Helsinki Metropolia University of Applied Sciences
Degree Program in Information Technology

Lawrence Nwaogo

Implementation of an Embedded System Networking and Vehicle Information Display

Bachelor Thesis, 10th July 2009
Instructor: Sami Ruotsalainen, Project Manager
Supervisor: Antti Piironen, Principal lecturer

Author Title Number of Pages Date	Lawrence Olisakwu Nwaogo Implementation of an Embedded System Networking and Vehicle Information Display 57 11 February 2011
Degree	BEng
Degree Program	Information Technology
Specialisation option	Digital Signal Processing
Instructors	Sami Ruotsalainen, Project Manager Antti Piironen, Principal Lecturer

The purpose of the project was to implement an embedded system networking and vehicle information display. This was realized by the use of the Controller Area Network (CAN) as the protocol and the front end software Centrafuse for the display.

There are many hardware implementation of CAN, the implementation used for the design was the Microchip PIC32MX795F512L microcontroller which has the CAN controller module integrated on it. The transceiver was the Microchip MCP25515 and the dongle was the CAN232 from Lawicel.

My task in the project which was the display of vehicle information has been completed and tested using a simulation program. The result was the display of information identical to the stimuli. The other part, the implementation of embedded system networking is still in progress. When the entire project is completed, the design can be modified to be used in other application areas such as elevators and heaters.

Keywords	Centrafuse, controller area network, front end , embedded system

Contents

1 Introduction

The Implementation of Embedded System Networking and Vehicle Information Display is an application developed specifically for use in automobiles. This application enables the vehicle user to see data broadcast on the controller area network (CAN) bus by different CAN nodes. It is then displayed on a computer screen using the front end software. This idea of vehicle information display on the computer screen is different from the normal information display on the dashboard as seen in vehicles. This is because the use of computer provides the vehicle user with the option of controlling the display.

The design has two parts: CAN application and Information Display application. The CAN application is an interface to the hardware, CAN nodes. While the Information Display application is a plug-in that plugs into a particular run-time environment, Centrafuse. It has graphical user interface. Centrafuse is a product of Flux-Media in the USA. With special aggreement, this software has been licenced to the project team for which I am a member. Centrafuse is software designed to execute plug-ins. It is very flexible in the sense that designers can design custom plug-ins. These plugins could be an interface to media player, Global Positioning System (GPS), or other devices that can be used in vehicles. In this design, the two areas of focus are on reading the communication port and displaying the information via Centrafuse. The data from the communication port are information from different CAN nodes. They are the vehicle speed, the battery state of charge, odometer and temperature. [1]

The CAN bus data have format, hexadecimal. It does not convey information to the vehicle user therefor it is useless. To make it useful, it has to be further processed in software and then displayed as information to the vehicle user. The display of vehicle information on the computer screen is my primary assignment at the Automotive Engineering Laboratory of Helsinki Metropolia University of Applied Sciences. As a student of Information Technology with major in Digital Signal Processing (DSP) and Embedded Systems I have decided to implement embedded system networking and vehicle information display in my final year project. This is because my assignment in

the project, display of vehicle information on the screen, has information source from the distributed systems.

2. Application Areas in Automobiles

Many applications use the CAN protocol for a distributed network. The power train of vehicles, communication between engine management, anti-lock braking system (ABS), transmission control, active suspension and electronic throttle signaling are examples.

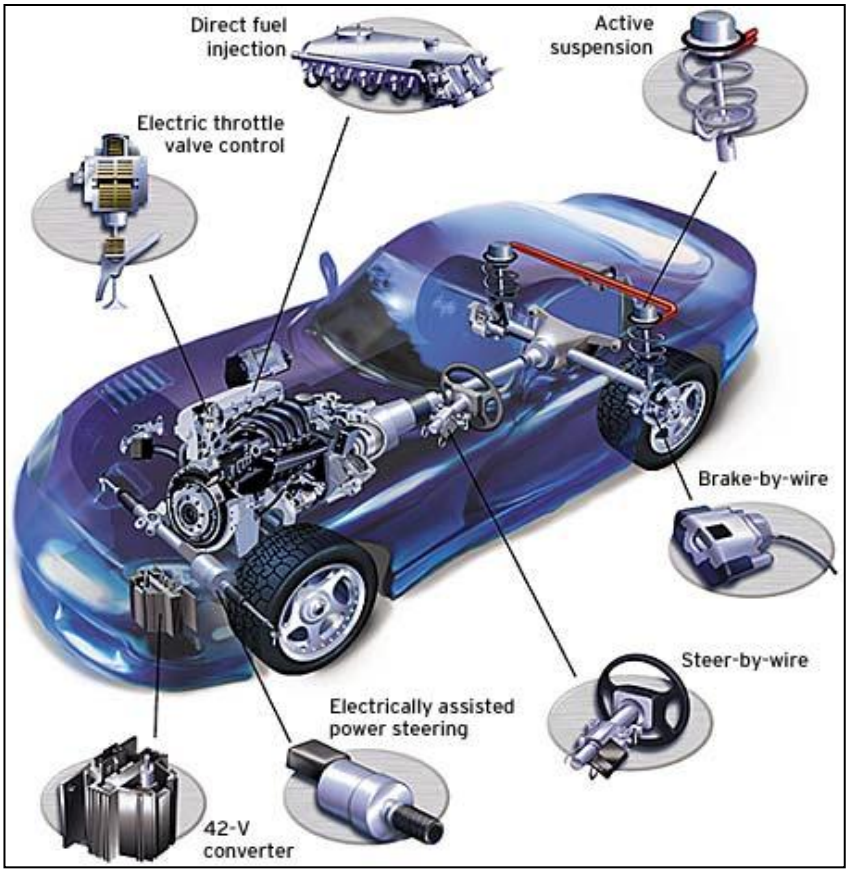

Figure 1: Car components connected to a can bus [1]

The bus exchanges information such as motor revolution per minute (RPM), vehicle speed and throttle setting. In the case of braking, messages from the ABS unit can go to the engine controller to switch off the injection and reduce the stopping distance. A car

radio, based on speed information broadcast on the CAN bus, can automatically adjust volume, radios, air bags, multifunctional display for satellite navigation, mobile phones, or intelligent sensors can exchange information on a second bus. The dash controller can then bridge high-speed and low speed CAN buses. The CAN controller can be a stand-alone or integrated into a microcontroller. Based on the CAN protocol, different nodes can accept data that is broadcast on the bus and check to see if the messages are relevant to them or not. [1]

3 Controller Area Network (CAN) Theory

There are many manufactures of microcontroller products and they differ from each other technologically. For microcontrollers from different manufacturers to participate in a network, there has to be a common protocol.

The Controller Area Network (CAN) is a protocol meant for this. The idea of CAN was introduced by R. Bosch GmbH in Germany at the beginning of the 1980s to address the growing complexity of vehicle functions and network. [2, 20] Bosch cooperated with Intel Corporation for further refinement of the protocol specification and its integration into silicon. [2, 20]

However, CAN was originally meant for small network such as in an automobile and in a hostile environment. Today, the CAN application is in use in other fields due to its many advantages.

The advantages of CAN are many. Amongst them are increased communication links due to the decentralized system architecture, decreasing device redundancies in complex systems, reduced cables and connector cost, decreasing system integration effort, improving device and system diagnostics, improving interoperability and exchangeability of devices. [2, 89]

The CAN is defined based on the Open Systems Interconnect (OSI) model which is widely used to describe the functionality of communication systems on the basis of hierarchically layered approach.

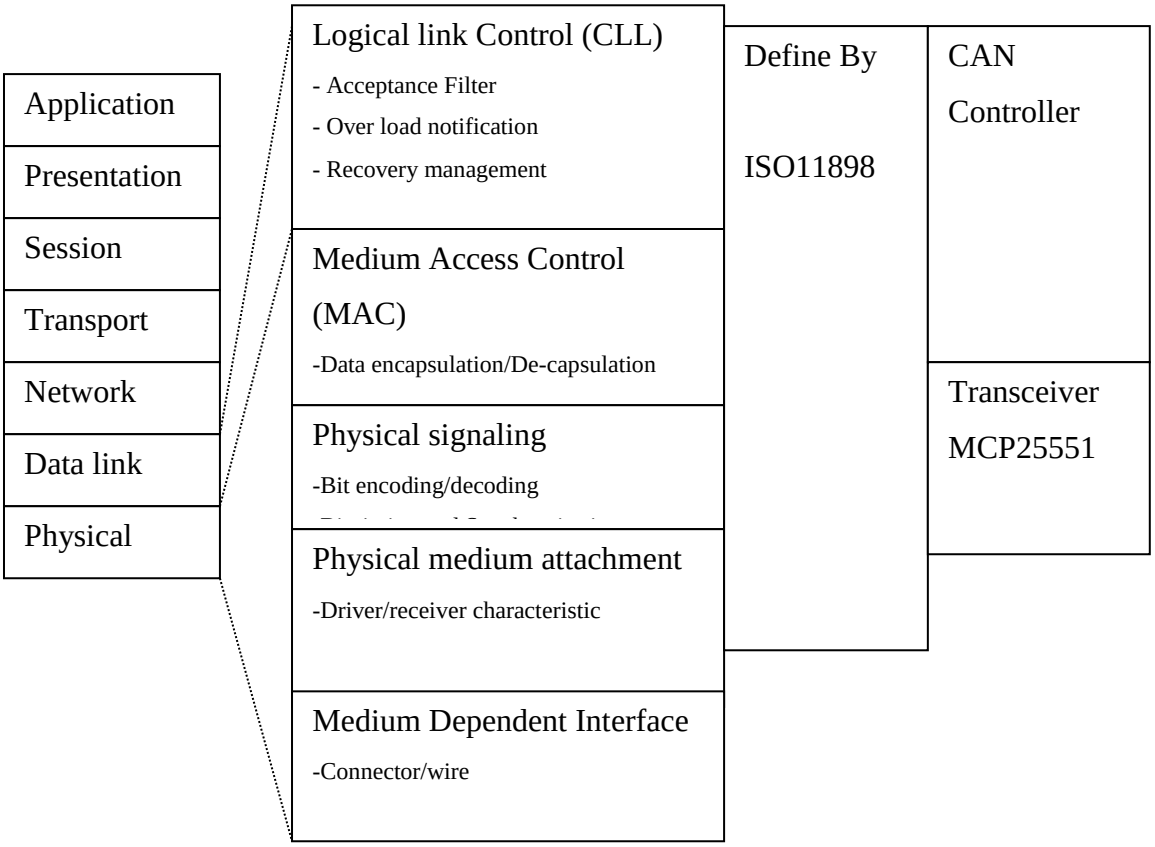

Figure 2: CAN and the OSI model [2, 4]

The CAN hardware implementations cover parts of physical layer, the first and the data link layer as can be seen in Figure 2, while various software solutions cover the seventh layer, the application layer. [2, 4]

3.1 The Physical Layer

The physical layer is the first in the OSI model. It is the entry port to and from the outside world. Its functionalities are divided into three parts:

- Physical signaling (bit representation, bit timing and bit synchronization)
- Medium access unit (transceiver)
- Medium interface (connectors)
- Bus medium [2, 77]

The above mentioned points will be discussed in full detail.

3.1.1 Bit Representation

Physical signaling involves bit coding. There are different kinds of bit coding. Among them are Non-Return-to-Zero (NRZ) and Manchester or Pulse Width coding. They differ in the following ways. With the Manchester coding, two time slots are required for representing a bit, whereas, with NRZ coding, signal level remains constant over the bit time and one time slot is required for representing the bit time.

The maximum time interval between resynchronization points is determined by the tolerance of the nodes oscillators. This is made possible by applying the method of bit-stuffing. By this method, a complementary bit after a specific number of constant bit levels is inserted into the bit streams. NRZ signal coding with bit stuffing of stuff width of five applied in CAN protocol guarantees the highest transport capacity with sufficient synchronisation ability.

The NRZ coding with bit stuffing of the CAN protocol encompasses the frame segment's Start of Frame (SoF), arbitration, control and data field as well as the Cyclic Redundancy Check(CRC) sequence of a CAN data frame as well as remote request frame. As soon as a transmitter detects a sequence of five bits of equal level, it inserts a bit of complementary value into the transmitted bit stream. The remaining frame segments of a data frame or remote request frame cyclic redundancy check (CRC), delimiter, acknowledgement (ACK) field as well as end-of frame (EoF)) have a fixed form recessive level and are transmitted without bit stuffing. The same applies to error and overload frame. [2]

3.1.2 Bit Timing

The CAN protocol is unique in a way because it uses synchronous data transmission instead of asynchronous transmission. The realization of synchronous transmission requires a method of bit synchronization. In a synchronous transmission protocol, there is only one start bit available at the beginning of the frame. This is not sufficient to keep the bit sampling of the receiver sufficiently synchronous with the transmitter. To enable the receiver to correctly sample the transmitter bit stream at the end of a message frame, continuous resynchronization of the receiver is required.

The resynchronization means that the detection of the clock period in the received signal is possible with every valid signal edge within the bit stream. The maximum time difference between transmitting and receiving oscillators during the maximum time period between signal edges must be compensated. This means sufficient spare time (phase buffer segment) before and after the nominal sample point within a bit interval.

The nominal bit time is divided into four segments:

1. Synchronization segment (sync_seg)
2. Propagation delay segment (prog_seg)
3. Phase buffer segment 1 (phase_seg_1)
4. Phase buffer segment 2 (phase_seg_2). [2, 82]

Among the CAN protocol features are the uses of non-destructive bitwise bus arbitration and the dominant acknowledge bit. Signal propagation from a transmitter to a receiver and back to the transmitter must be completed within one bit time. [2, 82]

Therefore, in addition to the time reserved for resynchronization, a further time segment (propagation delay segment) is required for compensation of the signal propagation over the bus line as well as for internal signal delay in the transmitting and receiving nodes.

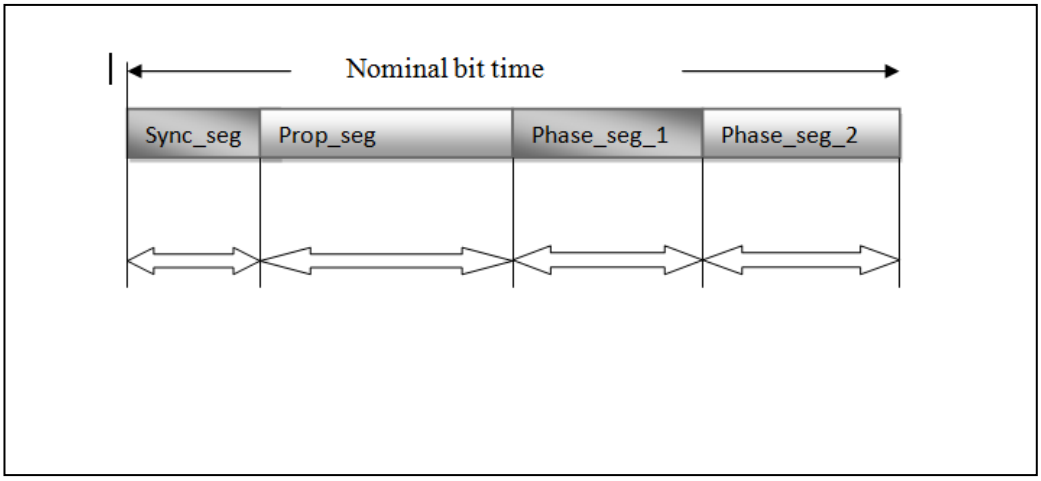

Figure 3: The bit time segments for synchronization [2, 85]

The length of the time segments of a bit interval is specified as a multiple of the basic time unit, time quantum (TQ). This is derived from the oscillator period. The TQ is the time resolution of the synchronization mechanism and is taken into account in the bit time by the synchronization segment. It follows from this that a signal edge can happen just shortly after a basic time unit has passed and therefore be missed. The synchronization segment is that part of the bit time where edges of the CAN signal level are expected to occur. The distance between an edge that occurs outside of the synchronization segment and the synchronization segment is called the phase error, and it is denoted by the letter, e. [2, 80]

The propagation delay segment provides the times necessary for settling the maximum signal propagation delay within the network. This segment must be twice as long as the sum of the minimum signal propagation delay between two nodes along the bus line and the internal delay times of the transmitting and the receiving nodes. [2, 81]

By means of the phase buffer segments before and after the nominal sampling points, spare time is reserved for shifting of the actual sampling point during resynchronization. The Synchronization occurs only on edges from recessive to dominant bit levels. Edges are detected by sampling the actual bus level in each time quantum and comparing it with the bus level at the previous sampling point.

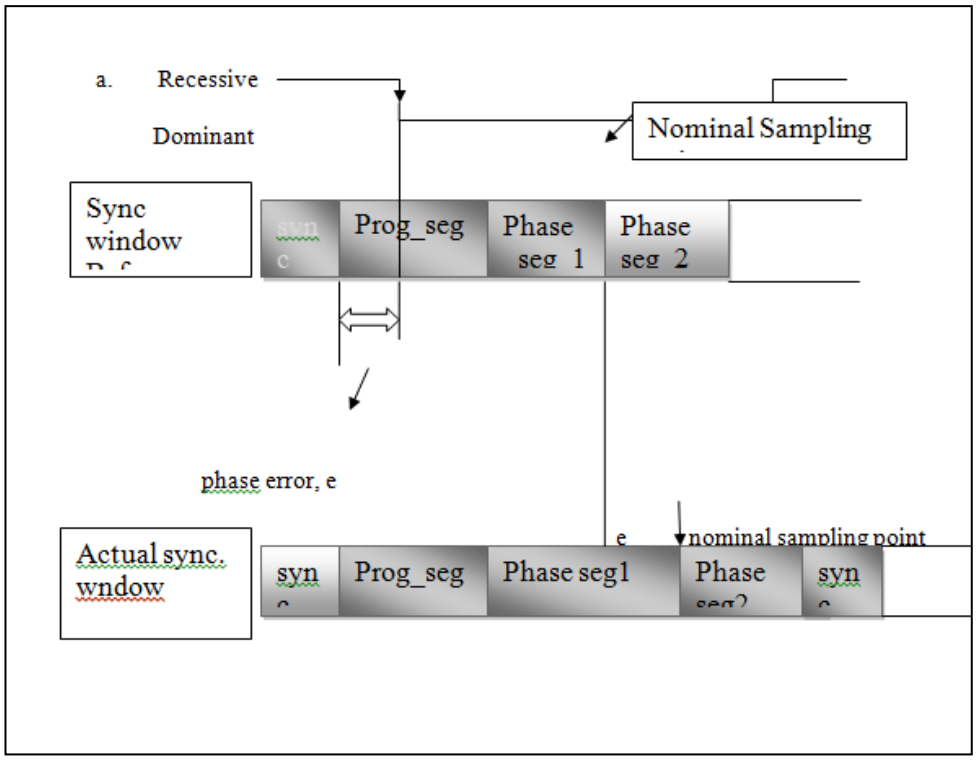

Figure 4: Transmitter faster than receiver [2, 83]

An edge is synchronous if it is detected inside the synchronous segment, otherwise the distance between the edge and the end of the synchronous segment is the phase edge error measured in time quanta. If the edge occurs before the synchronous segment,
the phase error is negative otherwise it is positive. For a positive phase error, phase segment 1 is lengthened as shown in Figure 4.

The amount of shortening or lengthening of the phase buffer segments allowed per resynchronization is limited and can be programmed between one and maximum of four (phase segment 1). If the magnitude of the phase error is less than the resynchronization jump width (SJW), phase segment 1 is lengthened by the magnitude of the phase error, else it is lengthened by the SJW. For a negative phase error, phase segment 2 is shortened accordingly. [2, 79]

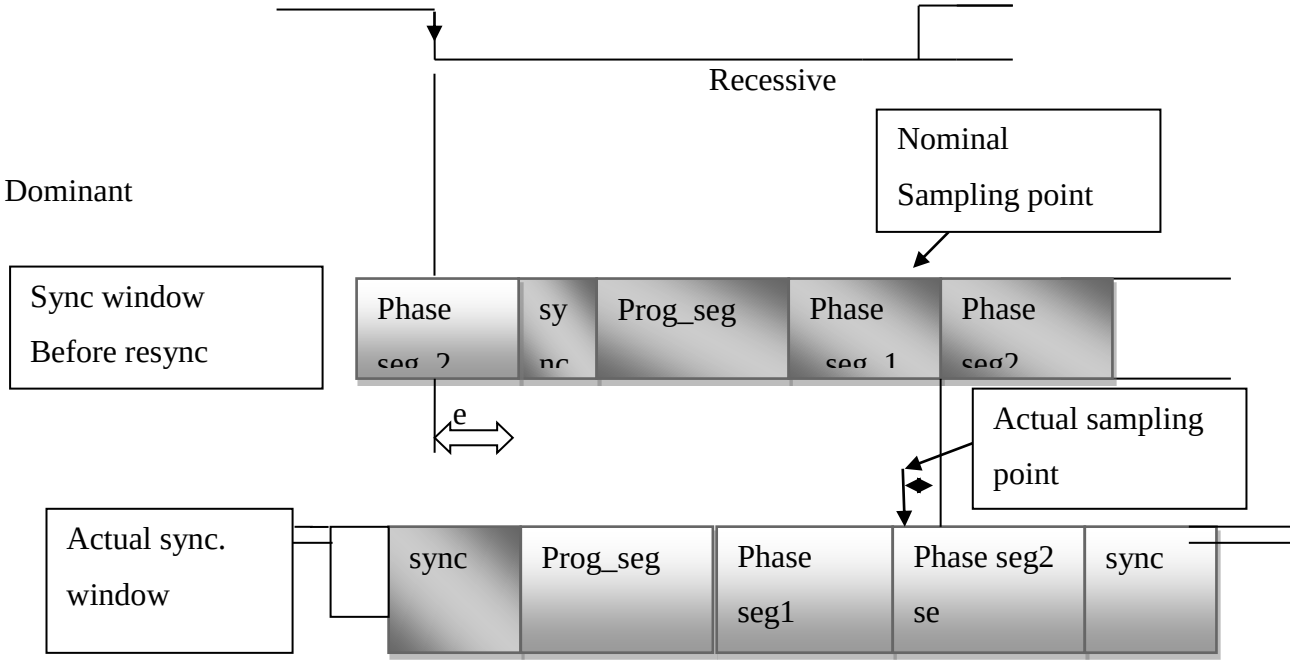

Figure 5: Transmitter faster than receiver [2, 79]

When the magnitude of the phase error of the edge is less than or equal to the programmed value of the SJW, the results of the hard-synchronization and resynchronization are the same. If the magnitude of the phase error is larger than the SJW, the resynchronization cannot compensate for phase error completely, thus an error of (phase error - SJW) remains. Only one synchronization is allowed between two sample points. The resynchronizations maintain a minimum distance between edges and sample points, giving the bus level time to stabilize and filter out spikes that are shorter

than the sum of the propagation segment and the phase segment 1. Figure 6 illustrates resynchronization for transmitter faster than receiver. And Figure 6 illustrates bit timing and errors. If the transmitter is slower than the receiver, signal edge will be late. The dimensioning of the propagation delay segment defines the maximally possible bus length at a specific data rate or the maximally possible data rate at a given bus length. Figure 6 describes a situation between two bus nodes, N and M farthest apart from each other in a network.

During bus arbitration, node N is transmitting a recessive bit level at point of time t_1. This signal level reaches the node M at point of time t_2 delayed by signal propagation t_d. The maximum time delay is defined by the signal propagation time between the nodes which are farthest apart.

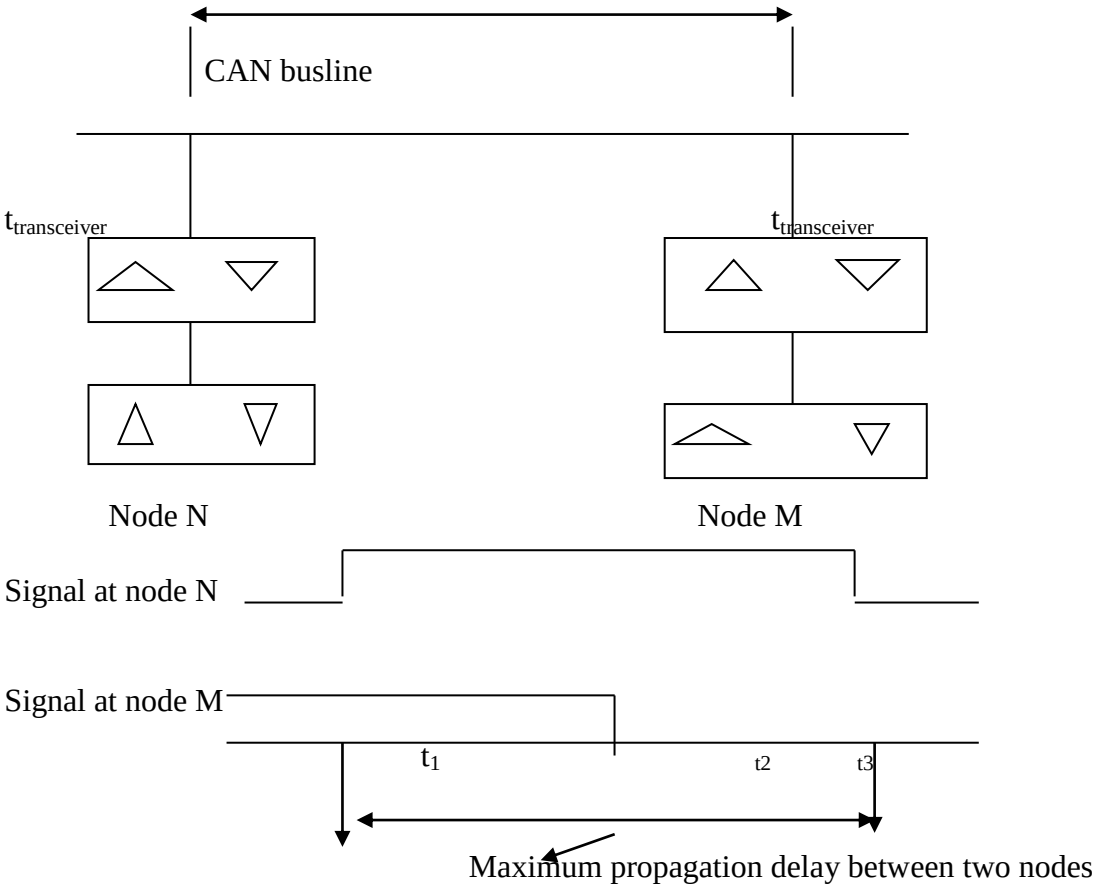

Figure 6: Length of the propagation delay segment [2, 90]

Based on the existing propagation time, the synchronisation of node M with node N is also delayed by this time so that node M itself can start transmitting a new bit level, e.g.

dominant level still at point of time t_2. The level cannot be detected by node N until point of time t_3. This means that until this time, node N is not able to decide whether the signal level which it transmitted was retained or whether it was replaced by a dominant level from node M. [2, 78]

The length of the propagation delay segment can be dimensioned based on the following:

1. $T_{prop\_seg} \geq 2*t_p$
2. T_{prop_seg} - the time of the propagation delay segment.
3. t_{el} - propagation delay of the electrical medium.
4. T^*_p - specific propagation of the bus medium [2, 91].

The maximum bus length L_{max} of a paired wire bus is obtained with the specific signal propagation time t^*_p on a pair of wire of propagation 5 ns/m according to the following:

$$L_{max} = \frac{0.5*\text{tprop}_{\text{seg}} - tel}{5 \text{ ns/m}} \quad (1)$$

Table 1 Bit rate and maximum bus length based on relation 1 [2 92].

Bit rate (Kbit/s)	Maximum Bus length(m)
500	110
250	620
125	790
100	1640

Based on the equation 1 above, the maximally possible bus length is obtained through the largest possible propagation delay segment 2 m for every 10 ns of additional signal delay. Table 1 is calculated from the relation between admissible bit rate and maximum bus length.

From the equation above, it can be seen that the electrical signal delay due to the signal delays of the transceiver unit, CAN controller and the optoelectronic couplers reduce the maximally possible bus length by

A rough estimation of the relationship between the maximum bus length and maximum data rate is obtained by assuming that a certain share x of the bit time, t_{bit} is available as t_{prop_seg} to account for the signal propagation. Then it can be written as

$$T_{prop\_seg} = x*t_{Bit} = x/\text{Bit rate} \quad (2)$$

for several bit rates under the assumption :

$$x= 0.85,\ t_{el} = 300 \text{ ns.} \quad (3)$$

3.1.3 Transmission Medium

The basic requirement of the CAN bus arbitration approach is the ability to represent a recessive and dominant signal level. This principle is realized with electrical media. With electrical transmission media, the physical media that is commonly in use in the implementation of CAN network is a differential driven pair of wires with common return. A single bus line is also in use especially in the area of vehicle body electronics as shown in Figure 7.

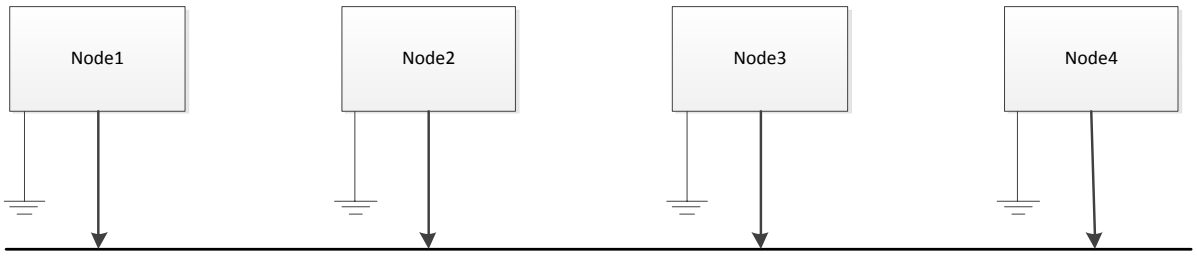

Figure 7: CAN bus with an electrical medium (single wire) [2, 93]

This means that the transmission medium is not in a recessive condition unless all nodes have applied a recessive bit level. Its operation is similar to that of an AND gate.

As soon as a node has started transmitting a dominant bit level, the bus is in a dominant level. In addition, electromagnetically induced radio interference can be compensated by a twisted pair of wire thereby increasing interference immunity. Physical layer CAN protocol was based on two-wire bus. In addition to interference immunity, it is easy to implement.

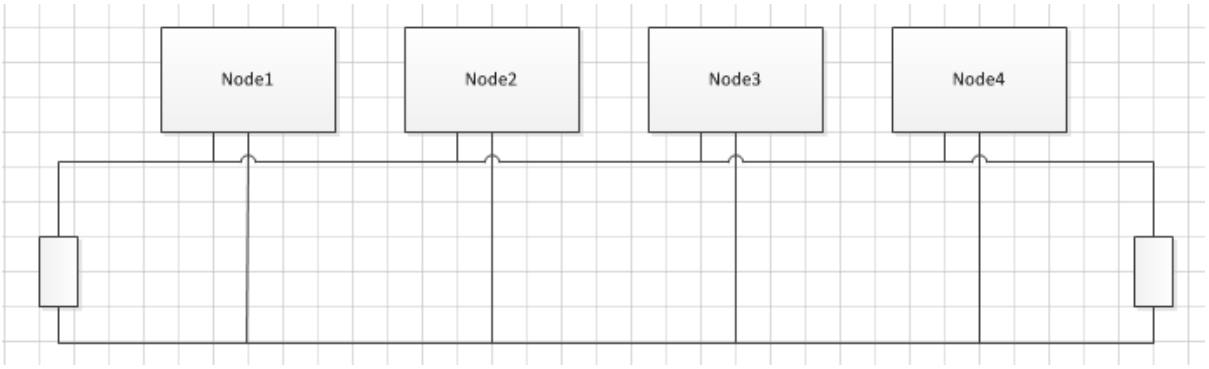

Figure 8: CAN bus with electrical medium (two wire bus, high and low) [2 93]

The two-wire bus enables differential signal transmission and is thus resistant to common mode errors. The bus line must be terminated at each end by resistors of 120 Ohms to avoid signal reflection.

Single wire bus

The use of single wire bus is common in motor vehicle body electronics. This solution assumes a common ground available to the nodes. Since the single wire bus line is exposed to electrically induced radio interference. When it is not shielded, a larger signal shift level is necessary to improve the signal-to-noise ratio. This in turn affects the degree of electromagnetic emission and requires the reduction of the signal slope and thus the maximum possible data rate. The single wire bus line is specified by the standard Society of Automotive Engineering (SAE). [2 94]

3.1.4 Bus Medium Accesses

The connection between a CAN controller chip and the physical bus medium is designated bus access medium. For the connection of a CAN controller chip to a two-wire differential bus, a variety of integrated CAN transceiver chip are available. Basically, the interface to the electrical bus medium consists of a transmitting amplifier and a receiving amplifier. This is enclosed in an integrated circuit (IC) called a transceiver. One of the functions of the bus access medium is the adaptation of the signal representation between the chip and the bus medium. In addition to that, the bus

interface unit has to meet a series of additional requirements. In particular, as a transmitter, the transceiver provides the following functions:

1. Provision of sufficient driver output capacity
2. Protection of the on-controller-chip driver against overloading
3. Reduction of electromagnet radiation. [2, 108]

At the receiver's side, a CAN transceiver performs the following functions:

1. Provision of a defined recessive signal level
2. Protection of the on-controller-chip input comparator against over-voltage on the bus lines
3. Extension of the common mode-range of the input comparator in the CAN-chip
4. Provision of a sufficient input sensitivity
5. Detection of bus error, such as line breaking, short circuit, shorts to ground or voltage shorts as well as switching to asymmetrical single wire operation. [2, 109]

3.2 The data link layer

The transmission of data between nodes of a network is regulated via the definitions of the Data link layer. The first main function of this layer is the construction of data frames, which not only contain the data to be transmitted but also control information and an error detecting code. The transmitted control information for example enables the receiver to recognize whether a frame is intended for it and to detect whether a received frame has been corrupted by transmission error. Corrupted data frames are normally rejected by the receiver and retransmitted automatically by the sender. [2, 43]

The second main function concerns controlling medium access by a node denoted by Medium Access and Control (MAC). This is shown in Figure 10. Types of frames are the following:

1. Data frame

2. Remote frame
3. Error frame
4. Overload frame [2, 47- 51]

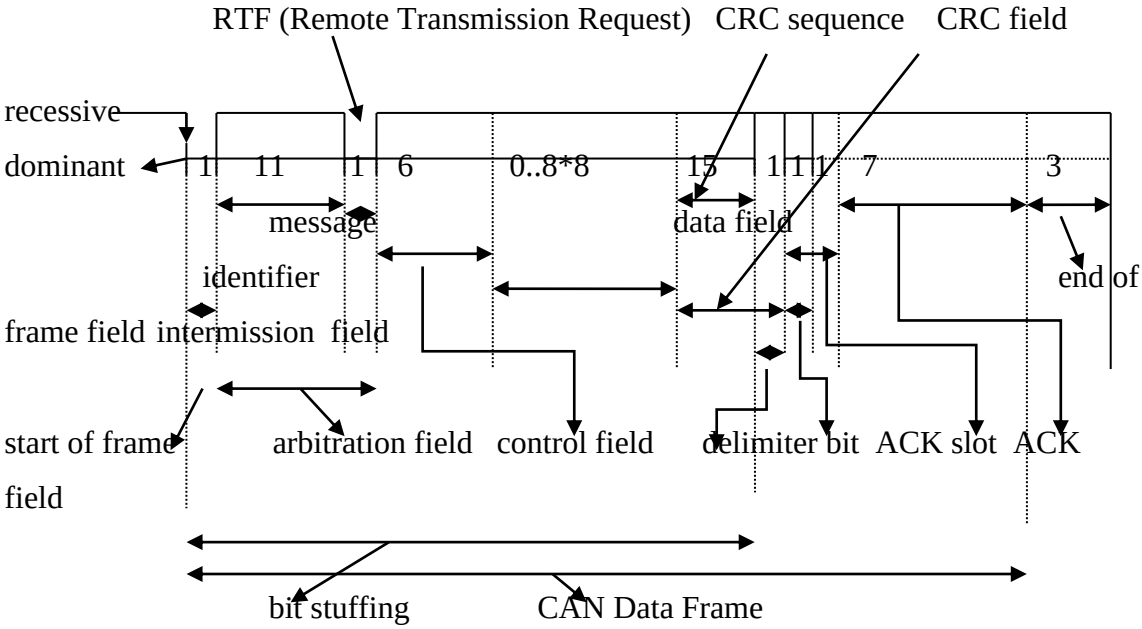

Figure 9: Message frame format [2, 48]

It necessary to cope with the problem of two or more nodes accessing the medium at the same time, either by preventing the problem from happening or by recognizing an access conflict and resolving the problem so that no data are lost.

The different segments of the standard message frame are the following:

1. Start of Frame (SoF) field - a “0” indicates the beginning of a message frame
2. Arbitration Field - it contains a message identifier and the remote transmission request (RTR) bit that discriminates between a transmitted data frame and a request for data from a remote node. “1” means request for data (remote frame) and “0” means data frame.
3. Control Field – consist of six bits, two 0 bits (r0 and r1) reserved for future use. A four bit data length code (DLC), the number of data bytes.
4. Data field – zero to eight bytes.
5. Cyclic Redundancy Check (CRC 16 bits) – fifteen-bit cyclic redundancy check code and a one delimiter bit.
6. Acknowledgement Field - two bits

7. Slot Bit – transmitted as one, but subsequently overwritten by a zero from any node that successively receives the transmitted message
8. Delimiter Bit – a bit always one.
9. The end of Frame (EoF) field seven bits always zero.
10. The Intermission Frame – three bits always one. After the three bit intermission period, the bus is recognized to be free and may remain idle for any length of time [2, 41-59].

3.2.1 Remote Frame

A remote frame is used by a node to request the transmission of a data frame by the receiving node. The requested frame is specified by the identifier transmitted with the remote frame. Whereas a specific frame can be requested by all of the receiving nodes, only one transmitter node must be responsible for the requested frame. The frame format of a remote frame corresponds to that of a data frame except that the RTR bit is transmitted recessively. A remote frame that arbitrates simultaneously with a data frame with the same identifier loses arbitration. Furthermore the data field of a remote frame is empty.

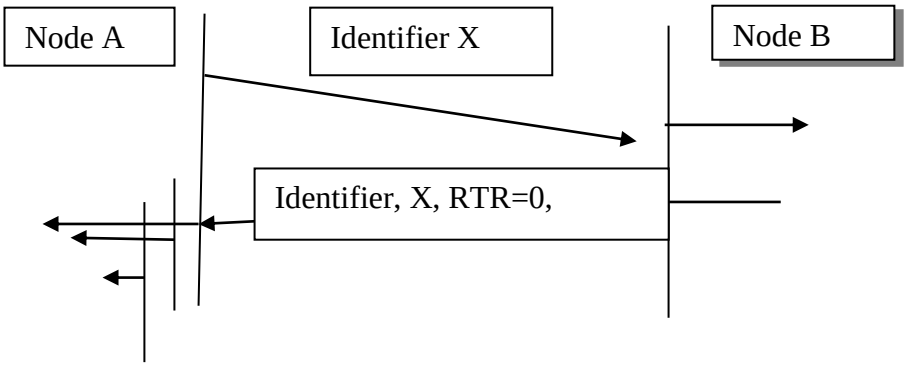

Figure 10: Principle of data request cycle [2, 53]

The data length code in the remote frame must correspond to that of the requested data frame [2, 52]. Figure 10 shows the principle of a data requested cycle. As can be seen, Node A requests transmission of frame with the identifier, X (sequence of bits) by sending a remote frame.

Node B is responsible for the transmission of this frame and transmits it to node A and to all the other nodes in the network (Broadcasting) interested in the frame. Depending on the capability of the protocol chip used in node B, the data frame is automatically transmitted by the protocol chip (transceiver) [2, 53].

3.2.2 Error Frame

The detection of any error during transmission or reception of a data frame is signalled by an error frame which purposely violates the rule of bit stuffing and causes the transmitter of the frame to repeat the transmission. Also the detection of an error during transmission or reception of an error or overload frame causes the transmission of a new error frame. An error frame consists of bit fields. The first is given by the superposition of error flags transmitted by one or several nodes. The second field is a sequence of eight recessive bits, indicating the end of the frame (Error Delimiter) analogously to the data and remote frame. When a receiver detects a ’dominant’ bit as the first bit after sending an error flag the receiver error count will be increased by 8 [2, 53].

3.2.3 Overload Frame

The function of the overload frame is to request a delay of the next data or remote frame by the receiver node (Requested overload frame) or to signal certain error condition related to the intermission field. Reactive overload frames are transmitted after detection of the following error conditions:

1. Detection of a dominant bit during the first two bit of the intermission field. A dominant bit detected in the third bit of the intermission field is interpreted as SOF.
2. Detection of a dominant bit in the last bit of EOF by a receiver or detection of a dominant bit by a receiver and transmitter at the last bit of an error or overload frame delimiter [2, 57].

An overload frame can be considered as a special form of an error frame and is similar to an error frame composed of an overload flag and an overload delimiter. In contrast to

the transmission of an error frame, the generation of an overload frame is limited to very specific condition related to the intermission field [2, 57].

Another important distinction is that the transmission of the overload frame does not cause the retransmission of a previous frame as an error frame generated due to errors detected in data or remote frame does.
The overall flag consist of a sequence of six consecutive dominant bits and destroys the fixed form of the intermission field. As a consequence, all other nodes also detect an overload condition and on their part transmit an overload flag.

The overload delimiter consists of eight recessive bits. After transmitting an overload flag, every node monitors the bus until it detects a recessive bit. Then every node has finished transmitting its overload flag and all nodes transmit further seven recessive bits to complete the eight bits overload delimiter. The start of an overload frame to delay further frames must begin with the first bit of an expected intermission field. Not more than two overload frames are allowed to be generated successively to delay the next frame. Reactive overload frames begin with the following next bit interval [2, 57].

The demands for security of data transmission result from the originally intended application of the CAN protocol in vehicles. To meet these demands, several mechanism are provided by the CAN protocol to detect errors. These are: bit check, frame check, cyclic redundancy check, ACK check and stuff rule check [2, 57].

3.3 CAN Application Layer (CAL)

CAL is an implementation of the ISO/OSI application layer, it is the seventh layer in the hierarchy. Different implementations exit, this is partly because of different integrated circuit (IC) manufacturing technologies. However, they all adhere to the same ISO specification [2, 192].
The application layer is realized in software. The choice of implementation depends on the manufacturing technology. Microchip has one of the most advance CAN technologies, which is version 2.0b active of the extended CAN (ECAN).

Microchip is suitable for beginners and practicing engineers. It offers a range of options from 8 to 32 bit microcontrollers or digital signal processing units, which has CAN or ECAN modules. Microchip also produces transceivers. In order words, a complete CAN node solution can be realized on Microchip products. Another advantage of using this product is the free integrated development environment, MPLab (IDE) and the student version of the C 30 compiler.

4 Development Tools

The development tools were a very important component of this project. There are three parts to this set of development tools: the device programmer, the integrated development environment (IDE), and the compiler suite. Before I can begin to design any hardware or software, I have to make sure that these tools are available; programs cannot be compiled without a compiler and they cannot be downloaded to the PIC hardware without the device programmer.
The use of Microchip's products is encouraging because Microchip offers development tools that are powerful, easy to use, and relative inexpensive. The availability of inexpensive, high-quality development tools, is key to choosing this product.

4.1 Device Programmer

A device programmer is used to get the code from the MPLAB onto the device. It contains a USB connection to the programming interface, which connects to the PIC board. It also functions as real-time debugger. Debugging is a very important feature since it gives much needed feedback about variable contents during execution. The device programmer of choice is the In-Circuit Debugger 3 (ICD3). The reason is that, though expensive, it supports RS232 unlike the Picket2, and it programs very fast [4].

4.2 Integrated Development Environment (IDE)

Microchip offers a free development environment (IDE) called MPLAB for developing software for Microchip product. MPLAB is the de facto IDE for PIC programming. It has all of the major feature common to all integrated development environments such as

a text editor, a notion of projects and workspaces, compiler and linker integration, build script and integrated hardware debugger support. The debugging capabilities include the standard mechanism for setting break points and stepping through code as well as watch window for monitoring variables and registers. The disassembly window is quite good because it list the lines of c code and the associated lines of assemble code, allowing the user to see how the c code is implemented. It is worth mentioning that this IDE supports other tools like DSPic Filter Designer. This tool is useful when designing filters because all you need to do is enter your filter parameters and the software will do the designing [5].

4.3 Compilers and Assemblers

A powerful compiler is very important when developing code for any micro-controller in C language. This design is in C language so a compiler and an assembler are required. Microchip has many compilers that integrate quite easily with the MPLAB IDE. For higher optimization and performance, Microchip has commercial version of the C compiler which is a collection of C 18 for 8 bits, C 30 for 16 bits and C 32 for 32 bits respectively the price is $8,950.

There is a free student version of this compiler that has all the features of a commercial compiler and it is for 60 days. After this period, code optimization is disabled. It is important to add that there is no limitation to code size. All the essential features like DSP library, ANSI-C compliant, standard C libraries are supported [6].

5 Hardware Components

The implementation of this design is based on the Institute of Electrical and Electronics Engineering (IEEE) CAN standard of the International Standard Organization (ISO) 11898-1, 11898-2, 11898-3. The 11898-1 defines the CAN Data link Layer and Physical Signaling. The 11898-2 defines CAN high Speed Medium Access Unit. The 11898-3 defines CAN low speed, Fault Tolerance Medium-Dependent-Interface. [7]

5.1 The PIC32MX795F512L

The implementation of this design was made possible with hardware components. The hardware consists of the microcontroller the PIC32MX795F512L. The transceiver, the MCP25515 was connected to the PIC32MX795F512L and the CAN adaptor, the CAN232 was connected to the MCP25515. Each of these components will be explained.

5.2 The CAN Controller

The PIC32MX795F512L is a 32 bit micro-controller. It has many modules among them are the CAN and the Analogue to Digital Converter (ADC). These are the required modules for this design. The CAN module of PIC32MX795F512L is the Microchip MCP2515 it implements CAN 2.0b active called ECAN. ECAN is different from standard CAN because it has 29 bit identifier in the arbitration field frame. It is active because it can process both 11 and 18 bit identifier. The ADC is the physical interface to the analogue world. It has ten bits. [8]

5.3 The CAN Transceiver

The CAN Transceiver MCP25515 is a high speed CAN fault-tolerant device that serves as interface between the CAN protocol controller and the physical bus. This device provides differential transmit and receive capability for the CAN protocol controller and is fully compatible with the ISO- 11898 standard, including 24V requirement. It has a maximum operation speed of 1Mb/s. Each node in a network system must have a device to convert the digital signal generated by a CAN controller to signal suitable for transmission over bus cabling (differential output). It also provides a buffer between the CAN controller and the high voltage spike that can be generated on the CAN bus by outside sources (EMI). [9] The MCP25515 CAN outputs will drive a minimum load of 45 Ohms and a maximum load of120 Ohms, making it possible to have a maximum of 112 nodes to be connected in a network. The Receive Data (TXD) output pin reflect the differential bus voltage between CAN High (CANH) and CAN Low (CANL). The high and low state of the RXD output pin corresponds to the dominant and recessive of the CAN bus, respectively.

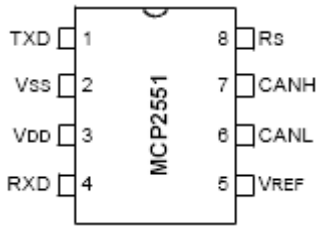

Figure 11: MCP25515 signal names and pin position. [9]

CANH and CANL are protected against battery short circuit and electrical transients that occur on the CAN bus. Figure 11 is the block diagram of the Microchip MCP25515 CAN transceiver [9].

Table 5: Signal name and pin function. [9]

Pin Number	Pin Name	Pin Function
1	TXD	Transmit Data Input
2	VSS	Ground
3	VDD	Supply Voltage
4	RXD	Receive Data Output
5	VREF	Reference Output Voltage
6	CANL	CAN low-level Voltage I/O
7	CANH	CAN High-level Voltage I/O
8	RS	Slop control Input

It describes the pin-out of the device. Table 5 shows the pin numbers on the first column, the signal name on the second column and functions on the third column. During installation, the pin number or signal name is meant to help the designer in accurate configuration.

5.4 CAN Adaptor

In order to read the data on the CAN bus by the plugin, there has to be a way to translate from one protocol to the other. The different protocols in this case are CAN and RS232.

The translation is done by a device called CAN232. It can be connected to any PC running DOS, Windows, Mac or linux and talk with the unit in ASCII format. It could also be used together with embedded single board computer that needs a simple CAN connectivity without changing the existing hardware.
The CAN232 handles both the 11 bit ID (identifier) as well as the extended 29 bit ID. Figure 14 is the CAN232 device. [9]

Figure 12: Top view of CAN232. [9]

The installation of this device is simple. The RS232 side of the dongle is inserted into the PC's COM serial port or via a cable to the host system (such as an embedded system). The CAN side of the dongle has the same pin out as the standard CAN in Automation.

This device must be powered via the CAN side with 6 to 16V direct current (DC). It needs between 40- 100mA depending on the number of nodes on in the network. Figure 13 is the block diagram showing how to connect it.

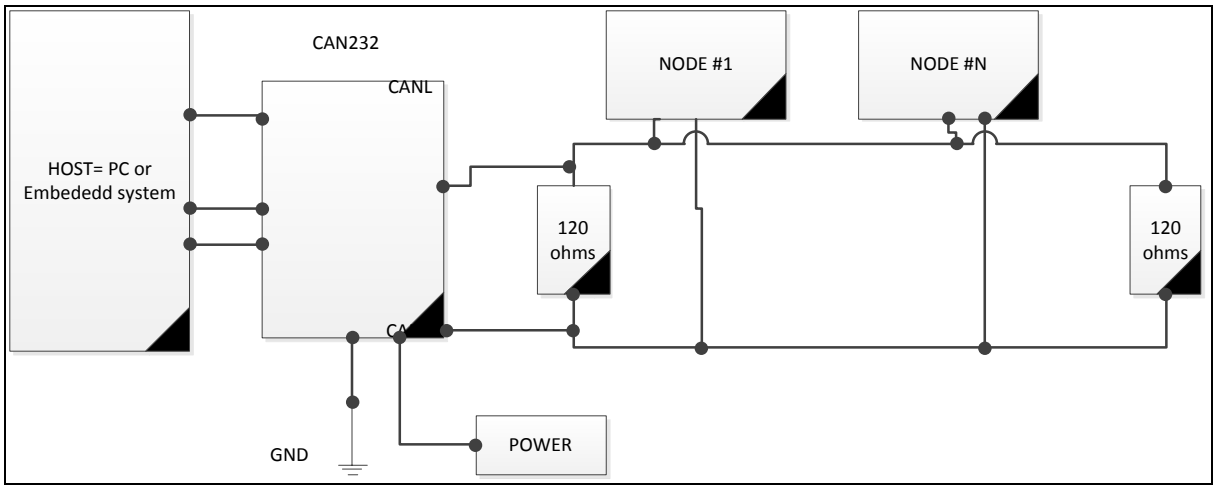

Figure 13: CAN network connected to a PC via CAN232. [9]

5.5 Software Design

The development approach for this design is modularity. This simply means the grouping of functions of similar functionality under one file. The benefit of this approach is usability. Modules are treated as black boxes after being created. It means that the module can be used without knowing how it was implemented. Only the designer knows but those who are curious to know the contents are free to do so. The highest level of abstraction in this design is the Header File. The header file is a collect of files. It begins with include files that are system specific. This is required in every software development because during the process of linking, the address of registers and system functions will be resolved.

5.5.1 User Header File

Based on the principle of modularity, I have defined and included the following files in a single file. This is because it can be reused in a similar application, making time to the market shorter.

5.5.2 File CANFunction.h

The file CANFunction.h is defined to do the following:

1. System_ Frequency is a constant initialized to 80 MHz this value is used for computing baud rate pre-scaler (maximum clock)
2. CAN_Bus_Speed is a constant initialized to 250 Kb/s
3. CAN1MessageArea is a two dimensional array.

The following function prototypes are also included in this file: CAN1Init function is responsible for the initialization of the CAN1 module. It sets up speed, FIFOs, filters and interrupts. FIFO is set up for receive (RX) with 8 message buffers. Filter 0 is set with mask 0 for standard message frame identifier (SID). This means that whenever a message is received, its identifier is ANDed with the mask bits. Should there be a change in the bit sequence of the identifier, it will be detected and the frame will be rejected. This function takes no parameter and no return type.

5.5.3 File GenericType.h

The second file that is included is the file GenericType.h. It contains all the data types, structures, unions, enumeration and type definition. This file is very important for many reasons. First, the user can freely choose what name to use for variable types. Secondly, the structure can be modified or populated by the designer. Data alignment can also be specified.

5.5.4 File Timer.h

The third include file is the Timer.h. The contents of this file are constants, otherwise called macros for timer period and delay. The bellow list contains the function prototypes necessary for the configuration of the timer.

1. IsOneSecondUp – this function will return Boolean value true if one second has expire since the last time the function returned true otherwise false. The parameter is void.

2. TimerInit – this function will initialize and start Timer 1 for the period specified by the timer period macro. The return value is void and the parameter is void.

5.6 User Implementation

The files in this section are the implementation of the prototypes defined in the header file. They are responsible for the configuration of the register, including message acceptance register, mask register.

5.6.1 File CANFunction.c

The file CANFunction is the implementation of the function prototypes in the header file. It starts with the inclusion of header files. This is necessary because the prototypes, structures and macros are in different files and without their inclusion, the linker will emit linker error at run-time. To get the ball rolling, few variables were defines. These variables are of type Boolean and their default value is false. They are intended to be updated in an interrupt subroutine should the execution of the program proceed from there. This means that an interrupt has occurred.

The overall idea of this program is to simulate the real-life situation in a vehicle whereby one node is monitoring a process like the engine temperature or vehicle speed and broadcasting the information on the CAN bus and be able to receive information from the CAN bus too. The inclusion of two CAN modules CAN1 and CAN2 in the PIC32MX795F512L has made it possible for a single node to transmit and receive data through different channels. They are 32 first in first out (FIFO) buffers and each FIFO buffer has 32 registers. The first thing in the configuration of a CAN controller according to the Microchip implementation of CAN protocol is to configure the CAN registers. To do this, the CAN controller must be put in configuration mode. My design approach is to have a function in which the setting up of the registers will be done. I call this function CAN1 initialization (CAN1Init). CAN1 will be configured to transmit message to CAN2 and receive from CAN2.

5.6.2 File CAN1Init.c

The file CANInit.h was designed using following steps:

1. Declaration of variable of type CAN_BIT_CONFIG. CAN_BIT_CONFIG is a structure with many members as will be seen shortly.
2. Switching the CAN module ON and switching it to configuration mode. Switching the CAN module on is done by calling the system API CANEnableModule and passing as argument the module name, CAN1 and a Boolean value, TRUE. This Boolean value represents "1". The next sub-step is to enter configuration mode. This step is very important because no CAN configuration register can be access without entering configuration mode. The API to do that is the CANSetOperationMode. This API takes as argument, the module, CAN1 and a constant, CAN_CONFIGURATION. It is worth mentioning that the API and constant names are written as they are in the system libraries.

At this point, it is not known if the system has entered configuration mode or not. It is important to wait for the system to enter configuration mode. The system function CANGetOperationMode in a WHILE loop is called passing as argument system API CANGetOperationMode. The system will block pending the completion of entering configuration mode. As soon as the system enters configuration mode, the following registers can be configured using the variable defined earlier, starting with the clock register. CAN is a serial communication protocol, transmitting one bit at a time synchronously. The time slot for the transmission of one bit has been divided into:

1. Phase segment 1 = assigned 3 time quanta (3TQ)
2. Phase segment 2 = assigned 3 time quanta (3TQ)
3. Propagation delay segment = assigned 3 time quanta (3TQ)
4. Synchronous jump width = assigned 2 time quanta (2TQ)
5. Sample three times = assigned TRUE
6. Phase segment 2 time select = assigned TRUE

The completion of timing register configuration is followed by calling system function CANSetSpeed. CANSetSpeed takes three argument, CAN1, address of CAN bit configuration register and CAN bus speed.

The CAN module must know how much memory space is reserved for its operation. The reservation of memory space is done by system function CANAssignMemoryBuffer. This function takes three arguments, CAN1, pointer to CAN message FIFO area. Configuration of transmission registers is necessary in CAN. This is because the registers are bi-directional and half duplex. For this design, the system function CANConfigurationChannelForTx is called. This function takes five arguments CAN1 (module name), channel 0 (as transmission channel), size of message (pay load) is eight, remote transmission request is disable and the message priority is medium.

The CANConfigureChannelForRx is the system function responsible for the configuration of receive register. It takes four arguments, CAN1, channel 1, message size (pay load) is eight and receive the full is enabled. Configuration of filter and mask. Every received message is first stored in assemble buffer. This message's identifier is then compared with the acceptance filter identifier and the filter mask. The following functions are responsible for setting up acceptance filter, filter mask, link to channel and enable the filter:

1. CANConfigureFilter - takes four argument; CAN1 (module name), CAN filter 0 (channel 0), message identifier (0x8004001), message type (extended message frame).
2. CANConfigureFilterMask – takes five arguments; CAN1 (module name), mask 0 (filter mask), mask bits, message type (extended identifier), filter mask identifier type.
3. CANlinkFilterToChannel – this takes four arguments; CAN1 (module name), acceptance filter, acceptance mask and channel.

The completion of the above mention step is followed by the assignment of vector priority, vector sub-priority and interrupt using the functions below:

1. INTSetVectorPriority – this function takes two arguments; CAN1 vector interrupt and interrupt priority level (4).
2. INTSetVectorSubPriority – this function takes two arguments; CAN1 vector interrupt and interrupt sub-priority (0).

3. INTEnable – it takes two arguments; CAN1 interrupt and interrupt enabled

At this point all the necessary registers are configured therefore the system will be restored to normal mode from the configuration mode. The following functions will perform this task:

1. CANSetOperationMode – this takes two arguments; CAN1 and operation mode (normal mode).
2. Check to see if the system has been restored by entering a WHILE loop and calling system function CANGetOperatingMode. The argument of this function is CAN1, the return type is compared with a constant. If the result is true, that means that the system is now in normal mode.

5.6.3 File Timer.c

The file Timer contains the code for the timer initialization, timer interrupt handler and delay code. It starts with the declaration of two variables, one is of type Boolean and the other of unsigned integer 32. The functionalities of the functions in this file are described below:

1. TimerInit – this function takes no arguments and no return type. It is a good practice to disable the timer before the configuration of its registers. This is important as it will prevent interrupt during the course of the configuration. Having disabled the timer, the registers can now be configured. The timer period register is assigned the value defined in the timer header file. The interrupt flag is cleared and the timer 1 global interrupt is enabled. The timer 1 interrupt priority is medium level (level 4). The two global variables are now given default values. With all the registers configured, the timer is now enabled.
2. IsOneSecond – this function takes no arguments and has no return type. Its basic function is to synchronize the transmission of data. By calling this function, data can be transmitted every second. First it checks to see if the global variable has been updated in the interrupt handler. If TRUE, it means that the data can be transmitted.

5.6.4 File CAN2Init.c

The CAN_BIT_CONFIG is a structure with many members. Switching the CAN module ON and switching it to configuration mode is done by calling the system API CANEnableModule and passing as argument the module name, CAN1 and a Boolean value, TRUE. This Boolean value represents “1”. The next sub-step is to enter configuration mode. This step is very important because no CAN configuration register can be access without entering configuration mode. The API to do that is the CANSetOperationMode. This API takes as argument CAN1 and a constant, CAN_CONFIGURATION. It is worth mentioning that the API and constant names are written as they are in the system libraries. At this point, it is not known if the system has entered configuration mode or not. It is important to wait for the system to enter configuration mode. There is a call to system function CANGetOperationMode in a WHILE loop passing as argument system API CANGetOperationMode. The system will block pending the completion of entering configuration mode.

The CAN is a serial communication protocol, transmitting one bit at a time synchronously. The time slot for the transmission of one bit has been divided into:

1. Phase segment 2 = assigned 3 time quanta (3TQ)
2. Propagation delay segment = assigned 3 time quanta (3TQ)
3. Synchronous jump width = assigned 2 time quanta (2TQ)
4. Sample three times = assigned TRUE
5. Phase segment 2 time select = assigned TRUE

The completion of timing register configuration is followed by calling system function CANSetSpeed. CANSetSpeed takes three argument, CAN1, address of CAN bit configuration register and CAN bus speed.

The CAN module must know how much memory space is reserved for its operation. The reservation of memory space is done by system function CANAssignMemoryBuffer. This function takes three arguments, CAN1, pointer to CAN message FIFO area. The Configuration of transmission registers is necessary in CAN. This is because the registers are bi-directional and half duplex. For this design, the system function CANConfigurationChannelForTx is called. This function takes five

arguments, CAN1 (module name), channel 0 (as transmission channel), size of message (pay load) is eight, remote transmission request is disable and the message priority is medium.

The CANConfigureChannelForRx is the system function responsible for the configuration of receive register. It takes four arguments, CAN1, channel 1, message size (pay load) is eight and Receive Full is enabled. The Configuration of filter and mask is necessary. Every received message is first stored in Message Assemble Buffer (MAB). This message's identifier is then compared with the acceptance filter identifier and the filter mask.

The following function are responsible for setting up acceptance filter, filter mask, link to channel and enable the filter.

1. CANConfigureFilter - takes four argument; CAN1 (module name), CAN filter 0 (channel 0), message identifier (0x8004001), message type (extended message frame).
2. CANConfigureFilterMask – takes five arguments; CAN1 (module name), mask 0 (filter mask), mask bits, message type (extended identifier), filter mask identifier type.
3. CANlinkFilterToChannel – this takes four arguments; CAN1 (module name), acceptance filter, acceptance mask and channel.
4. The following functions will enable vector priority, vector sub-priority and interrupt.
5. INTSetVectorPriority – this function takes two arguments; CAN1 vector interrupt and interrupt priority level (4).
6. INTSetVectorSubPriority – this function takes two arguments; CAN1 vector interrupt and interrupt sub-priority (0).
7. INTEnable – it takes two arguments; CAN1 interrupt and interrupt enabled
8. At this point all the necessary registers are configured therefore the system will be restored to normal mode from the configuration mode. The following functions will perform this task:

9. CANSetOperationMode – this takes two arguments; CAN1 and operation mode (normal mode).
10. Check to see if the system has been restored by entering a WHILE loop and calling system function CANGetOperatingMode. The argument of this function is CAN1, the return type is compared with a constant. If the result is true, that means that the system is now in normal mode.

This brings to completion acceptance filter, mask and the timing register configuration.

5.6.5 File Main.c

The Main is the entry point of this program. It starts with the inclusion of device specific header files and user defined header files. What this function does is to call defined function to send data over the bus or receive data transmitted by other nodes in the network. CAN is a synchronous serial transmission protocol. This simply means that clock edge is needed for transmission and the transmission speed or baud rate should be known in advance. To determine the baud rate, the oscillator must be configured to output a given clock. I have chosen the maximum clock rate of 80 MHz This is achieved by using the macro. The following fields are initialized:

1. FPLLMUL – the oscillator is scaled by a factor of 20
2. FPLLIDIV – the oscillator is divided by 2
3. FPLL0DIV – the oscillator is post divided by 1
4. FWDTN – the watch dog timer is disabled
5. POSCMOD – the primary oscillator mode is high speed
6. FNOSC – phase lock loop is enable for primary oscillator
7. FPBDIV – the pre-scaler is divided by 1

After the configuration of the oscillator registers using the macros, the CAN buffers are created followed by calls to initialization functions.

6 Displaying Vehicle Information

It is my task for the Automobile Engineering laboratory of the Metropolia University of Applied Sciences. The application’s objective is to read the CAN bus and display on the

screen vehicle information. In order to display the data transmitted on the CAN bus, the following hardware and software requirements must be met:

1. 1 GHz or faster processor
2. 512 MB or more of RAM
3. Windows operating system
4. The front end software

The operation of Centrafuse is realized by the design of plugin. The plugin is designed by the use of software development kit (SDK) provided by Centrafuse. The default programming language is C #.

6.1 Software Installation

Upon downloading the software or from the CDROM, the following steps are needed.

1. Run the installation wizard and choose the language of choice (English) then click OK. The installation wizard will automatically install missing software dependencies such as .Net Framework 2.0 or higher version, DirectX and Bluetooth.
2. Click Install, then Next ,agree to the end user license.
3. Click Next, enter your name and organization information.
4. Click Next , select the installation folder: Identify the location where you will save the program and its supporting files. The default install location is C:\Program Files\Flux Media\Centrafuse. Click Change to modify the install directory then confirm installation directory.
5. Click Next, choose the display options to your specifications. Click Next, and then Install to start the installation. The installation progress bar will indicate progress of the installation.

When the installation is complete, click Finish to complete the installation and open Centrafuse. Figure 14 will be displayed. Figure 15 shows the front panel of Centrafuse.

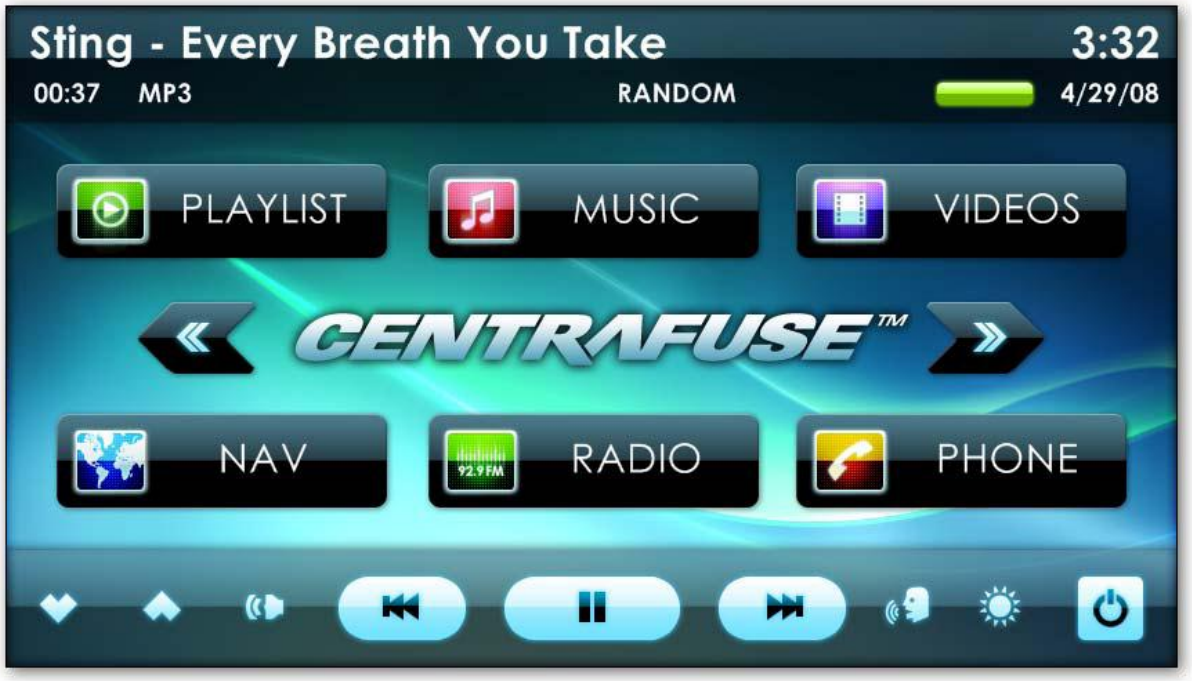

Figure 14: The front view of the front end software. [10]

It consists of buttons and arrows for navigation. The forward arrow is used to navigate to un-used buttons which can be put to use by designing custom plugins.

6.2 Software tool

C # is one of the languages used in the design. This is because the system in which this design will be plugged in, supports it. It is a programming language that has its roots in different programming languages such as java, C, C++ and visual basic. Therefor my knowledge of C language and C++ makes this design easier. The second reason for using C # is its support for graphical user interface (GUI) such as buttons which is required in this design. Access to the communication port has been made easier with the inclusion of serial port object in the resent version of .Net. It is a product of Microsoft. [11, 8]

The Extensible Mark up Language (XML) is the other language used for this design. This is because XML makes it possible for the information about button, font colour, dimensions of the label needed in this design to be stored in a separate file and loaded into the plugin at run-time. This approach is in line with the design model, which is modularity. This information can be read and displayed using C #. [12]

6.3 File Hello.cs

The file **Hello.cs** is the main module because it references all the other modules. It start by including all the libraries it needs at run-time using the Using keyword followed by dot, the base class and derived class or classes. For this design, keywords start with uppercase letter, method and variable names start with uppercase letter with no space in between. The Centrafuse proprietary library is also included. This is because this library is not part of the C # library. Any reference to its functions without its inclusion will emit run-time error.

It is known that there are references to files in different directories so the name of each directory and its path must be known to the program. To achieve this, string constant variables are declared and initialized with the name of the directory path and the file. Haven done this, log file is written to the user application data directory.

The class constructor is used to initialize object variables. The following variables are initialized:

1. CentrafusePluginPauseAudioFlag - This Boolean variable is initialized to false. The reason is that when it is initialized to true, all system audio will be paused when the plugin is shown. The main title, position and status labels will be cleared and can be populated. It is worth mentioning that the string in parentheses are declared in system library and must not be changed.
2. CentrafusePluginName (_CFPluginName) – this must be initialized with the name of the plugin
3. CentrafusePluginIsGUI (_CFPluginIsGUI) – A Boolean value true must be assigned to this variable because only GUI plugins are available button action
4. CentrafusePluginHasSettings (_CFPluginHasSettings) – true value is assigned to this Boolean variable. It enables the system to be configured in advanced setting.
5. CentrafusePluginHasBasicSettings - It enables the system to be configured in basic setting.
6. Serial Port Instance Variable – This initializes these properties of the serial port, port name, stop bit, baud rate, parity bit, data bits.

6.4 Creation of Configuration Files

The configuration file is a separate file with an XML extension. This file has to be made known to the program. The way to do this is to initialize a string variable with the path of the configuration file and then create and write to it. The Environment class member method of C # its member were used. This is followed by the creation of application folder. This folder is the root of the application file system. Its contents are centrafuse, plugins and plugin name.

6.5 Centrafuse Methods

LoadSettings - Centrafuse has its set of libraries, the methods in the libraries are used to setup the system. Before a plugin designed for Centrafuse can be used, it has to be loaded into the system. This is the method responsible for this. The word in parentheses is system specify. CentrafuseLocalSkinSetup (_CFLocalSkinSetup) - this method is called to setup the skin and when the system resolution has changed. It encapsulates methods that are closely related. It calls method ClearControl to clear all control arrays. This is necessary for the system to work properly and the initialization of the skin with specified skin path follows.

The skin has images that has to be loaded using the method CentrafuseLoadImages. Should the plugin be scaled, ScalePlugin method is called. The location and dimension of the rectangle where the vehicle information will be written to is known as label. This is designed using the Centrafuse method CreateLabel. With the label created, the method responsible for writing information to the label is then called. The plugin is closed by calling the method PluginClose which calls method Dispose. In order to show the plugin, method CentrafusePluginShow is called by the system to initialize a Boolean variable with TRUE value. CentrafusePluginUpdatelanguageSkin (_CFPluginUpdateLanguageSkin) - The default language of the system is English but it supports these languages too: Czech, Danish, Dutch, Finnish, French, German, Greek, Hebrew, Italian, Norwegian, Portuguese, Russian, Spanish, Sweden and Turkish. Should there be language change, first the program checks if the language file exits, if

so then this method is called by the system to initialize an instance variable with the parameter value passed to it.

The configuration file must be updated too. To do this, an instance of XML document is created first, this is known as xml node. With this, member method Load is called. The list of arguments to method Load are method GetFolderPath from type Environment, which takes as argument localApplicationData a member of SpecialFolder structure derived from Environment type. Haven done this, a single node (XML node) is created using the method SelectSingleNode which is a member of XmlDocument type. The argument to the select single node method is a string constant. To get the property, method HtmlEncode is called passing as argument the name of the folder where the language file is located. Finally the folder path is saved on the XmlDocument type object and then load settingmethod is called.

The system has to update its skin each time the plugin is loaded. The process of updating the skin starts by the program checking to see if the skin exits. If so, an instance of XmlDocument type is created, xml node. With this, member method Load is called. The list of arguments to method Load are method GetFolderPath from type Environment, which takes as argument localApplicationData a member of SpecialFolder structure derived from Environment type. Haven done this, a single node (XML node) is created using the method SelectSingleNode which is a member of xml document type. The argument to the select single node method is a string constant. To get the property, method HtmlEncode is called passing as argument the name of the folder where the skin file is located. Finally the folder path is saved on the

XmlDocument type object and then loadSetting method is called. The plugin has to be setup each time it is loaded. This is because some of the default settings have to be changed. To do this, the method CentrafusePluginShowSetup is called to do the following:

1. Initialize variable of type DialogResult with the value Cancel of type DialogResult.
2. Create object of type Setup

3. Initialize Setup object member MainForm with the data field Mainform of this class instance.
4. Call Setup object member method SetupSection.
5. Initialize DialogResult object with the return value of Setup member method ShowDialog.

Check to see if the return value of ShowDialog method is the Boolean value True, if so then the method LoadSetting is called. To close the Setup object, method Close is called and then a value is returned to the calling method.

6.6 File Setup.cs

The file Setup.cs is derived from the system class Centrafuse Setup (CFSetup). It starts by including libraries using the keyword Using followed by derived class(s). It creates two objects of class XmlDocument (XmlDocument) and defined the following string constants:

1. PluginPath – this variable is initialized with the path to the plugin.
2. PluginPathLanguages – this variable is initialized with the path to the language folder.
3. ConfigurationFile – the name of the configuration file is assigned to this variable.
4. ConfigurationSection – the name of the main tag in the xml file is assigned to this variable
5. LanguageSection – the path to the setup section is assigned to this variable
6. LanguageControlSection – the path to the name of the language control section is assigned to this variable.

The class Setup constructor will be called each time this plugin's setup is opened from the Centrafuse Settings Page. This setup is opened as a dialog from the Centrafuse PluginShowSetup method call into the main application form. What this class constructor does, is to initialize an instance member variable with the total number of pages. Should the user of the plugin have advanced setting turn on, then the Centrafuse ShowAdvancedSettingFlag will be set to TRUE. Should the plugin require more pages

for advanced setting, then the value assigned to the total number of pages variable will be updated with the new value in the software. The default value for the current page is one. This value must be assigned to the instance member variable CurrentPage of type CentrafuseSetup (CFSetup). Building the controls is done by calling method BuildControls in this constructor. These methods below are called:

1. CentrafuseloadConfiguration – this loads the plugin configuration file from the specified path.
2. CentrafuseLoadLanguage – this loads the plugin language file from the specified path. It is loaded using the language specified in the configuration file.

Having the path to the configuration and language file, the next step is to load the configuration and language files into the local xml document. This is followed by calling the Centrafuse method SetupSection which in turn calls method BuildSetup, loadSetupPage for this object. To populate the buttons, the following Centrsfuse methods are called:

1. CentrafuseUpdateButtonText – it takes as argument a string constant, OK and method GetText whose argument is a path to a string SAVE.
2. CentrafuseUpdateButtonText – it takes as argument a string constant, CANCEL and method GetText whose argument is a path to CANCEL.
3. CentrafuseUpdateButtonText – it takes as argument a string constant, TITLE and method PluginLang. ReadPluginField whose argument is a path to a string TITLE.

6.7 Helper Function

The function Helper is responsible for reading the CAN bus. Its operation is event driven, the event is the availability of data on the communication port. When this happens, delegate is called. Its design starts by creating an object of type SerialPort. The initialization of this type member is done in the class constructor according to the following:

1. ObjectName field is initialized to COM1
2. ObjectParity is initializes to NONE

3. ObjectStopBit is initialized to ONE
4. ObjectDataBits is initialized to eight
5. ObjectBaudRate is initialized to 9600

The initialization of the member variables is followed by calling the method Open. This method opens the communication port. Method Readline of type Console is called to read the communication port at every DataReceive event. The read data is assigned to a string variable for further processing. Because this data is CAN data, basic CAN theory says that a message frame has identifier segment and data field segment respectively. Both segments of the message frame must be separated in software. The identifier is used to check the authenticity of the message frame by matching it against a given string constant. This means that the identifier should be extracted first. The processing of the extracted data is not possible under the present data type, therefor it has to be converted to type double. With the completion of data conversion, mathematical operations are now possible. In order to write the vehicle information on the screen using method Writeline of type Console, it has to be reconverted to string.

6.8 File Skin.xml

The Skin is an XML file, its function is to store the dimensions of the button controls, labels, co-ordinates, font colour and font size. This file is loaded into the system at run-time. The entire file is segmented as follows:

1. Screen window – Has property, dimension (width = 848, height = 480).
2. Title window – Has property, co-ordinates (x = 0, y = 84, width = 848, height = 310).
3. Label - Has property, FALSE, FONTCOLOUR = white, FONTNAME = Century Gothic, FONTSIZE = 18, FONTSTYLE = Bold, Co-ordinate(x = 8,y= 8), Width = 832, height = 24, align = left.

6.9 File English.xml

The English XML file contains the text to be displayed on the plugin button. This is called by the main application each time the plugin is clicked. The configuration file stores the path to the Skin folder, Language file and Log events. This is called by the system during setup.

7 Testing and Outcome

The design is in process but part of it has been completed. The uncompleted part was tested using a simulation program written in C #. The program was installed on a different PC. The PCs were connected by a cable and the transmission protocol is the RS232. The purpose of connecting both PCs together was to have one PC transmit the data and the other to read the data and display it on the computer screen. The transmitting PC transmits hexadecimal code at the rate of 9600 Baud, and the receiving PC reads this data at the same rate.

In order to make sure that the simulation program resembles the intended program, I included a string constant which acts as the identifier. The front end software reads the communication port, compares the identifier to see if it matches a given string constant. This step is required in CAN to identify the right message frame and discard the un-wanted frame. It guarantees that corrupt messages are not processed. The received data was processed before it was displayed. The result of the testing was that the transmitted data was received without errors.

The CAN network was also tested. The platform used for the test was the Explorer 16 Development board. Two nodes were connected in a network and buttons were used to trigger transmission. It was expected that the node that receives data will light LEDS. The result was the lighting of LEDs as expected. Although it was a two-member network, the design principle will apply to a large network. The result was communication between the members of the network. The next step is to translate CAN to RS232 using the dongle (hardware).

8 Conclusion

Although the project is on-going, additional features will be added to the front end software as time goes on. The goal of the project is to read data from the distributed systems in a network and display the result in real-time on the screen. With due consideration, CAN was the choice for communication in network as it provides immunity against interference and noise due to reduced number of cables in the network. It also supports error detection and arbitration.

The result of the project was the display of the simulated information on the screen. The information was analyzed and found to be consistent with the transmitted data. Another set of stimuli was also used to test the program. In this case no information was displayed. The reason for this was the wrong message identifier used. Should the wrong identifier be displayed, the filtering processes has failed. Many challenges lie ahead. One of them is the gate-way between the two protocols, CAN and the RS232. The program to interface the gate-way is being design. When completed, the project can be used in other areas where client server relation applies.

As a student of embedded system, working on this project has illuminated Embedded System to me and I now have a better understand of this subject. The knowledge I acquired will be used in other areas where the principle of client server applies.

References

1 Controller Area Network Diagnostics (CAN) [online]
URL: http://www.aa1car.com/library/can_systems.htm.
Accessed 23 December 2009.

2 Etchberger E. Controller Area Network, Basics, Protocol, Chip and Application Weingarten, Germany: IXXAT Press; 2001.

3 Lawrenz W. CAN System Engineering from Theory to Practical Applications. Wolfenbuettel, Germany: Springer; 1997.

4 MPlab ICD3 Inter-Circuit Debugger [online].
URL: http://www.microchip.com/icd3.
Accessed 1 January 2010.

5 MPlab Integrated Development Environment[online].
Microchip Technology Inc. 2010.
URL: http://www.microchip.com/stellent/idcplg. Accessed 23 December 2009

6 MPLAB C Compiler for PIC32 MCUs User's Guild[online].
Microchip Technology Inc. 2010
URL: http://www.microchip.com/downloads/en/DeviceDoc/51686.pdf.
Accessed 23 December 2009.

7 Controller Area Network.[online].Wikipedia. 9 December 2010
URL: http://en.wikipedia .org/wiki/Controller_area_network.
Accessed 30 June 2010.

8 PIC32MX5XX/6XX/7XX Family Data Sheet, High-Performance, USB, CAN and Ethernet 32.bit Flash Microcontroller [online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/61156D.pdf
Accessed 30 June 2010.

9 ECAN/LIN PICtail Plus Daughter Board User's Guild[online].
URL: http://ww1.microchip.com/downloads/en/DeviceDoc/70319A.pdf.
Accessed 4 July 2010.

10 Mobile Systems Integration Specialization.
URL:http://store.mpcar.com/Centrafuse_Navigation_p/sof-018-0001.htm.
Accessed 20 August 2010

11 Troelson A. Pro C# 2010 and the .Net 4 Platform 5th edition. New York: Apress; 2010.

12 Introduction to XML[online]. W3School.1999-2010.
URL: http://www.w3schools.com/XML/xml_whatis.asp. Accessed 2 Sept. 2010

Appendices

Appendix 1: Reading Serial Port C# code

```
using System;
using System.Collections;
using System.Diagnostics;
using System.Drawing;
using System.IO;
using System.IO.Ports;
using System.Threading;
using System.Windows.Forms;
using System.Xml;
using centrafuse.Plugins;

namespace HelloWorld
{
      public class HelloWorld : CFPlugin
      {
#region Constants
            private const string PluginName = "HelloWorld";
            private const string PluginPath = @"plugins\" + PluginName
+ @"\";
            private const string PluginPathSkins = PluginPath +
@"Skins\";
            private const string PluginPathLanguages = PluginPath +
@"Languages\";
            private const string PluginPathIcons = PluginPath +
@"Icons\";
            private const string ConfigurationFile = "config.xml";
            private const string ConfigSection = "/APPCONFIG/";
            private const string LanguageSection =
"/APPLANG/HELLOWORLD/";
            private const string LogFile= "helloworld.log";
        SerialPort sp = new SerialPort();
#endregion
```

```
        public static string LogFilePath =
Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationDa
ta) + "\\Centrafuse\\Plugins\\" + PluginName + "\\" + LogFile;

#region Local variables

        private const int ImageFolderIndex = 0;
            private const int ImageEmptyIndex = 1;

        private string language = null;
        string s, s1, s2, s3, s4,od;
        int t, t1;
        static int odd;

#endregion

#region Construction

            public HelloWorld()
            {

                  this.CF_pluginPauseAudioFlag = false;

                  this.CF_pluginName = "HELLOWORLD";

                  this.CF_pluginIsGUI = true;

            this.CF_pluginHasSettings = true;
            this.CF_pluginHasBasicSettings = true;

            AddConfigXml();
            sp.PortName = "COM1";
            sp.BaudRate = 9600;
            sp.DataBits = 8;
            sp.Parity = Parity.None;
            sp.StopBits = StopBits.One;
            sp.Open();
        }
```

```
        private void speed( object sender, SerialDataReceivedEventArgs
e)
        {

            while (true)
            {
                s = sp.ReadLine();

                    s2 = s.Substring(3, 2);

                    s3 = s2 + "" + "km/h";
                    Thread.Sleep(500);
                    this.CF_updateText("LBLHEADER", s3);
                    Thread.Sleep(700);
                }

            }
        private void odometer(object sender,
SerialDataReceivedEventArgs e)
        {

            while (true)
            {
                od = s.Substring(5, 1);
                this.CF_updateText("LBLTEXT", od);
                Thread.Sleep(1000);
            }
 }
```

Appendix 2: Label Design with XML

```
<SKIN>
  <APPCONFIG>
    <WIDTH>848</WIDTH>
    <HEIGHT>480</HEIGHT>
```

```
</APPCONFIG>
<HELLOWORLD>
  <X>0</X>
  <Y>84</Y>
  <WIDTH>848</WIDTH>
  <HEIGHT>310</HEIGHT>
  <LABELS>
      <LBLHEADER>
          <WORDWRAP>False</WORDWRAP>
          <FONTCOLOR>#ffffff</FONTCOLOR>
          <FONTNAME>Century Gothic</FONTNAME>
          <FONTSIZE>18</FONTSIZE>
          <FONTSTYLE>Bold</FONTSTYLE>
          <Y>8</Y>
          <X>8</X>
          <WIDTH>832</WIDTH>
          <HEIGHT>24</HEIGHT>
          <ALIGN>Left</ALIGN>
      </LBLHEADER>
      <LBLHELLO>
          <WORDWRAP>False</WORDWRAP>
          <FONTCOLOR>#ffffff</FONTCOLOR>
          <FONTNAME>Century Gothic</FONTNAME>
          <FONTSIZE>36</FONTSIZE>
          <FONTSTYLE>Bold</FONTSTYLE>
          <Y>48</Y>
          <X>8</X>
          <WIDTH>832</WIDTH>
          <HEIGHT>183</HEIGHT>
          <ALIGN>Center</ALIGN>
      </LBLHELLO>
      <LBLTEXT>
          <WORDWRAP>False</WORDWRAP>
          <FONTCOLOR>#000000</FONTCOLOR>
          <FONTNAME>Century Gothic</FONTNAME>
          <FONTSIZE>14</FONTSIZE>
          <FONTSTYLE>Bold</FONTSTYLE>
          <Y>270</Y>
          <X>8</X>
```

```
            <WIDTH>832</WIDTH>
            <HEIGHT>18</HEIGHT>
            <ALIGN>Center</ALIGN>
        </LBLTEXT>
      <LBLTEXTG>
        <WORDWRAP>False</WORDWRAP>
        <FONTCOLOR>#000000</FONTCOLOR>
        <FONTNAME>Century Gothic</FONTNAME>
        <FONTSIZE>14</FONTSIZE>
        <FONTSTYLE>Bold</FONTSTYLE>
        <Y>270</Y>
        <X>32</X>
        <WIDTH>832</WIDTH>
        <HEIGHT>18</HEIGHT>
        <ALIGN>Center</ALIGN>
      </LBLTEXTG>
    </LABELS>
    <BUTTONS>
        <BTNHELLO>
            <ENABLETEXT>True</ENABLETEXT>
            <XOFFSET>0</XOFFSET>
            <YOFFSET>0</YOFFSET>
            <SECONDCOLOR>#0</SECONDCOLOR>
            <FONTCOLOR>#ffffff</FONTCOLOR>
            <FONTNAME>Century Gothic</FONTNAME>
            <FONTSIZE>18</FONTSIZE>
            <FONTSTYLE>Bold</FONTSTYLE>
            <Y>249</Y>
            <X>280</X>
            <WIDTH>312</WIDTH>
            <HEIGHT>48</HEIGHT>
            <ALIGN>Center</ALIGN>
            <TABINDEX>0</TABINDEX>
        </BTNHELLO>
        <BTN1>
            <X>138</X>
            <Y>45</Y>
            <WIDTH>286</WIDTH>
            <HEIGHT>70</HEIGHT>
```

```
        <FONTNAME>Century Gothic</FONTNAME>
        <FONTSIZE>21</FONTSIZE>
        <FONTSTYLE/>
        <FONTCOLOR>#FFFFFF</FONTCOLOR>
        <SECONDCOLOR>#003366</SECONDCOLOR>
        <XOFFSET>0</XOFFSET>
        <YOFFSET>-3</YOFFSET>
        <ENABLETEXT>True</ENABLETEXT>
        <ALIGN>Center</ALIGN>
        <TABINDEX>1</TABINDEX>
    </BTN1>
    <BTN2>
        <X>138</X>
        <Y>121</Y>
        <WIDTH>286</WIDTH>
        <HEIGHT>70</HEIGHT>
        <FONTNAME>Century Gothic</FONTNAME>
        <FONTSIZE>21</FONTSIZE>
        <FONTSTYLE/>
        <FONTCOLOR>#FFFFFF</FONTCOLOR>
        <SECONDCOLOR>#003366</SECONDCOLOR>
        <XOFFSET>0</XOFFSET>
        <YOFFSET>-3</YOFFSET>
        <ENABLETEXT>True</ENABLETEXT>
        <ALIGN>Center</ALIGN>
        <TABINDEX>2</TABINDEX>
    </BTN2>
    <BTN3>
        <X>138</X>
              <Y>195</Y>
              <WIDTH>286</WIDTH>
              <HEIGHT>70</HEIGHT>
              <FONTNAME>Century Gothic</FONTNAME>
              <FONTSIZE>21</FONTSIZE>
              <FONTSTYLE/>
              <FONTCOLOR>#FFFFFF</FONTCOLOR>
              <SECONDCOLOR>#003366</SECONDCOLOR>
              <XOFFSET>0</XOFFSET>
              <YOFFSET>-3</YOFFSET>
```

```
            <ENABLETEXT>True</ENABLETEXT>
            <ALIGN>Center</ALIGN>
            <TABINDEX>3</TABINDEX>
    </BTN3>
<BTN4>
    <X>428</X>
    <Y>45</Y>
    <WIDTH>286</WIDTH>
    <HEIGHT>70</HEIGHT>
    <FONTNAME>Century Gothic</FONTNAME>
    <FONTSIZE>21</FONTSIZE>
    <FONTSTYLE/>
    <FONTCOLOR>#FFFFFF</FONTCOLOR>
    <SECONDCOLOR>#003366</SECONDCOLOR>
    <XOFFSET>0</XOFFSET>
    <YOFFSET>-3</YOFFSET>
    <ENABLETEXT>True</ENABLETEXT>
    <ALIGN>Center</ALIGN>
    <TABINDEX>4</TABINDEX>
</BTN4>
<BTN5>
    <X>428</X>
          <Y>121</Y>
          <WIDTH>286</WIDTH>
          <HEIGHT>70</HEIGHT>
          <FONTNAME>Century Gothic</FONTNAME>
          <FONTSIZE>21</FONTSIZE>
          <FONTSTYLE/>
          <FONTCOLOR>#FFFFFF</FONTCOLOR>
          <SECONDCOLOR>#003366</SECONDCOLOR>
          <XOFFSET>0</XOFFSET>
          <YOFFSET>-3</YOFFSET>
          <ENABLETEXT>True</ENABLETEXT>
          <ALIGN>Center</ALIGN>
          <TABINDEX>5</TABINDEX>
    </BTN5>
    <BTN6>
          <X>428</X>
          <Y>195</Y>
```

```
                    <WIDTH>286</WIDTH>
                    <HEIGHT>70</HEIGHT>
                    <FONTNAME>Century Gothic</FONTNAME>
                    <FONTSIZE>21</FONTSIZE>
                    <FONTSTYLE/>
                    <FONTCOLOR>#FFFFFF</FONTCOLOR>
                    <SECONDCOLOR>#003366</SECONDCOLOR>
                    <XOFFSET>0</XOFFSET>
                    <YOFFSET>-3</YOFFSET>
                    <ENABLETEXT>True</ENABLETEXT>
                    <ALIGN>Center</ALIGN>
                    <TABINDEX>6</TABINDEX>
                </BTN6>
            </BUTTONS>
        </HELLOWORLD>
    </SKIN>
```