ASEE 2014 Zone I Conference, April 3-5, 2014, University of Bridgeport, Bridgpeort, CT, USA.

Reverse Engineering the Volcano CAN BUS Framework for Engine Control Unit Programming

Robert A. Hilton
Department of Electrical and Computer Engineering
University of Hartford
West Hartford, Connecticut
robert.a.hilton.jr@gmail.com

***Abstract*— The goal of this project is to design a combination ISO 9141 (K-Line) and CAN BUS (Controller Area Network Bus) interface in order to load new software into an engine control unit through the diagnostics connector of a vehicle. The specific platform target is a 2006 Volvo S60R. This vehicle requires a high-speed CAN BUS (500kbps) connection and a medium-speed CAN BUS (125kbps) connection. In most older vehicles, the wires in the diagnostics connector are not directly connected to the engine control unit, requiring all data to pass through a CEM (central electronic module), which requires an ISO 9141 keep-alive message to keep the CAN BUS communication path open. This presents the need for custom hardware that is capable of both ISO 9141 and CAN BUS communication, and there are very few devices that exist in the industry that can perform this task. Most devices are impractical, mostly due to the expense. After the hardware was developed, custom software was written for a computer operating system platform in order to actually communicate with the car. This particular test vehicle did not require an ISO 9141 keep-alive message to open the CAN channel, so even though the device is capable of applying the ISO 9141 signal, research efforts were focused on programming the ECU with the CAN BUS. The chosen design successfully accomplished the goal and was able to flash the chosen ECU in approximately 100 seconds, which is approximately 10 seconds faster than a flash done with the factory Volvo tool for the same purpose.**

Keywords—CAN BUS, CEM, ECU, ISO 9141, Volcano, Volvo

I. INTRODUCTION

Many car owners have a desire to be able to reprogram (flash) the software in their vehicles' Engine Control Unit (ECU) through the diagnostics connector [1]. A vehicle owner may want to have different versions of this software; one could be for increased performance while another could be for fuel economy, etc.

Unfortunately, devices to program new software into an ECU may not be available for a specific make and model. In addition the devices that are available can cost upwards of $1200 and can be difficult to obtain.

The immediate goal of this project was to target a specific make and model vehicle and develop a device that can load new control software into the ECU. The eventual goal of this project is to create a generic programmer that can be used to load software into most makes and models of cars.

This device will implement two communication specifications. The first standard is known as CAN BUS (Controller Area Network Bus). This protocol is a differential asynchronous serial protocol in which many modules reside on a single bus. The design of the bus is such that if two or more modules transmit at the same time, the dominant output of all of the modules is used. An output is considered to be dominant if a module is asserting a signal on the bus. The resting voltage for both CAN BUS wires CAN H and CAN L is such that the difference between the two is less than a certain voltage. This is considered a recessive bit and is considered a logical 1. When a dominant bit is asserted (logical 0), CAN H and CAN L are driven and the difference between the two is increased above the minimum detection threshold. The second standard is known as K-Line or ISO 9141 [2]. This is a simple asynchronous serial communication standard that uses a single line for both transmit and receive. Both of these standards are common in vehicle communication, however each manufacturer's specific use of these standards is different. As such, software must be written specifically for each model of vehicle. The chosen test vehicle (a 2006 Volvo S60R) contains two CAN BUS networks [3]. Specific hardware needs to be chosen to support this configuration. Many Volvo models manufactured between the years 1999 and 2007 use a communication layer over CAN BUS called Volcano [4]. The commands that were reverse engineered to complete this project belong to the Volcano command set and are universal to most vehicles that implement this communication layer. Therefore, this particular design will be able to support most models of Volvos between these years.

II. METHODS AND MATERIALS

Much of the information gathered for this project has come from reverse engineering pre-existing equipment. A data-logging device was connected to the CAN BUS while the original factory tool designed by Volvo programmed the ECU of the car. This information was then analyzed and compared with data from production devices and an open source project [5] to further understand the communication layer. The hardware created in this project is based on a

PIC32MX795F512L microcontroller (PIC32) connected to an FTDI (Future Technology Devices International) USB (Universal Serial Bus) to UART (Universal Asynchronous Receiver/Transmitter) device for communication with the host computer. There are two CAN BUS transceivers and a K-Line (ISO 9141) interface to add flashing compatibility to older vehicles. For initial testing purposes, a development board was chosen to implement the program on as a proof of concept. The development board chosen was a chipKIT Max32 board with a chipKIT Network Shield from Digilent Inc. In future research, the microcontroller side of the computer communication will be changed from UART to SPI (Serial Peripheral Interface).

III. Results

In order to decode the communication framework known as Volcano, multiple devices that contain this functionality were examined. A CAN BUS packet logger was programmed on the development board to intercept all traffic related to ECU flashing. The operation speed of the "high speed" CAN BUS was 500kbps and the operation speed of the "medium speed" CAN BUS was 125kbps in the test vehicle. Due to this high data transmission speed, a comparable serial baud rate of 460800 was chosen (approx. 450kbps). Because the baud rate of the serial connection is lower than the baud of the CAN BUS, the software was designed to use a FIFO (First In First Out) buffer to hold the CAN BUS packets and then retransmit them over serial (UART) when the device was capable of doing so. CAN filters were disabled so that all packets on the bus were logged. An issue was encountered in which the data transmission during the flashing procedure was too fast for the PIC32 to keep up with while printing to the serial port. Packets were dropped when the FIFO buffer overflowed. However, enough information was received during the capture to infer the required missing information. This issue will be fixed when the computer communication is switched from UART to SPI.

Fig. 1 - Sample output from the CAN packet logger

```
EID:0xFFFFE 0xFF 0x86 0x00 0x00 0x00 0x00 0x00 0x00
EID:0xFFFFE 0xFF 0x86 0x00 0x00 0x00 0x00 0x00 0x00
EID:0xFFFFE 0x7A 0xC0 0x00 0x00 0x00 0x00 0x00 0x00
EID:0x21 0x7A 0xC6 0x00 0x00 0x00 0x00 0x04 0x04
EID:0xFFFFE 0x7A 0x88 0x00 0x00 0x00 0x00 0x00 0x00
EID:0x21 0x7A 0x8E 0x00 0x00 0x30 0x66 0x84 0x78
EID:0xFFFFE 0x7A 0x9C 0x00 0x31 0xC0 0x00 0x00 0x00
EID:0x21 0x7A 0x9C 0x00 0x31 0xC0 0x00 0x84 0x78
EID:0xFFFFE 0x7A 0xAE 0xE6 0xF4 0x60 0x16 0xF0 0x27
EID:0xFFFFE 0x7A 0xAE 0xF0 0x50 0xF0 0x61 0xF0 0x73
```

The packets with an EID (Extended CAN Identifier) of 0xFFFFE are packets sent from the diagnostic tool. Packets with an EID of 0x21 are responses from the ECU. All of these responses were examined and categorized. Determinations were then made as to the function of the command. For example, we can see the command 7A 9C 00 31 C0 00 00 00. From the structure of the previous and following commands, we can assume that 7A is the module address of the ECU, since all packets sent only to the ECU are prefixed with this byte. We can then infer that the second byte is a function to perform. In figure 1, the byte 9C is showing that the ECU needs to jump to the memory address 0x31C000. Once the first commands were identified, simple assumptions were made concerning the function of the remaining commands and then tested and verified. Corrections were made where necessary. Results were compared with a pre-existing open-source project [5], however the open-source project was never finished**.**

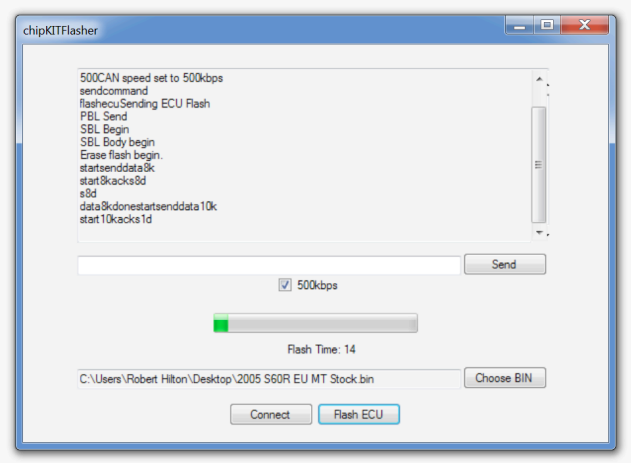
After the basic command set of the ECU was determined, a library of these commands was created. These commands are referenced multiple times from the main firmware of the PIC32. It was determined that the main flash areas of the ECU could be split into 24kB segments and transmitted to the PIC from the computer. This data is stored in the PIC's RAM (Random Access Memory) until the entire 24kB array is built. Once each segment transfer is completed, the data is transmitted to the ECU. The entire ECU file is 1MB, however there are regions of the ECU that are not modified. The flash area from 0x0 to 0x8000 contains the PBL (Primary Boot Loader) [6], of the ECU. This section contains the main execution code for the ECU and is required to stay the same during the CAN flashing process. All addresses from 0xE000 to 0x10000 point to the internal processor RAM, called XRAM. By definition, the RAM is volatile so there is no point in flashing data to that space. This results in a smaller amount of data being transferred over the CAN BUS, with the size shown in Eq. 1.

$$0x100000 - 0x8000 - 0x2000 = 0xF6000 = 984kB \quad \textbf{(1)}$$

Since the data block size is chosen as 24kB, we can determine the number of blocks required to be sent (Eq. 2).

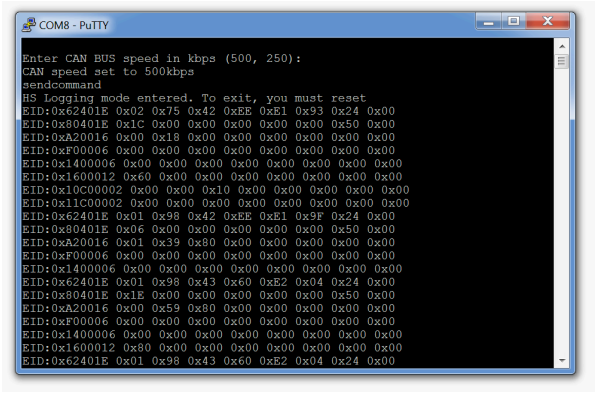
$$984kB / 24kB = 41 \quad \textbf{(2)}$$

Fig. 2 – The graphical user interface of the software



The programming of the ECU was split into two separate parts. There is a firmware component that runs on the PIC32 and a software component that controls the data sent to the PIC32, which provides a GUI (Graphical User Interface). The serial communication between the PIC32 and the computer was designed such that the menu is ASCII (American Standard Code for Information Interchange) driven so the logging functionality of the firmware can be used with any serial client rather than requiring a specific software program to be able to interpret the results.

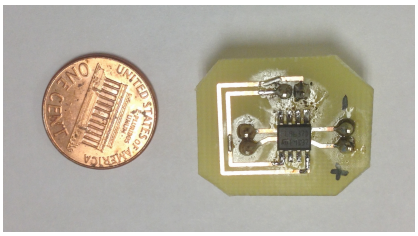
Fig. 3 – A demonstration of the ASCII menu compatibility using a popular serial client.



The software allows the user to select a CAN BUS speed and choose a binary file to flash to the ECU. It also contains functionality to update the checksums of the binary file. These checksums exist to protect against incorrect programming. When a binary file is modified, these checksums need to be updated to reflect the programming changes. This is done so that modified binary files that were manipulated for the purposes of engine tuning will load successfully. The algorithm is open source [5].

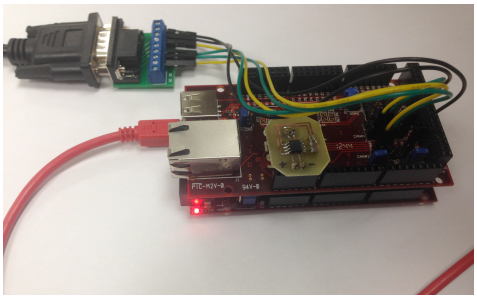
On older model cars, an additional type of communication is required in order to connect to the CAN BUS. The CEM module blocks all traffic from the diagnostics connector to the CAN BUS, unless an ISO 9141 device sends a keep-alive message [7]. However, the chipKIT board does not have an ISO 9141 transceiver available, however. Instead, a separate board was made to handle the ISO 9141 signal and convert it to a UART compatible form [8].

Fig. 4 – The ISO 9141 transceiver breakout board.



Successful flashing of the ECU was verified. Many different model years (2001, 2002, 2004, 2006) of ECU were tested to verify that the software was compatible with all.

Fig. 5 – The full design on the chipKIT max32 board.



IV. Conclusion

The goal of this project was to design a hardware tool that is capable of reloading the software on an engine control unit through the vehicle diagnostics connector. The device described has successfully completed this task. The results of the reverse engineering of the Volcano framework are shown in table 1. The flash time was faster than the factory flash tool using the factory software by an average of 10 seconds. This inspired the question how fast can a flash be done? Using a heavily modified version of a J2534 library, new flash software was implemented for the factory Volvo flash tool known as a DiCE. Once the new software was tweaked appropriately, a full flash was completed in 81 seconds. This was, on average, about 29 seconds faster than the factory Volvo tool and software and 19 seconds faster than the chipKIT flash tool. This shows that there is some room for improvement in the flash time of the chipKIT flash tool.

TABLE I. ECU CAN BUS COMMANDS (ALL VALUES IN HEX)

	EID	D1	D2	D3	D4	D5	D6	D7	D8
Whole Bus Silence	FFFFE	FF	86	00	00	00	00	00	00
Whole Bus Reset	FFFFE	FF	C8	00	00	00	00	00	00
ECU Reset	FFFFE	7A	C8	00	00	00	00	00	00
Start PBL	FFFFE	7A	C0	00	00	00	00	00	00
Response PBL Started	21	7A	C6	xx[a]	xx	xx	xx	xx	xx
Run Code Segment at Jump Point	FFFFE	7A	A0	00	00	00	00	00	00
Response Code Running At Jump Point	21	7A	A0	xx	xx	xx	xx	xx	xx
Jump To Address	FFFFE	7A	9C	00	Address			00	00
Response Jumped to Address	21	7A	9C	xx	xx	xx	xx	xx	xx
Erase	FFFFE	7A	F8	00	00	00	00	00	00
Response Erased	21	7A	F9	xx	xx	xx	xx	xx	xx
Send Data	FFFFE	7A	AE	Data					
End Data Get Checksum	FFFFE	7A	B4	00	Initial Address + Data Length			00	00
Response Data Checksum	21	7A	B1	Chk Sum	xx	xx	xx	xx	xx
Set End SBL (Secondary Boot Loader)	FFFFE	7A	A8	00	00	00	00	00	00

[a] xx indicates the value at this location is irrelevant.

V. Future Work

There are many improvements that can be made to this tool. The first improvement would be to design a dedicated board, rather than using a development board. Using a USB to SPI interface rather than UART would improve data transmission speeds for software flashing and for logging. Another improvement that could be made would be to include an SPI flash storage module for storing different ECU binary files. Using this method, a hardware menu interface could be designed and the device would not require a connection to the computer; it could function as a standalone unit.

Acknowledgments

I would like to express my deepest gratitude to Ying Yu, my research supervisor, for her encouragement and guidance. I would also like to thank John Currie for his constant guidance and support. I extend my appreciation to Bent Aadne Ose for helping to answer my endless questions. I would like to thank Timothy Zimmerman for forcing me to continue my research when my level of frustration became too high and I felt like it was impossible to succeed. I would also like to thank R. Arnold for sharing his apparent lack of knowledge on this research topic and for driving me to complete my research with such a high level of success. And finally, I would like to thank Saeid Moslehpour and Hisham Alnajjar for ensuring I had the resources necessary to complete this research.

References

[1] Society of Automotive Engineers, “SAE J1962: Diagnostic Connector Equivalent to ISO/DIS 15031,” 2001.
[2] Accutest, “K-Line Protocol.” pp. 1–4, 1998.
[3] Volvo Corporation, “Volvo 2006 S60R Wiring Diagram,” *Volvo Wiring Diagrams*, vol. TP 3988202, pp. 34–39, 2006.
[4] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg, “S80 networks Technical concepts,” *Volvo Technology Report*, pp. 1–14, 1998.
[5] Dilemma, “MotronicCommunication.” pp. 1–30, 2011.
[6] Mentor Graphics, “Volcano Bootloader.” pp. 1–2, 2007.
[7] Olaf, “Our mysterious friend, CAN bus,” 2013. [Online]. Available: http://hackingvolvo.blogspot.com/2012/11/our-mysterious-friend-can-bus.html.
[8] STMicroelectronics, “Monolithic bus driver with ISO 9141 interface - L9637,” pp. 1–15, 2013.