5-2014

EcoCAR2 Center Stack Development

Westley Logan Harris
wharri22@utk.edu

Chris Winstead
cwinste2@utk.edu

Nicholas Alexander Cavopol
ncavopol@utk.edu

William Willie Wells
wwells4@utk.edu

Tate Glick Hawkersmith
thawkers@utk.edu

EcoCAR2 Center Stack Development

4/29/2013

Team 1

Chris Winstead

Westley Harris

Nicholas Cavapol

Tate Hawkersmith

William Wells

Dr. David Irick
(Customer)

Executive Summary

EcoCAR2 is a three year student design competition sponsored by the U.S Department of Energy and General Motors. The hybrid electric vehicle market experienced rapid growth in the United States from 2004 to 2011. With the Environmental Protection Agency in the processe of establishing more stringent emissions reductions standards within the next four years, the need for more efficient and environmentally friendly vehicles continues to increase. To produce a successful hybrid vehicle, it must provide the option for luxury amenities that the consumers have come to expect from automotive vehicles, including an easy-to-use, feature-rich, center stack entertainment and control system. The goal of design team one was to develop a center console mounted infotainment system for the UTK EcoCAR, which is a subset of the overall EcoCAR2 project, with a suite of features comparable to vehicle systems currently available in the automotive market. The project incorporated both hardware and software components, but is ultimately classified as a software design project. The project underwent many phases of prototyping and testing using various software techniques and tools in order to design an efficient and functional center stack system. Implementing the system involved many different forms of software engineering and other skills including web development with HTML, CSS, and Javascript, C systems programming, and graphic design using Adobe Photoshop. Using a Freescale Semiconductor i.MX6 SABRE AI board and the QNX CAR software development platform, the team successfully implemented a center stack system for the UTK EcoCAR.

Requirements Specification

1. Hardware Requirements
    a. The team must use an i.MX6SABREAI (board) from Freescale Semiconductor
    b. The team must use a touchscreen display
    c. The hardware system must fit within the Chevrolet Malibu center console space
    d. The hardware systems must have a low enough power demand to be run by the vehicle electrical system
2. Software Requirements
    a. The unit will run on one of the following operating systems:
        i. QNX Neutrino
        ii. Linux
        iii. Android
        iv. Embedded Windows
3. Required Features
    a. The unit must have the following HVAC control capability:
        i. AC on/off control
        ii. Auto temperature control on/off
        iii. Left side temperature control
        iv. Right side temperature control
        v. Left side fan speed control
        vi. Right side fan speed control
        vii. Air circulation source control

viii. Front windshield defrost control

ix. Rear window defrost control

x. Heated seat control

b. The unit must have the following audio control capability:

i. Radio On/Off control

ii. Volume control

iii. Bass control

iv. Treble control

v. Balance control

vi. Fade control

vii. Radio tuning control

c. The unit must be able to display these vehicle diagnostics:

i. Fuel level

ii. Engine coolant level

iii. Transmission fluid level

iv. Brake fluid level

v. Windshield washer fluid level

vi. Tire pressure for each tire

vii. Status of front lights

viii. Status of rear lights

ix. Engine oil level

x. Battery charge state

xi. Battery temperature

xii. Regenerative braking function

xiii. Series operation mode

xiv. Parallel operation mode

xv. MPG information or equivalent

xvi. Additional vehicle statistics, diagnostics, and fault information will be added at the request of the customer as the project progresses. Required information is subject to change as the rules of the EcoCAR2 competition change

4. Time-line requirements.

   a. March 2014 – Pre-Competition Safety and Technical Inspection

   b. May 18, 2014 – Final Technical Report

   c. June 1, 2014 – EcoCAR2 Year Three Final Competition Milford, Michigan

Design Process, Investigation, and Prototyping

Hardware Design and Investigation

Implementation of the center stack system required two key hardware components, an embedded computer and a compatible touchscreen. All of the EcoCAR2 competition participants were donated an i.MX6SABRE AI application board from Freescale Semiconductor. Freescale is a high level sponsor of the EcoCAR2 competition and they have a considerable market presence in automotive application hardware. Due to sponsorship issues, it was a requirement for the center stack development team to use this donated hardware as our main system computer. This board contains the i.MX6 Quad CPU card containing a quad core ARM Cortex – A9 running at up to 1.2 GHz [1]. The Freescale i.MX6 provides robust functionality and a plethora of I/O interfaces necessary for successful implementation of the system. These interfaces include HDMI output, SD card interface, high-speed USB interface, Ethernet interface, Sirius/XM radio module connector, GPS module connector, Bluetooth module connector, CAN interface ports,

and radio input jacks. Retail value of this product is approximately $800.

In order to provide comparison to similar in-vehicle entertainments systems, the design team briefly researched possible alternatives to the Freescale board. Another embedded hardware device option the design team explored was the Raspberry Pi Model-B processor board. The Raspberry Pi includes an ARM1176JZF-S processor running at 700 MHz [3]. An advantage of this processor board is its low cost, selling for roughly $35 dollars. It is also credit card sized, and so would meet the space requirement of the Center stack system perfectly. There are several major disadvantages of this choice, however. First is the lack of necessary I/O interface devices. It might be possible to add these necessary devices through external modules, but would add significant overhead work to the project and would negatively impact the timeline under which the Center stack project is working. Furthermore, if the design team were to go down this path and spend time and effort in an attempt to make the Raspberry Pi an appropriate hardware device and the end result turned out to be that the Raspberry Pi was incompatible with the requirements of the project, it would be an extremely costly and possibly fatal design error. An additional disadvantage is that the Raspberry Pi is not compatible with the QNX operating system and tools, a key software option for the Center stack project.

A third hardware option that was briefly explored was a custom designed board by Intel. The Intel website offers an automotive application board that is subject to design by the customer. As understood, the customer is required to submit a design for their desired board and Intel creates it. This avenue presents an ambiguous route that creates a substantial probability of errors and failure. No members of the design team have enough knowledge in these areas to adequately design an appropriate board. To gain the knowledge to create such a design would add costly overhead work to the project, and would likely result in fatal design errors. Further,

such a board would be expensive and would increase the cost of the project.

Freescale also provided a touchscreen display and controller from a third party company called TTX based in Canada. Initial prototyping and testing were done using this touchscreen, but one major flaw existed in this touchscreen that was unable to be resolved. This flaw was the display of flickering green scan lines when the system was running. Attempts to research and correct this error were unsuccessful. As a result, a new touchscreen display had to be purchased for use with the final system. The touchscreen that was obtained is an integrated display and touchscreen manufactured by Lilliput. This display was compatible with the rest of the system and contained no display flaws when running the center stack user interface.

Software Design and Investigation

The most important software choice that had to be made was what operating system to run on the Freescale board. Freescale documentation stated that four operating systems were compatible with the i.m6 board: embedded Windows, Android, Linux, and QNX. Of these four, embedded Windows and Android were removed from consideration early on. This was due to the results of our own research and recommendations from Freescale representatives. Little or no support exists for developing with embedded Windows and Android on the Freescale i.MX6, and so it was decided that these two operating systems were not feasible choices.

Initial testing and evaluation was done using QNX and Linux. The benefits of using a Linux operating system include the existence of a broad development community with a great amount of knowledge to draw from and the center stack team's familiarity with Linux as a result of academic work within the UTK EECS department. The drawbacks of Linux are that the team would be programming and developing a system from the ground up, and the fact that while

there is a large support community, there isn't necessarily a wide array of support for developing a vehicle infotainment system. A further negative aspect associated with using Linux was the computing power and time required to build a working version of the operating system to launch on the board. The proposed version of Linux was associated with an open source project called the Yocto Project. The Yocto Project is devoted to aiding developers in creating custom builds of embedded Linux. There are approximately 60 GB of software and build files required to create a Yocto build, and some 3 – 4 hours of compiling and building time on a native Linux machine to achieve a successful build. There were multiple errors and problems encountered when attempting to build the Linux OS, and there were also problems using the associated development tools. In addition, there were very few existing tools available to aid development of an automotive entertainment system using Linux.

The second most viable choice for an operating system was QNX. QNX is a Canadian based software systems company that specializes in embedded systems development. In the last few years the company has received attention for their QNX CAR platform. The QNX CAR software development platform is another item donated to the EcoCAR2 teams by QNX. This platform is collection of software tools and programs specifically designed for developing in vehicle entertainment systems. The software package includes the QNX CAR operating system based on the QNX neutrino operating system, the QNX Momentics integrated development environment, the Ripple web emulator for developing user interface applications, the QNX/Blackberry web inspector tool for web application debugging, and a pre-packaged user interface for reference and modification. The QNX CAR platform also provides a number of API's and services that cater to the creation of a center console information and control system. Documentation on developing with this software platform is very comprehensive and available

to the members of EcoCAR2, which is an important feature of any software tool. Also, as a sponsor of the EcoCAR2 competition, QNX has made available a community forum for EcoCAR2 members were software problems related to the project are discussed and resolved amongst members of the EcoCAR2 competition and employees of QNX.

CAN Interface Design and Investigation

Another key aspect of the system is the ability to successfully communicate with the vehicle's CAN networks. The CAN networks in the vehicle are responsible for carrying the information that must be used to display the desired metrics to the end user, as well as controlling the desired functions such as radio and HVAC. For more information on vehicle CAN networks see reference[2]. The vehicle has 3 separate CAN networks that must be accessed in order to extract all of the relevant data. There are two stock GM CAN networks, one that is dual-wire high speed CAN, and one that is single-wire Low Speed CAN. The third network was put in place by the UTK EcoCAR2 controls team and contains information related to the hybrid components installed in the vehicle. It is only necessary for the system being developed to interface with UTK's high speed network and GM's low speed network. The information needed from GM's high speed network can be ported to the center stack system's CAN interfaces by a MicroAutoBox device used by the UTK controls team.

The Freescale i.MX6 board has both low speed and high speed CAN ports. However, initial testing and prototyping showed that the QNX supplied software drivers for those ports did not function properly. As a result, alternative methods of connecting to the vehicle's CAN networks. After research and discussion with QNX, the decision was made to purchase two KVASER CAN interface products. These products are USB interfaces to the CAN bus. A

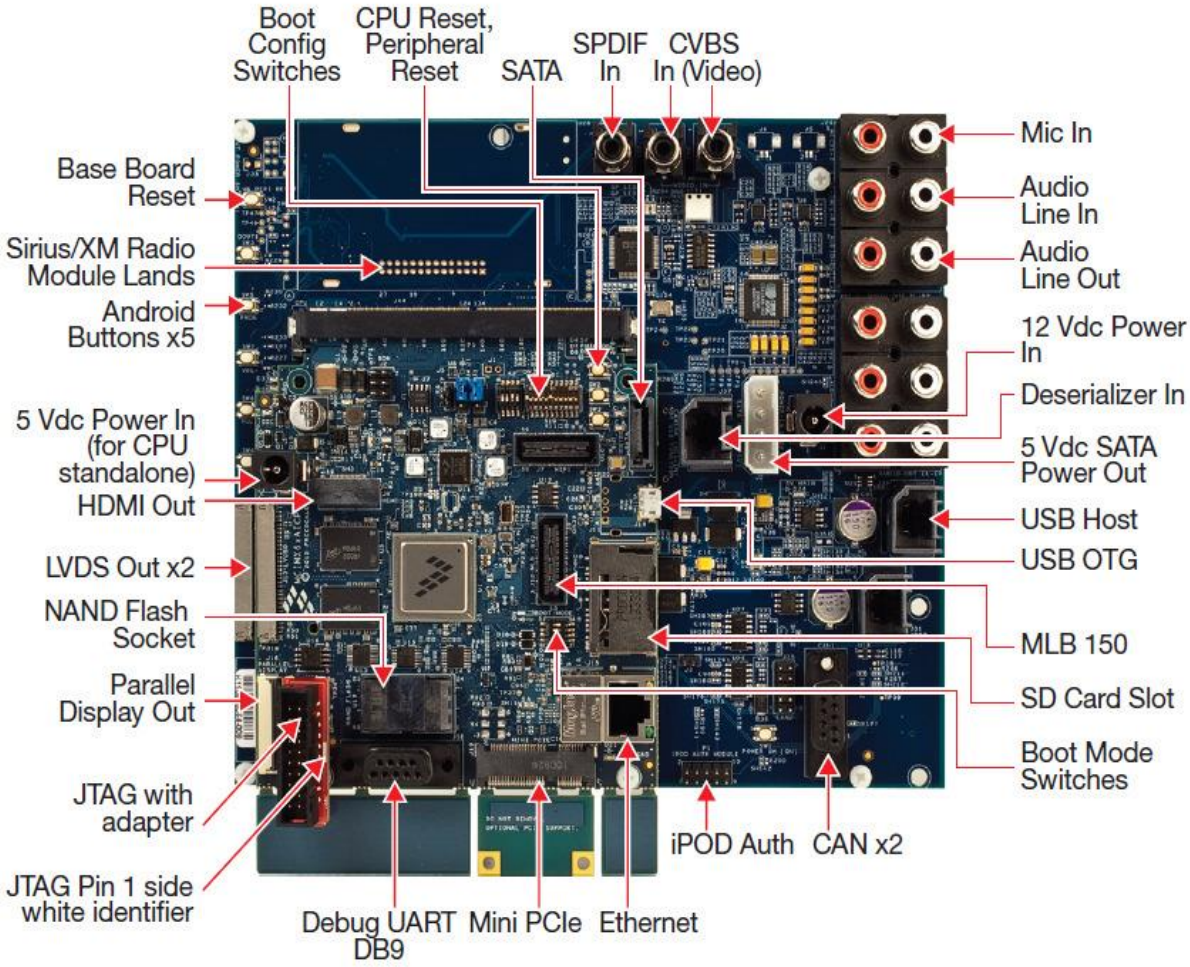
KVASER Leaf Lite HS was obtained for communication with the high speed CAN network, and a KVASER Leaf Semi-Pro SWC was obtained for communication with the low speed CAN network. Initial testing and prototyping with these devices proved successful.

Final Design and Implementation

Hardware Design

The use of the Freescale i.MX6 SABRE AI embedded computer was a project requirement, and so it was chosen as the main hardware component for the center stack system. It provides all the necessary functionality for implementing the system and is a high quality product used in many automotive applications. Below is an image of the Freescale board used in the system.

Figure 1: The Freescale i.MX6 SABRE AI



The final hardware item used in the implementation of the center stack system is the Lilliput integrated display and touchscreen. After initial problems with the donated TTX touchscreen, this item was procured for use with the system. It meets all requirements and performs adequately. Below is an image of the Lilliput display.

Figure 2: Lilliput 869GL Touchscreen[2].

Initial Installation of QNX OS and Overview of Development Environment

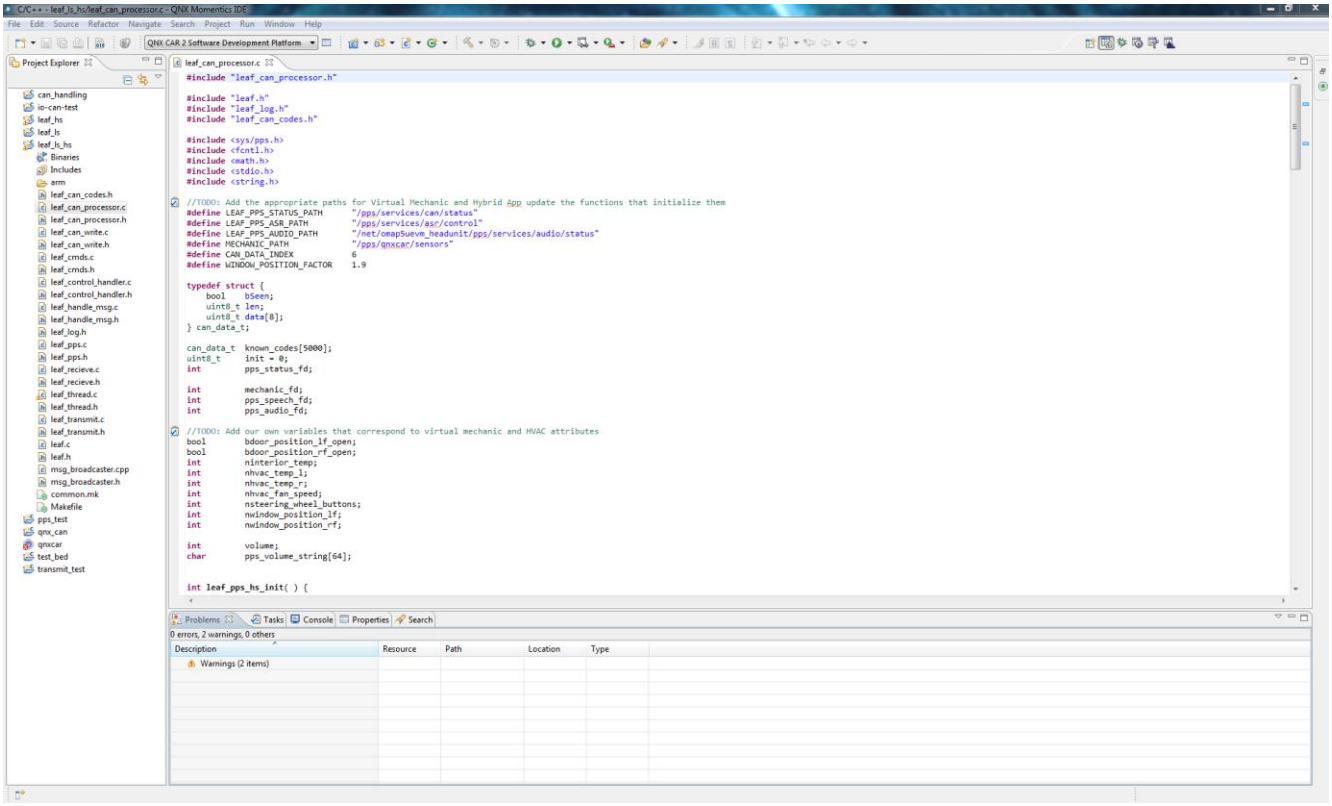
To build an initial QNX operating system image to boot the Freescale board requires several steps. QNX provided a Board Support Package (BSP) for the i.MX6SABREAI, and a QNX CAR image compatible with the i.MX6 hardware. The BSP contained the source code for building a flash image of the QNX Neutrino operating system. The image was built using the QNX Momentics IDE, part of the QNX software development platform. The boot process for i.MX6 board involves booting from a DOS file system partition on an SD card as the board itself has no hard drive. Once initialized, a secondary partition containing the QNX file system is created on the SD card. The SD card is then used as the means of storage for new files and programs added. In order to observe the function of the board and manipulate the boot sequence to correctly initialize the QNX CAR image, the target board must be connected to via serial cable to the host computer and must be connected to using a terminal program (such as PuTTy or similar). Once the initialization steps are completed, the QNX CAR system was installed on the

target. In addition to serial connection to the target board, the QNX Momentics IDE provides the means to connect to the target via Ethernet. This must be done using either an Ethernet hub to which both devices must be connected, or an Ethernet crossover cable to connect both devices together. The team chose to use the crossover method, and had to manually configure the network and IP communications between the host and target. Once a connection was established the QNX IDE provides a clean drag-and-drop file system functionality that allows the programmer to easily manipulate the target file system. The use of SD cards for this project has allowed us to retain multiple versions of the QNX CAR system on separate SD cards, providing a means to track changes made to the system and thus a method of returning to a previous version if irreversible errors are made during modification.

Developers can then use the Momentics IDE to write C or C++ programs to run on the target. By connecting to the target via Ethernet you can launch programs from within the IDE to do testing before you copy an executable to the system. The IDE also provides many different "perspectives" that allow a developer to investigate different aspects of the system or their code. In addition to these features, the IDE provides many common features of IDE's including IntelliSense, code completion, and syntax highlighting.

QNX also provided a web emulator tool called Ripple that allows developers to code web applications within an environment that mimics a vehicle center console. This makes developing the user interface applications much easier than the alternative of making all code changes on the physical system. QNX also includes the Blackberry web inspector tool that allows a system running QNX CAR to connect to a computer over Ethernet, and then view the running applications in a web browser. This enables developers to debug and modify web code in real time.

Figure 3: Screen Shot of the Momentics IDE



User Interface Design and Implementation

Included with the QNX CAR software development platform is a reference user interface. The user interface exists as a collection of standalone web applications written in HTML5, CSS, and Javascript and hosted on a web server included with the QNX CAR operating system. The implementation of a custom user interface was done by modifying these applications and applying new graphics, as well as using the QNX supplied applications as a template for creating a custom application to display hybrid components statistics and data.

A new theme and new graphics were created using Adobe Photoshop software to create and manipulate images used by the user interface applications. Coding work in CSS and HTML was also done to customize the layouts of the default applications. Modification and

development of new user interface components was done using the QNX Ripple web emulator. This tool allows a developer to load their application into a web browser designed to look like a vehicle center console.

The new hybrid specific application was created using HTML, CSS, and Javascript. This application was also developed and tested using the Ripple web emulator. Custom images were created and added to this application to give it a unique look. These images include a CAD model of the actual UTK Malibu that was converted to a PNG image. Once an application is completed in Ripple, it was be packaged and installed on the system. This process involves using another QNX tool called webworks. Webworks is a windows command line tool that packages an application into a bar file that can then be copied to the system and installed. Details of this process can be read in the QNX CAR documentation [3]. Below are various screen shots of the user interface.

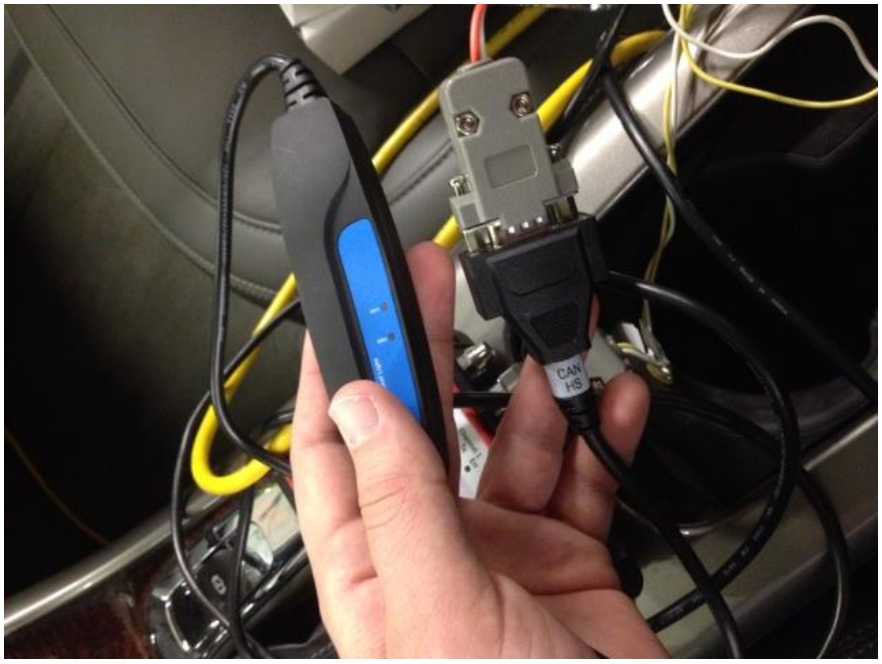
Figure 4: Home Screen

Figure 5: Hybrid Application Screen

CAN Communication Design and Implementation

Interfacing between the center stack system and the vehicle CAN networks was done using two KVASER Leaf USB CAN interface devices. Receiving and processing of CAN data was done by programming in the C language. The programming was done such that for each KVASER device plugged into the system, the product ID is obtained to determine whether the device plugged in is the low speed driver or the high speed driver, and an appropriate thread is launched to handle CAN communication. Subsequent threads for each driver are then launched to handle sending and receiving of data.

Figure 6: A KVASER Leaf CAN Interface Device



The software design for handling and processing CAN data was divided into two sections, a status path and a control path. A status path is the programming in which CAN messages are only received and information is written to the user interface level applications. The control path is the programming which only reads information from the user interface and user input and sends the corresponding messages on the CAN bus. For the purposes of the center stack, radio and HVAC functionality lie only in the control path while the virtual mechanic an hybrid application lie only in the status path. Viewing the code created for these purposes in a status vs control way allowed for easier programming and understanding of the software.

A library of CAN messages was given to the EcoCAR2 team by GM, and CAN message definitions for each of the Hybrid components was provided by the vendor. The center stack team was required to study these CAN message lists to determine which messages needed to be monitored on the system. For each relevant message, programming was added to parse the data

and post appropriate information to the user interface.

The lower level software that intercepts and processes CAN messages and the user level applications are able to communicate through a QNX service called PPS (Persistent Publish/Subscribe). This service provides a number of PPS objects and parameters that exist as specialized files on the file system. The lower level software writes information to these objects using standard *open()* and *write()* calls in conjunction with a QNX API for encoding and decoding PPS objects with parameters, and the user interface applications extract that data through Javascript API's provided by QNX.

Testing and Evaluation

UI Testing

Team One's Testing approach consisted of three avenues. These avenues aligned with the division of responsibilities of the team, and consisted of UI Development and Testing, CAN network communication and testing, and full system integration and testing.

The UI aspect of the project was developed by team members Nicholas Cavopol and Chris Winstead. This section of the project included modification of the aesthetics of the HTML5 applications that the center stack system uses to communicate with the end user. The QNX car package donated to the team contained a number of pre-loaded applications which were used by the team, and in these instances the team modified the CSS and HTML5 code of these applications to produce the desired UI. Additionally, a custom application relating to the hybrid features of the car was created by Willie Wells, and the user interface of this application was also modified by the UI development team. Both the pre-packaged applications and the custom

application were tested in two ways after being modified by the UI team.

The first way in which the UI was tested was a visual review by members of the team not connected to UI development. In the beginning stages, different visuals and themes were created and then presented at weekly meetings. The approval of these graphics were subjected to team voting and, upon approval, were integrated into the applications in the center stack system. When the UI of the application was completed, they were again presented to the team and subject to final approval by the team.

The second objective of UI testing was to ensure that the modified applications would correctly display information sent to them. The user interface communicates with vehicle's CAN network through a QNX service called PPS (Persistent Publish/Subscribe). This service allows the applications to persistently monitor the PPS file object for any updates to the vehicle's operation, and subsequently display them through the different applications. Testing the operation of this system involved manually writing data into the PPS service and then monitoring the UI for updates. An additional method of testing the interface of the UI and the CAN network was the Blackberry Web Inspector development tool provided by QNX. This system allows a developer to connect a computer to the Freescale board through USB and Ethernet. This provides console and debugging tools that allow easier testing and monitoring of the UI applications. All testing of the UI was performed by team members not involved in development of the user interface, and in the case of the custom hybrid application, the testing was performed by Westley Harris to ensure unbiased testing. Ultimately all testing yielded satisfactory results and the UI was successfully implemented. Below are screenshots of the testing of the HVAC functionality.

```
# cd /pps/qnxcar
# cat hvac
@hvac
airCirculation_setting:b:false
airConditioning_enabled:b:false
fan_setting_l:n:3
fan_setting_r:n:1
fan_speed_l:n:1
fan_speed_r:n:3
fan_temperature_l:n:21
fan_temperature_r:n:19
heatedSeat_level_l:n:2
heatedSeat_level_r:n:0
rearDefrost_enabled:b:true
zoneLink_enabled:b:false
# echo "fan_speed_l:n:4" > hvac
#
```

Figure 6: Commands to manually change PPS HVAC data.

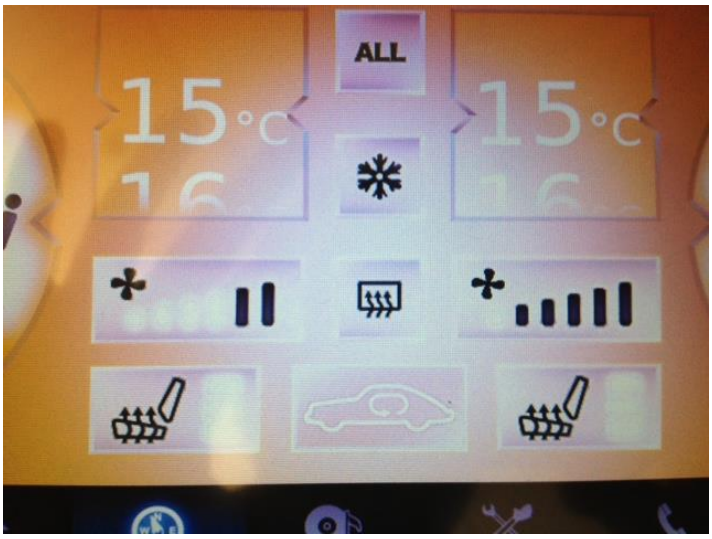


Figure 7: Updated UI Fan setting

CAN Communication Testing

The CAN network interface was developed using two KVASER Leaf USB interface can drivers, and C code that processed the information. Testing was performed incrementally with a member of the EcoCAR2 Controls Engineering team outside of the center stack team. Initial testing of the CAN communication interface involved the use of a Vector CANCaseXL and Vector CANoe software, which simulates a CAN bus. This testing was done to ensure that CAN messages could be read by the center stack system. Upon successful testing using the CAN bus simulation, the system was then connected to the vehicle's CAN network, and the CAN output was monitored. CAN logs showing successful reading of CAN data can be seen below in Figures 8 and 9.

```
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 28 05 00 40 00 00
Received can msg: ID=0x214,     len=6,  flags=0,    msg= 06 00 01 fd 00 00
Received CAN message for Brake Pedal Status
Received can msg: ID=0x2f9,     len=7,  flags=0,    msg= 3b 00 10 00 00 00 00
Received can msg: ID=0x135,     len=8,  flags=0,    msg= 00 00 18 58 36 0b 0e 67
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 00 01 00 04 00 01 00 04
Received can msg: ID=0xc5,      len=8,  flags=0,    msg= 00 00 00 00 00 01 00 04
Received can msg: ID=0x348,     len=5,  flags=0,    msg= 00 00 00 00 1b
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 34 05 00 40 00 00
Received can msg: ID=0xc7,      len=4,  flags=0,    msg= 00 00 00 00
Received can msg: ID=0xf9,      len=8,  flags=0,    msg= 00 00 40 00 00 00 00 95
Received can msg: ID=0x139,     len=8,  flags=0,    msg= 00 00 00 00 00 00 00 00
Received can msg: ID=0x18e,     len=8,  flags=0,    msg= 00 00 00 00 00 00 06 2e
Received can msg: ID=0x1bc,     len=7,  flags=0,    msg= 00 00 00 00 00 00 00
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 10 01 00 04 10 01 00 04
Received can msg: ID=0x1e1,     len=7,  flags=0,    msg= 00 00 04 00 00 00 00
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 00 04 00 40 00 00
Received can msg: ID=0x185,     len=2,  flags=0,    msg= 00 03
Received can msg: ID=0x1f1,     len=8,  flags=0,    msg= 80 0e 00 00 08 00 00 7a
Received can msg: ID=0xc7,      len=4,  flags=0,    msg= 00 00 00 00
Received can msg: ID=0xf9,      len=8,  flags=0,    msg= 00 00 40 00 00 00 00 95
Received can msg: ID=0x1f3,     len=3,  flags=0,    msg= 00 00 00
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 20 01 00 04 20 01 00 04
Received can msg: ID=0x1e5,     len=8,  flags=0,    msg= 44 06 e8 70 02 00 03 09
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 1c 06 01 40 00 00
Received can msg: ID=0x189,     len=8,  flags=0,    msg= 8f ff 0f ff 2f ff ff 0c
Received can msg: ID=0x199,     len=8,  flags=0,    msg= 8f ff 0f ff ef ff 00 ff
Received can msg: ID=0x1f5,     len=8,  flags=0,    msg= 0f 0f 00 21 00 00 03 00
Received can msg: ID=0x1f3,     len=3,  flags=0,    msg= 40 40 00
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 30 01 00 04 30 01 00 04
Received can msg: ID=0xc5,      len=8,  flags=0,    msg= 30 00 00 00 30 01 00 04
Received can msg: ID=0x1eb,     len=2,  flags=0,    msg= 00 00
```

Figure 8: High Speed CAN Data Log

```
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x624,    len=8,  flags=0,    msg= 00 20 00 00 00 00 00 00
Received can msg: ID=0x621,    len=8,  flags=0,    msg= 00 52 00 00 00 00 00 00
Received can msg: ID=0x62c,    len=8,  flags=0,    msg= 00 40 00 00 00 00 00 00
```

Figure 9: Low Speed CAN Data Log

Once it was confirmed that the CAN data could be read, C code was written by team member Westley Harris that could process said data, so that it could be displayed on the user interface. This was tested by connecting the user interface to the CAN drivers, and monitoring the resulting changes on the display. These tests were performed in tandem with a member of the UT controls team to ensure independence of testing. Figs. 10, 11, and 12 show the sequence of the initial UI display, the CAN log showing retrieval of updated status, and the subsequent changes to the UI.

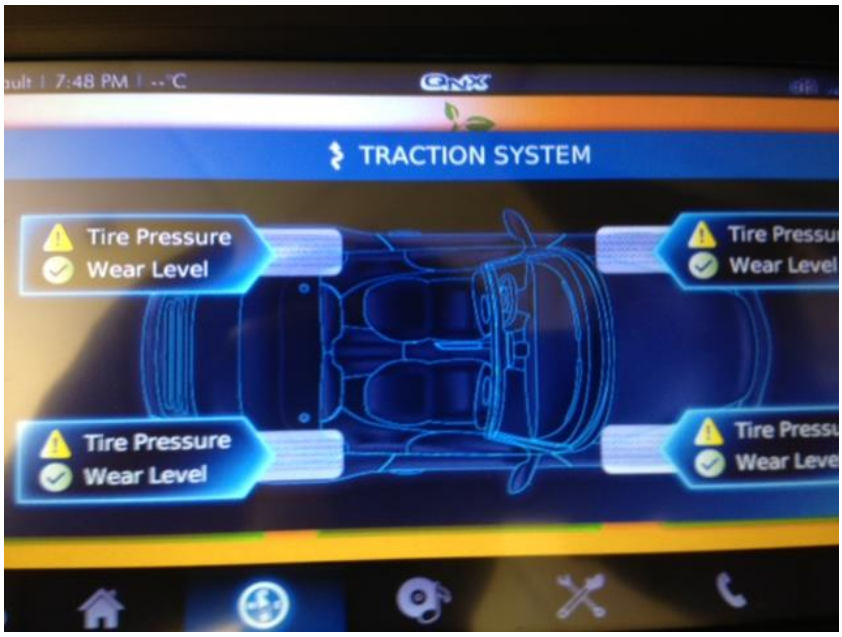


Figure 10: Display Showing Initial Tire Pressure Warning

```
Received can msg: ID=0xf9,      len=8,  flags=0,    msg= 00 00 40 00 00 00 00 95
Received can msg: ID=0x52a,     len=6,  flags=0,    msg= 00 00 37 37 37 37
Received CAN message for Tire Pressure Sensors
Received can msg: ID=0xc5,      len=8,  flags=0,    msg= 10 00 00 00 10 01 00 04
Received can msg: ID=0x1e9,     len=8,  flags=0,    msg= 00 00 00 0e 8f fc 60 00
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 00 05 00 40 00 00
Received can msg: ID=0x185,     len=2,  flags=0,    msg= 00 03
Received can msg: ID=0x189,     len=8,  flags=0,    msg= 0f ff 0f ff 30 01 ff 0c
Received can msg: ID=0x19d,     len=8,  flags=0,    msg= 00 00 00 00 00 00 00 ff
Received can msg: ID=0x77f,     len=7,  flags=0,    msg= 00 00 00 00 00 00 00
Received can msg: ID=0x1f3,     len=3,  flags=0,    msg= 80 80 00
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 20 01 00 04 20 01 00 04
Received can msg: ID=0x1e5,     len=8,  flags=0,    msg= 44 06 e8 70 00 00 03 07
Received can msg: ID=0xc9,      len=8,  flags=0,    msg= 00 00 00 07 00 10 08 00
Received can msg: ID=0x1ba,     len=8,  flags=0,    msg= 16 a0 6a 06 2d 6c 67 83
Received can msg: ID=0xf1,      len=6,  flags=0,    msg= 1c 05 00 40 00 00
Received can msg: ID=0xc1,      len=8,  flags=0,    msg= 30 01 00 04 30 01 00 04
Received can msg: ID=0xc5,      len=8,  flags=0,    msg= 30 00 00 00 30 01 00 04
```

Figure 11: CAN Log Showing Successful Monitoring of CAN Updates

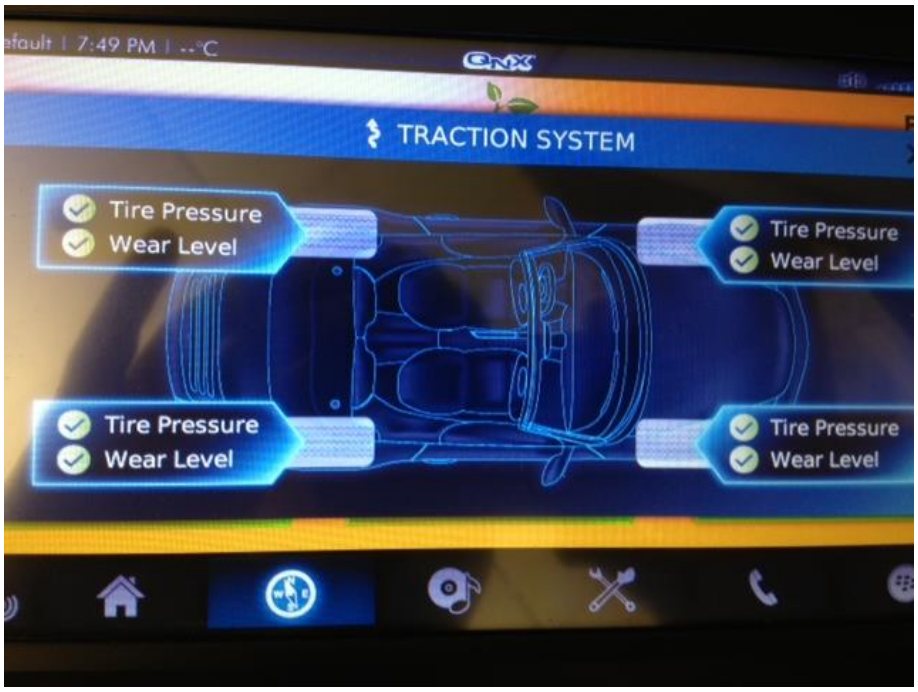


Figure 12: Display Showing Newly Updated Tire Pressure Status

Full System Testing

Full system testing will done by integrating the overall system into the vehicle and launching the CAN processing program. The resulting output and functionality will be tested and verified by the members of the center stack development team, controls team and the EcoCAR2 team leaders and supervisor. This final stage of testing has not yet been completed as aspects of the vehicle necessary for this test are still being modified by the EcoCAR2 mechanical and controls teams. This testing will be performed at a later date, before the late May deadline in place for completed modification of the vehicle.

Conclusion and Final Remarks

The EcoCAR2 center stack development team was tasked with creating a fully functional center stack display with a unique interpretation that would showcase the University of Tennessee team's contributions. Consumer acceptability was a required focus in the development and as such, care was taken to allow easy access to the functions and metrics that are important to the average driver. Ultimately these controls included at a minimum, full audio control, full HVAC control, and an adequate display of performance metrics related to the use of both the traditional combustion engine and the electric motor. A full list of the included functions can be seen in the requirements section.

The current EcoCAR2 center stack team began work on the system in the third and final year of the overall competition. As such, transfer of knowledge was expected from the veteran members of the team as to the tools and goals of the project. However, this infrastructure was not in place, and the team was forced to start from scratch. Two team members were brought to a training conference and introduced to the tools that would be used, which was valuable assistance, but ultimately the learning curve of using such unfamiliar tools was sharp, and early progress on the project consisted entirely of familiarization of the development environment the team would be using. Once the team's bearings were set, the decision was made to divide team responsibilities into the avenues of UI development, CAN communication and network development, and the creation of the custom hybrid operation application. All avenues were developed and tested independently.

While the construction of a correctly functioning system was the premier accomplishment, a secondary objective that the team took upon itself was the collection of meaningful and organized documentation of the design process. This was done using tools such

as the GIT repository provided to the team, and will be passed on to the team competing in the next competition, EcoCAR3.

As the conclusion of the project fast approaches, all team members can say that they have gained invaluable knowledge and experience in not only the creation and integration of automotive center stack system and controls, but also the nuances of working as a part of a sub-team of the greater overall team working on the vehicle.

Team Member Contributions

1.) **Team Leader** - Chris Winstead

As the team leader, Chris Winstead served as the main liaison through which various administrative functions were coordinated. This includes coordinating both Senior Design responsibilities such as calling meeting and delegating responsibilities, and coordinating responsibilities related to the greater EcoCAR2 project. Additionally Chris served as lead tester, and as such developed a working knowledge of the tools and development environment necessary to create the custom applications required by the project.

2.) **Solutions Architect and Lead Report Writer** - Westley Harris

As solutions architect, Westley Harris maintained the hardware components and main operating system SD card that holds the current system software. He researched and provided documentation for setting up the QNX software development platform tools to other team members. He is responsible for task delegation related to research and implementation of system components and custom applications. He was the lead

programmer for CAN network communication and took part in extensive testing of CAN message processing. As lead report writer, he was responsible for the majority of written reports, and delegating review and editing to other team members.

3.) **Researcher and Lead Presenter** - Nicholas Cavapol

As lead presenter, Nicholas spearheaded the creation of presentations as well as deciding how the presentations would be split up between all of the team members. This also meant taking lead on poster sessions and helping to keep a cohesive presentation. Additionally, Nicholas researched viable options for the design throughout the process. Most notably, Nicholas undertook the graphical design of the hybrid app that Willie implemented. This included designing a suitable and informative icon to be used as the home screen button.

4.) **Lead Designer and Librarian** - Tate Hawkersmith

Tate managed the teams GIT repository by saving our documentation, code, reports, presentations, and any other documentation created. As Lead Designer, He developed potential designs, themes, and color schemes for the interface and the features we will be developing. He helped the team with the software initialization and center stack system, control flow design, ripple simulation, testing, and board implementation. He also helped with research for the next semester for the displaying CAN messages and implementing a virtual machine on the SD card used on the Freescale board.

5.) **Tester and Reviewer** - William Wells

As a tester, William was responsible for various software testing and configurations. He researched the feasibility of working with the Yocto project and Qt software for user interface development using a Linux virtual machine. He researched the use of HTML5 and Blackberry WebWorks as tools to create custom infotainment system applications. He also researched and worked with the Ripple Emulator as a means of testing user interface functionality. As a reviewer, he reviewed all presentations and reports and provided appropriate feedback. He used the QNX Virtual Mechanic application as a template to design and test a hybrid application. The hybrid application displays some of the hybrid features of the newly converted hybrid Malibu. The successful operation of the application involved synchronization between javascript, cascading style sheets, and hypertext scripts. He instructed Chris in cascading style sheet editing. He tested images and color schemes as they were made available by Nick and Chris.

References

1) "SABRE for Automotive Infotainment Based on the i.MX6 Series." http://cache.freescale.com/files/32bit/doc/fact_sheet/IMX6SABREAIFS.pdf?fpsp=1&WT_TYPE=Fact%20Sheets&WT_VENDOR=FREESCALE&WT_FILE_FORMAT=pdf&WT_ASSET=Documentation. Freescale Semiconductor, 2012. Web. 22 Nov. 2013.
2) "Controller Area Network (CAN) Overview". http://www.ni.com/white-paper/2732/en/ . National Instruments. Web. 28 April, 2014.
3) "QNX SDP Documentation". http://www.qnx.com/developers/docs/qnxcar2/index.jsp. QNX Software Systems, 2012. Web. 28 April, 2014.

Signatures

Chris Winstead _____

Westley Harris _____

Nicholas Cavopol _____

Tate Hawkersmith _____

William Wells _____

Dr. David Irick, Customer _____