.drive_mode;
data2 = (CON.throttle_filter);
data3 = (CON.ns_sp_sig*100);
data4 = (CON.nb_sp_sig*100);
data5 = (CON.servo_sig*100); //-0.892)*150);
data6 = (CON.tps_in*33);
data7 = CON.fwd_rev_in;
data8 = CON.ns_regen;
data9 = ((((CON.ns_rpm/2)/4)*60*2.01)/1000); // Speed calculation from rpm to km/h //
data10 = CON.nb_fwd ;
data11 = CON.ns_rev;
data12 = CON.nb_rev;
data13 = CON.ns_inverter;
data14 = CON.nb_inverter;
data15 = CON.nb_regen;
```

```
data16 = CON.eng_kill;
// CAN Message Torque Formatting //
if(CON.NB_torque_direction == 1){
NB_torqueP = (CON.nb_sp_sig*8); // Torque Demand positive value
NB_torqueP_MSB = NB_torqueP >> 8;
NB_torqueP_LSB = NB_torqueP;
NB_torque_MSB = NB_torqueP_MSB;
NB_torque_LSB = NB_torqueP_LSB;
}
else{
NB_torqueN = (0x8000 + (0x7FFF - ((CON.nb_sp_sig*8)-1))); // Torque demand Negative val-
ue
NB_torqueN_MSB = NB_torqueN >> 8;
NB_torqueN_LSB = NB_torqueN;
NB_torque_LSB = NB_torqueN_LSB;
NB_torque_MSB = NB_torqueN_MSB;
};
if(CON.NS_torque_direction == 1){
NS_torqueP = (CON.ns_sp_sig*10); // Torque Demand positive value
NS_torqueP_MSB = NS_torqueP >> 8;
NS_torqueP_LSB = NS_torqueP;
NS_torque_MSB = NS_torqueP_MSB;
NS_torque_LSB = NS_torqueP_LSB;
}
else{
NS_torqueN = (0x8000 + (0x7FFF - ((CON.ns_sp_sig*10)-1))); // Torque demand Negative
value
NS_torqueN_MSB = NS_torqueN >> 8;
NS_torqueN_LSB = NS_torqueN;
NS_torque_LSB = NS_torqueN_LSB;
NS_torque_MSB = NS_torqueN_MSB;
};
};
/*TPS Loop Throttle Check */
void tps_checker(){
CON.tps_r();
};
void motor_test(){
if(pc.readable()){
if(pc.getc()== 'q'){
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH)
;
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,
ZeroH);
E_CAN.CANopen_send(COB_ADR0,id0,NMT_dataL,NMT_PREOP,NMT_D2,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
motor_test_kill = 0;
};
if(pc.getc()== 'r'){
E_CAN.CANopen_send(COB_ADR0,id0,NMT_dataL,NMT_D1,NMT_D2,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,Ze
roH);
// Enable PWM Output //
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,
ZeroH);
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E2_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E3_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
motor_test_kill = 1;
};
if(pc.getc()=='+'){
drive_f = (drive_f+1);
drive_r = (drive_r+1);
};
if(pc.getc()=='-'){
drive_f = (drive_f-1);
drive_r = (drive_r-1);
};
};
if(motor_test_kill == 1){
if(CON.fwd_rev_in == 1){
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NB_torque_LS
B,NB_torque_MSB);
pc.printf("Torque_V: %d\n\r",NB_torque_LSB);
```

```
}
else if (CON.fwd_rev_in == 0){
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,CW,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,NB_torque_LS
B,NB_torque_MSB);
pc.printf("Torque_V: %d\n\r",NB_torque_LSB);
}
else{
};
};
};
void servo_test(){
float i = 0.0;
// Sweep Servo Forward and back to check operation.
for (i=0;i<50;){
CON.servo_sig = 0.956 - (i/1000.0);
pc.printf("SERVO SIG: %.3f test number: %.0f\n\r",CON.servo_sig,i);
SERVO_SIG_O = CON.servo_sig;
// Call TPS Check Function here when TPS Enable
//tps_r(float &thr, float &tps)
wait(0.1);
i= (i+1);
};
for (i=0;i<50;){
CON.servo_sig = 0.906 + (i/1000.0);
pc.printf("SERVO SIG: %.3f test Number: %.0f\n\r",CON.servo_sig,i);
SERVO_SIG_O = CON.servo_sig;
// Call TPS Check Function Here when TPS Enabled
//tps_r(float &thr, float &tps)
wait(0.1);
i=(i+1);
};
};

int main() {

pc.baud(460800);
pc.printf("\n\rP-LSB: %02X P-MSB: %02X\n\r",NB_torque_LSB,NB_torque_MSB);
pc.printf("\n\rN-LSB: %02X N-MSB: %02X\n\r",NB_torque_LSB,NB_torque_MSB);

// Variable Set //
CAN_KILL = 0; // CAN Transceiver Control (On = 0/Off = 1)
l=0; // Diagnostics Test Loop Counter
E_CAN.tpdo1_node3_message = 0;
E_CAN.tpdo1_node4_message = 0;
//TPS_CHECK_T4.attach( &tps_checker,0.1); // 10 Hz

/// PWM Output Period Times //
SERVO_SIG_O.period_ms(20); // Servo period //
NB_SPD_SIG_O.period_ms(20); // Speed Signal period //
NS_SPD_SIG_O.period_ms(20); // Speed Signal Period //
NB_REGEN_O.period_ms(20); // Nb speed GHM Control //
UPDATE_T2.attach(&IO_UPDATE,0.03); // working = 0.05 // 10 Hz
// CAN NMT (Operational) Initialise //
wait(2);
E_CAN.CANopen_send(COB_ADR0,id0,NMT_dataL,NMT_D1,NMT_D2,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,Ze
roH);
//pc.printf("1\n\r");
wait(0.05);
// Enable PWM Output Nodes 3/4 Inverters //
// 207
//pc.printf("2\n\r");
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,
ZeroH);
wait(0.05);
//pc.printf("3\n\r");
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E2_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
wait(0.05);
//pc.printf("4\n\r");
E_CAN.CANopen_send(COB_ADR3,id3,RPDO_dataL,PWM_E3_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
wait(0.05);
// 228
//pc.printf("5\n\r");
```

```
E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,
ZeroH);
wait(0.05);
//pc.printf("6\n\r");
E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E2_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
wait(0.05);
//pc.printf("7\n\r");
E_CAN.CANopen_send(COB_ADR3,id4,RPDO_dataL,PWM_E3_D1,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH,ZeroH
,ZeroH);
pc.printf("Starting ticker messaging\n\r");
wait(0.05);
/// Function Call Tickers //
CAN_WRITE_T1.attach(&CAN_CALL_SEND,0.11); // 25 Hz
CAN_WRITE_T6.attach(&CAN_CALL_SEND2,0.1); // 25 Hz
CAN_WRITE_T5.attach(&CANOpen_CALL_SEND,0.09); // 100 Hz
CAN_READ_T3.attach(&CAN_CALL_RECEIVE,0.04); // 14 Hz
CAN_WRITE_T7.attach(&CAN_CALL_PARK_LOCK,0.15); // Sample Park Command
CAN_SEND_T8.attach(&CANOpen_SYNC,0.09); // ******************Attach these to new varia-
bles ***************
// System Timer //
//SYS_LOOP.start();
wait(4);
if((CON.throttle_filter >= 90) && (CON.brake_in == 1)){
CON.gen_mode();
};
while(1){
// Program Calibration Functions //
//servo_test();
//motor_test();
//CON.eng_kill = 1;
//pc.printf("********CAN DATA********** NS: %.1f NB: %.1f NA:
%.1f\n\r",CON.ns_rpm,CON.nb_rpm,CON.na_rpm);
//CON.NB_torque_direction = 0;
//CON.nb_sp_sig = CON.throttle_filter;
// ***********Start of program*********** //
CON.drive_selector(CON.battery_sense,CON.user_mode,CON.eng_kill,CON.ns_rpm,CON.brake_in,
CON.throttle_filter);
//tme = SYS_LOOP.read();
}; // End of Main While Loop //
}; // End of Main Program
```

```
// Header File //
#include "mbed.h"
//******************************** CAN COB_ID's ***************************
int COB_ADR0 = 0x000;
int COB_ADR1 = 0x380; // Message 1 Len 8 bytes
int COB_ADR2 = 0x480; // Message 2
int COB_ADR3 = 0x200; // CAN Open Rpdo1 Func Bits 0100b
int COB_ADR4 = 0x300; // CAN Open Rpdo2 Func Bits 1000b
int COB_ADR5 = 0x180; // CAN Open Tpdo1 Func Bits 0011b
int COB_ADR6 = 0x280; // CAN Open Tpdo2 Func Bits 0101b
int COB_SYNC = 0x080;
//**************************** Can Data Node ID // ****************************
int id0 = 0x00;
int id1 = 0x01; // Main MCU
int id2 = 0x02; // Diagnostics 1st message
int id3 = 0x03; // Park Lock
int id4 = 0x04; // Screen Message
int id5 = 0x05;
int data_length = 8;
int CANopen_data_length = 0x01;
int init_data_length = 1;
int data_length1 = 1;
int CANopen_data_length2 = 8;
int NMT_dataL = 0x02;
int RPDO_dataL = 0x08;
int PWM_EN_L = 0x02;
int SYNC_DL = 0x01;
//****************************** Can Data Test Variables // **************************
int CW = 0x0F; // Control Signal
int data1 = 0x00;
int data2 = 0x00; // Control Signal (Control Word)
```

```
int data3 = 0;
int data4 = 0;
int data5 = 0;
int data6 = 0;
float data7 = 0.0;
int data8 = 0;
int data9 = 0;
int data10 = 0;
int data11 = 0;
int data12 = 0;
int data13 = 0;
int data14 = 0;
float data15 = 0.0;
int data16 = 0;
int ENG_KILL = 0;
int data_ID;
int dummy;
int NMT_D1 = 0x01;
int NMT_D2 = 0x00;
int NMT_PREOP = 0x80;
int ZeroH = 0x00;
int PWM_E_D1 = 0x06;
int PWM_E2_D1 = 0x07;
int PWM_E3_D1 = 0x0F;
int Vel_LSB = 0x00;
int SYNC_COM = 0x00;
int drive_r = 15;
int drive_f = 15;
// Torque demand values //
uint16_t NB_torqueN;
int8_t NB_torqueN_MSB;
int8_t NB_torqueN_LSB;
uint16_t NB_torqueP;
int8_t NB_torqueP_MSB;
int8_t NB_torqueP_LSB;
uint16_t NS_torqueN;
int8_t NS_torqueN_MSB;
int8_t NS_torqueN_LSB;
uint16_t NS_torqueP;
int8_t NS_torqueP_MSB;
int8_t NS_torqueP_LSB;
int NB_torque_MSB;
int NB_torque_LSB;
int NS_torque_LSB;
int NS_torque_MSB;
// Throttle Sampling Variables //
float t_sample[5];
float t_sample_v;
int y;
float throttle_read_sample;
// Initialize Variables //
int ns_r = 0; // Attach to NS_RPM
int entry_mode = 0; // Attach to USER_MODE
int save_mode = 0; // Attach to BAT_LEVEL
float tme;
int l;
int sc;
// Park Lock sample Variables //
int motor_test_kill = 1;
int NB_rpm;
int NB_rpm_MSB;
int NB_rpm_LSB;
int NS_rpm;
int NS_rpm_MSB;;
int NS_rpm_LSB;
int NA_rpm;
int NA_rpm_MSB;
int NA_rpm_LSB;
// End of header file //
```

```
// CAN Messaging Class //


#include "mbed.h"
#include "CAN.h"
CAN can1(p30, p29);
DigitalOut L1(LED1);
DigitalOut L2(LED2);
DigitalOut L3(LED3);
DigitalOut L4(LED4);
Timer CAN_msg_t;
/** CAN Message Object */
CANMessage msg;
/** CAN Class */
class e_can{
public:
int CAN_HANDSHAKE;
int can_error;
int id;
int ident;
int value1;
int value2;
int value3;
int value4;
int value5;
int value6;
float value7;
int value8;
int value9;
int value10;
int value11;
int value12;
int value13;
int value14;
float value15;
int value16;
// TPDO1 NS207 Velocity
int value17;
int value18;
int value19;
int value20;
int value21;
int value22;
int value23;
int value24;
// TPDO1 NB228 Velocity
int value25;
int value26;
int value27;
int value28;
int value29;
int value30;
int value31;
int value32;
int COB_ID;
char data;
char data_s;
int flag;
int data_length;
int tpdo1_node3_message;
int tpdo1_node4_message;
/** Setup Function */
int can_setup();
/** Message Object */
int CANMessage();
/** Meesage Send*/
int can_send(int &tpo_address,int &id,int &data_length,int &data1,int &data2,int
&data3,int &data4,int &data5,int &data6,float &data7,int &data8);
/** Screen Send */
int can_send_screen(int &id,int &data_length,int &form,int &data2,int &data3,int
&data4,int &data5,int &data6,float &data7,int &data8);
/** Message Recieve*/
int can_recieve(int &id,int &data_length,int &data1,int &data2,int &data3,int &data4,int
&data5,int &data6,float &data7,int &data8);
```

```
/** CAN Init */
int can_init_send(int &id,int &data_length,int &data1);
int CANopen_send(int &tpo_address,int &id,int &data_length,int &data1,int &data2,int
&data3,int &data4,int &data5,int &data6,int &data7,int &data8);
/** Private Variables */
private:
bool msg_flag;
int err;
};
/** CAN BUS Frequency setup */
int e_can::can_setup(){
can1.frequency(500000);
return 0;
};
// Can Message Function SEND/RECIEVE //
int e_can::can_send(int &tpo_address,int &id,int &data_length,int &data1,int &data2,int
&data3,int &data4,int &data5,int &data6,float &data7,int &data8){
CAN_msg_t.start();
L1 = !L1;
char data[] = {data1,data2,data3,data4,data5,data6,data7,data8}; //1B Control Word,2B
Control Word,3B:4B:5B:6B,B7 torque(lsb),8B torque(msb)
COB_ID = (tpo_address + id);
//NMT Message 0x000,0x01,0x00
//Message Test Values 0x200+1 , 0x0F,0x00,0,0,0,0,0x1A,0x00
//set the empty object //
msg.id = COB_ID;
msg.len = data_length;
msg.format = CANStandard;
msg.type = CANData;
msg.data[0] = data[0];
msg.data[1] = data[1];
msg.data[2] = data[2];
msg.data[3] = data[3];
msg.data[4] = data[4];
msg.data[5] = data[5];
msg.data[6] = data[6];
msg.data[7] = data[7];
// Write message //
pc.printf("Sending Message to Node ID: 0x%03X DB1: %02X DB2: %02X DB3: %02X DB4: %02X
DB5: %02X DB6: %02X DB7: %02X DB8:
%02X\n\r",msg.id,msg.data[0],msg.data[1],msg.data[2],msg.data[3],msg.data[4],msg.data[5]
,msg.data[6],msg.data[7]);
can1.write(msg);
msg_flag = 0;
return 0;
};
// Most Important // *********************************
int e_can::CANopen_send(int &tpo_address,int &id,int &data_length,int &data1,int
&data2,int &data3,int &data4,int &data5,int &data6,int &data7,int &data8){
L2 = !L2;
char data[] = {data1,data2,data3,data4,data5,data6,data7,data8};
COB_ID = (tpo_address + id);
//set the empty object //
msg.id = COB_ID;
msg.len = data_length;
msg.format = CANStandard;
msg.type = CANData;
msg.data[0] = data[0];
msg.data[1] = data[1];
msg.data[2] = data[2];
msg.data[3] = data[3];
msg.data[4] = data[4];
msg.data[5] = data[5];
msg.data[6] = data[6];
msg.data[7] = data[7];
// Write message //
pc.printf("Sending Message to Node ID: 0x%03X DB1: %02X DB2: %02X DB3: %02X DB4: %02X
DB5: %02X DB6: %02X DB7: %02X DB8:
%02X\n\r",msg.id,msg.data[0],msg.data[1],msg.data[2],msg.data[3],msg.data[4],msg.data[5]
,msg.data[6],msg.data[7]);
can1.write(msg);
msg_flag = 0;
return 0;
};
```

```
int e_can::can_send_screen(int &id,int &data_length,int &form,int &data2,int &data3,int
&data4,int &data5,int &data6,float &data7,int &data8){
char data_s[] = {form,data2,data3,data4,data5,data6,data7,data8};
L3 = !L3;
// set the empty object //
msg.id = id;
msg.len = data_length;
msg.format = CANExtended;
msg.type = CANData;
msg.data[0] = data_s[0];
msg.data[1] = data_s[1];
msg.data[2] = data_s[2];
msg.data[3] = data_s[3];
msg.data[4] = data_s[4];
msg.data[5] = data_s[5];
msg.data[6] = data_s[6];
msg.data[7] = data_s[7];
// Write message //
can1.write(msg);
return 0;
};
int e_can::can_recieve(int &id,int &data_length,int &data1,int &data2,int &data3,int
&data4,int &data5,int &data6,float &data7,int &data8){
//pc.printf("\n\n\rReading for CAN message\n\r");
if(can1.read(msg)){
L4 = !L4;
if(msg.id == 0x581){
value1 = msg.data[0]; // 207 Rpm MSB
value2 = msg.data[1]; // 207 Rpm LSB
value3 = msg.data[2]; // 228 Rpm MSB
value4 = msg.data[3]; // 228 Rpm LSB
value5 = msg.data[4]; // Rotax Rpm MSB
value6 = msg.data[5]; // Rotax Rpm LSB
value7 = msg.data[6]; // CTS Drives
value8 = msg.data[7]; // CTS Inverters
pc.printf("CAN Message Received: COB_ID=0x%03X data_length = %d data =
ns_m:%d,ns_l:%d,nb_m:%d,nb_l:%d,na_m:%d,na_l:%d
cts:%.1f,%d\n\r",msg.id,msg.len,value1,value2,value3,value4,value5,value6,value7,value8)
;
}
else if(msg.id == 0xB2){
value9 = msg.data[0];
value10 = msg.data[1];
value11 = msg.data[2];
value12 = msg.data[3];
value13 = msg.data[4];
value14 = msg.data[5];
value15 = msg.data[6];
value16 = msg.data[7];
pc.printf("CAN 2nd Message Received: COB_ID= 0x%03X data_length = %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else if(msg.id == 0x183){ // TPDO 1 from NS node3
value17 = msg.data[0];
value18 = msg.data[1];
value19 = msg.data[2];
value20 = msg.data[3];
value21 = msg.data[4];
value22 = msg.data[5];
value23 = msg.data[6];
value24 = msg.data[7];
tpdo1_node3_message++;
pc.printf("CANOpen TPDO1 Node3 NS207 Message Received: COB_ID= 0x%03X data_length = %d
data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value17,value18,value19,value20,value21,va
lue22,value23,value24);
}
else if(msg.id == 0x283){ // TPDO 2 from NB node4
//pc.printf("CANOpen TPDO2 Node3 NS207 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
```

```
else if(msg.id == 0x383){ // TPDO 3 from NB node4
//pc.printf("CANOpen TPDO3 Node3 NS207 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else if(msg.id == 0x483){ // TPDO 4 from NB node4
//pc.printf("CANOpen TPDO4 Node3 NS207 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else if(msg.id == 0x184){ // TPDO 1 from NB node4
value25 = msg.data[0];
value26 = msg.data[1];
value27 = msg.data[2];
value28 = msg.data[3];
value29 = msg.data[4];
value30 = msg.data[5];
value31 = msg.data[6];
value32 = msg.data[7];
tpdo1_node4_message++;
pc.printf("CANOpen TPDO1 Node4 NS228 Message Received: COB_ID= 0x%03X data_length = %d
data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value25,value26,value27,value28,value29,va
lue30,value31,value32);
}
else if(msg.id == 0x284){ // TPDO 2 from NB node4
//pc.printf("CANOpen TPDO2 Node4 NS228 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else if(msg.id == 0x384){ // TPDO 3 from NB node4
//pc.printf("CANOpen TPDO3 Node4 NS228 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else if(msg.id == 0x484){ // TPDO 4 from NB node4
//pc.printf("CANOpen TPDO4 Node4 NS228 2nd Message Received: COB_ID= 0x%03X data_length
= %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
}
else{
/*pc.printf("*** CAN Message detected is not for this Node ID\n\r");
value9 = msg.data[0];
value10 = msg.data[1];
value11 = msg.data[2];
value12 = msg.data[3];
value13 = msg.data[4];
value14 = msg.data[5];
value15 = msg.data[6];
value16 = msg.data[7];
pc.printf("Message Received: COB_ID= 0x%03X data_length = %d data =
%d,%d,%d,%d,%d,%d,%.1f,%d\n\r",msg.id,msg.len,value9,value10,value11,value12,value13,val
ue14,value15,value16);
*/
}
};
return 0;
};
int e_can::can_init_send(int &id,int &data_length,int &data1){
CAN_msg_t.start();
char data[] = {data1};
// set the empty object //
msg.id = id;
msg.len = data_length;
msg.format = CANExtended;
msg.type = CANData;
msg.data[0] = data[0];
// Write message //
CAN_msg_t.reset();
```

```
can1.write(msg);
err = 0;
//pc.printf("\n\rCAN Message Request: Waiting for Node Reply\n\r");
while ((msg_flag == 0) && (err <= 10)){
pc.printf("Scanning for CAN Message Err: %d\n\r",err);
if(can1.read(msg)&& msg_flag == 0){
if (msg.id == 0x01 && msg.data[0] == data1){
value1 = msg.data[0];
float time = (CAN_msg_t.read_us());
pc.printf("\n\rCAN Message Recieved: ID: %d Data: %d
%.2fms\n\r",msg.id,value1,(time/1000.0));
msg_flag = 1;
};
};
err++;
};
msg_flag = 0;
return 0;
};
```

Main ECU PCB for Hybrid Vehicle