Department of Electrical and Computer Engineering

SAE Formula Car Display & Data Acquisition System

Authors: Joseph Groe, Miles Homler, Michelle Ohlson

Advisor: Professor Steven D. Gutschlag

May 12, 2017

Abstract

A data acquisition system was built to assist the Bradley University mechanical engineering team in the annual SAE formula car race. Previous vehicle failures with the formula car have prevented the team from competing, so the intent of this system is to permit the team to monitor the car for warning signs associated with various failures. Potential catastrophic failures can be identified before they occur through real-time information sent to the driver and crew, including a warning system that notifies the team when critical parameter values no longer lie within acceptable ranges. The system uses a microcontroller to read data from sensors on the car, converting the data to the proper format before sending it to the driver's display and a wireless transceiver that transmits the data to a track-side laptop. This system would not only help ensure the safety of the Bradley University Formula SAE team, but would provide the team with a continually updated overview of the car's operational status. Unfortunately, shortly after this project was started the Bradley University Mechanical Engineering Formula SAE team disassembled the vehicle that was originally specified as the target for the finalized system. Therefore, the completed data acquisition system could not be tested on a Formula SAE car.

I. Introduction and Overview

Problem Background

The Society of Automotive Engineers (SAE) holds an annual competition for teams of engineering students to design and build a single seat race car according to rules provided by the SAE. The teams' cars will then be inspected and subjected to various performance tests at the competition to judge its design, construction, and performance.

In past years, the cars built by the Bradley mechanical engineering team have been unable to race due to performance issues. To ensure the car is running properly during testing and development, the Electrical and Computer Engineering department SAE Formula Car Display and Data Acquisition System design team will design a touch-screen display and data acquisition system to interface with sensors on the car. The sensors will read information about the car, such as engine speed, oil pressure, and ground speed in MPH. A GUI for the driver will display sensor information as real time data from the car, or data generated by an onboard microcontroller to test the display and provide a demonstration mode for judging during the SAE competition. In addition, a wireless system will be set up for the car to send vehicle data to be displayed and stored on an offsite laptop to permit monitoring vehicle operation in real time for added reliability and safety. The stored information will also permit data analysis at a later time to improve vehicle performance.

The display on the car will show the driver the most important information. This system will be used primarily to provide engine speed and warn the driver when a vehicle system has failed or

is about to fail. As indicated above, the pit crew will also be able to monitor the car and warn the driver in case of failure. This is a very useful feature because the driver will frequently be busy driving instead of watching the display, and at least one engine has already been destroyed because the driver failed to see a critical temperature warning. It should be noted that although the students driving the Formula SAE car do the best they can, they are not professional race car drivers. Hence, they are not trained to regularly monitor the instrumentation for indications of imminent vehicle failure. For example, if the engine loses oil pressure or severely overheats, the engine can “lock-up” and scatter failed engine components, oil, and coolant over a wide area of the road course. The slippery oil and coolant can result in complete loss of traction, sending the vehicle spinning off of the roadway, quite possibly resulting in a serious accident. In addition, since the engine is transversely mounted in the vehicle, it is also possible that the scattering failed engine components can penetrate the firewall and seat, then enter the driver’s back. The pit crew can also monitor far more important information via the offsite laptop than could possibly made available to the driver (e.g., suspension travel which could indicate a tire losing pressure).

Review of Literature and Prior Work

Numerous times over the past 15 years the Electrical and Computer Engineering department at Bradley University has tried to collaborate with the mechanical engineering department for the SAE Formula car competition. Since part of the SAE competition includes judging the aesthetics of the car, the touch-screen display system alone would significantly improve the car’s rating during the judging portion of the competition. The SAE competition requires that all work is completed by non-professional students with a large emphasis on reliability, cost, and ease of maintainability. Unfortunately, for a variety of reasons the Mechanical Engineering department has never been able to provide a vehicle for the ECE students to mount the data acquisition system for testing.

Previous Bradley Electrical and Computer Engineering teams have successfully completed this project before using an Amulet Touchscreen display and an ATmega128 microcontroller. However, one of the objectives of this variation on the SAE Formula Car Display and Data Acquisition System was to develop a system with a Controller Area Network (CAN) since that technology is frequently used in industry. Therefore, the team used J1939 CAN bus for communication between subsystems, and needed to upgrade to the Parker IQAN MD4-7 touch-screen display and AT90CAN microcontroller.Problem Statement

System Overview

Requirements

The wireless transmission should be reliable, accurate, and have a minimum range of one mile. The on-car display should display critical sensor data to warn the driver of any imminent failures. The track-side laptop GUI should also display all critical sensor data, as well as accomplish data

storage and retrieval. The microcontroller should collect sensor data, and transmit the sensor data to the wireless receiver and the on-board display.

System Block Diagram

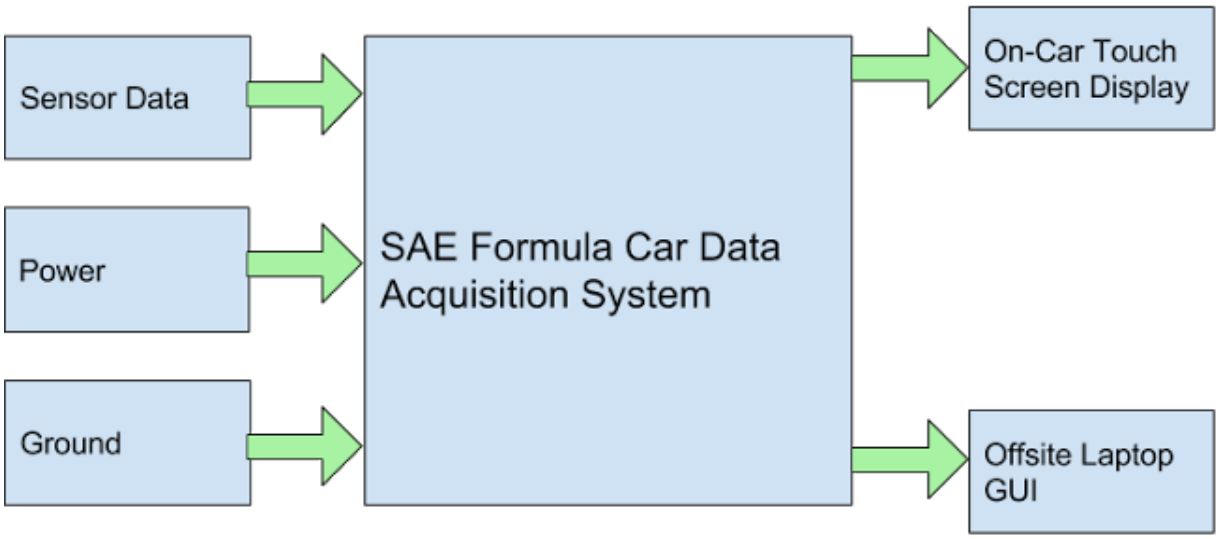

Figure 1: System Block Diagram

Subsystem Block Diagram

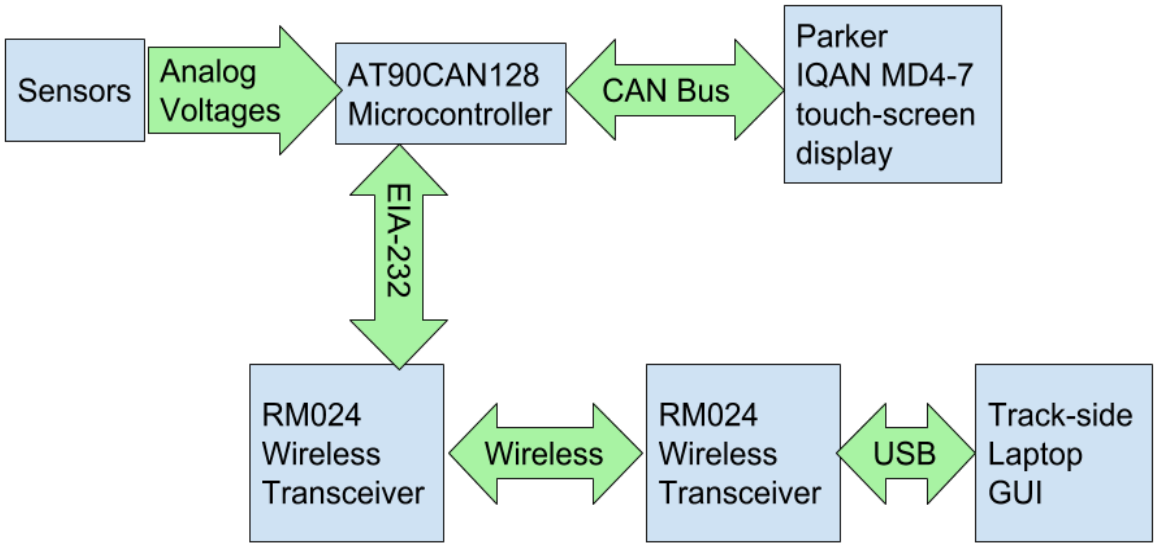

Figure 2: Subsystem Block Diagram

Division of Labor

Joseph Groe - Display software, sensor inputs
Miles Homler - Sensor inputs, wireless transceiver outputs
Michelle Ohlson - Wireless transceiver software, computer GUI, data storage and retrieval

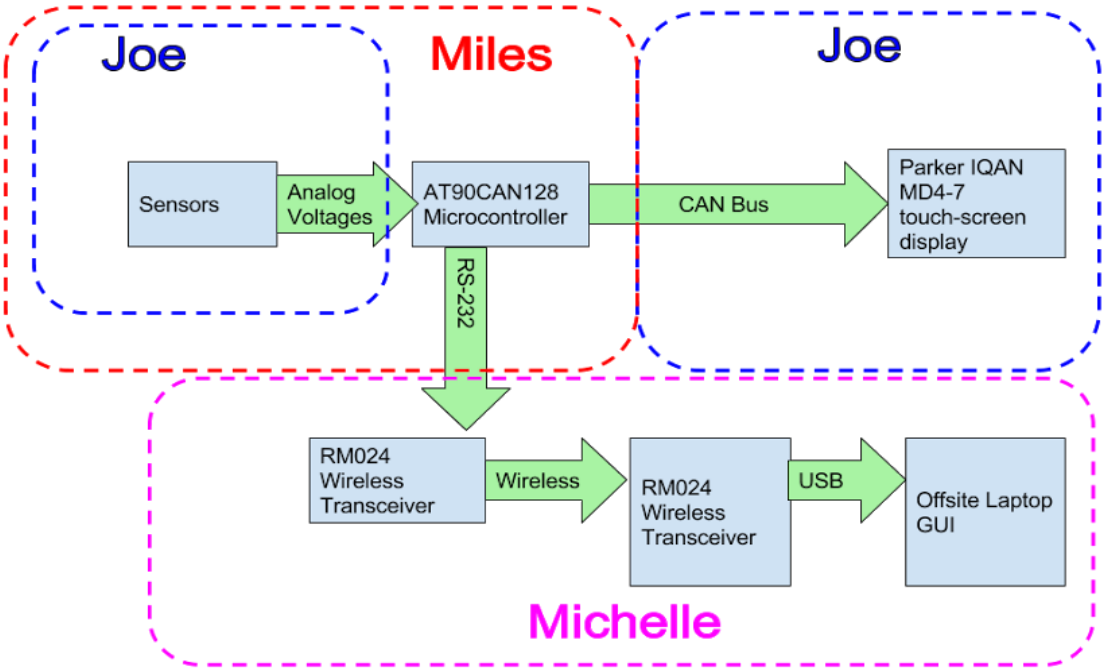

Figure 3: Division of Labor

Parts List

Table 1: Parts List

Part	**Price**
Display Harnesses	$100 (Donated by Parker-Hannafin)
IQAN MD4-7 display unit	$1,200 (Donated by Parker-Hannafin)
RM024 Transceiver Development Kit	$129.52
Sensors Sensors (All sensors already available from the project advisor)	$600
AT90CAN128 Development Kit	$129

Hardware

AT90CAN Development Kit

The group decided on an AT90CAN128 microcontroller for this project. The board features both CAN and EIA-232 support, and the group had a good amount of experience with Atmel microcontrollers because the ATmega128 was used by all team members in previous laboratory courses.

The AT90CAN128 is embedded on a development kit, the DVK90CAN1 (also from Atmel), and includes many options and functions beyond those implemented in the project. The board includes LEDS, five directional keys, and assorted sensors, as well as two 9-pin serial ports, which the team used to output EIA-232 and CAN messages. Multiple power options are also available: powering the chip directly from an external source, or powering just the board from an external source, where the board will supply a regulated 3V or 5V (user's' choice) to the microcontroller. Once the team starting testing CAN communication, a regulated supply of 5V was chosen, as activity was only present on the CAN bus at 5V. At 3V, no messages were being sent, and scoping the bus just showed 0V, regardless of what values the microcontroller attempted to send.

RM024 Development Kit

The group had to find a replacement for the AC4790-200A transceivers used in previous iterations of this project. The ECE department had only one functional transceiver module left, and the AC4790-200A is now an obsolete part. However, the AC4790-1000M is currently available and is pin compatible, so it can be used with the same Aerocomm development kit and software that came with the AC4790-200A.

Another transceiver that met the requirements was the RM024, which is still sold in a development kit. Both transceivers have adequate range, and can communicate using RS232. The RM024 is available in a development kit with everything needed for the project. In contrast, the AC4790-1000M does not come in a kit and requires antennas that must be ordered separately. The AC4790-1000M datasheet has a list of approved antennas, and the two approved antennas that are still manufactured are $16 each, bringing the cost for the AC4790-1000M option to $187.40.
The RM024 is cheaper, has a higher serial interface data rate, higher RF data rate, and it comes in a kit so the team was sure that everything will work together. It also has smaller power consumption when transmitting, so it was the logical choice for this project.

IQAN MD4-7 display unit

Donated by Parker-Hannafin Corporation.

Display Harnesses

Donated by Parker-Hannafin Corporation.

Sensors

The sensors used are two Prosense TTD25N-20-0300F-H temperature sensors to measure engine oil and coolant temperature, a Prosense PTD25-20-0100H pressure sensor to measure engine oil pressure, an AiM X05SNVS00 RPM sensor, and four Black Diamond LX-PA string potentiometers to measure suspension travel.

Software

Table 2: Software Programs

Software Program	**Use**
Laird Configuration/Test Utility	RM024 Wireless Transceiver configuration and testing
LabVIEW Student Edition	Laptop GUI and data storage & retrieval
AtmelStudio	AT90CAN128 Microcontroller programming
IQANdesign	IQAN-MD4 Display programming

Detailed Test Procedures

Wireless Transceivers

The first step was establishing communication between the two RM024 transceivers using the Laird Configuration/Test Utility. Both wireless transceivers were connected to the Laird Configuration/Test Utility on one laptop, and the Laird software Range Test feature indicated that the transceivers were capable of communicating. Then the transmitting RM024 was connected to the Laird Configuration/Test Utility and the receiving RM024 was connected to PuTTY on the same laptop. The group attempted to connect the receiving RM024 to the LabVIEW GUI on the same laptop, but there was an error because the Laird Configuration/Test Utility was preventing LabVIEW from using the com port, so two laptops were needed. The transmitting RM024 was connected to one laptop running the Laird Configuration/Test Utility, and the receiving RM024 was connected to another laptop running the LabVIEW GUI, which successfully received the same data sent from the first laptop. Further testing of the wireless transceivers was completed when LabVIEW and the microcontroller's EIA-232 output were tested, which is explained below.

LabVIEW GUI

The transmitting RM024 was connected to one laptop running the Laird Configuration/Test Utility, and the receiving RM024 was connected to another laptop running the LabVIEW GUI. The LabVIEW GUI was run using the "Highlight Execution" feature, which shows the data transmitted between different blocks in the code. Based on the data indicated using the LabVIEW "Highlight Execution" feature, the transceivers successfully completed the

transmission.

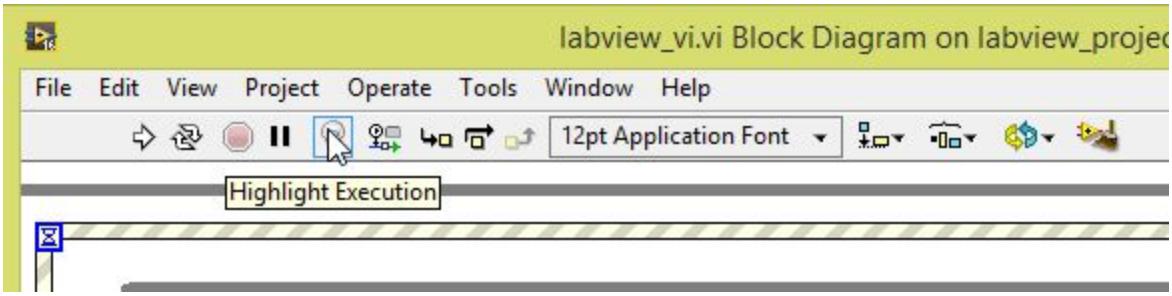

Figure 4 LabVIEW Highlight Execution Feature

Microcontroller EIA-232 Output

First the microcontroller was connected to the transmitting RM024 wireless transceiver via the serial port on the DVK90CAN1 board, and the receiving RM024 wireless transceiver was connected to a laptop running the Laird Configuration/Test Utility. The microcontroller sent 'Hello\n' in an infinite loop, which was displayed on the Laird Configuration/Test Utility. This tested the microcontroller and wireless only, without involving LabVIEW.
After that, the receiving RM024 was connected to the LabVIEW GUI, and the microcontroller sent the eight, 16-bit unsigned integers followed by a newline in an infinite loop. This tested the microcontroller, wireless transceivers, and LabVIEW as combined subsystems.

Wireless Transceiver Range Testing

To test the wireless operational range, the group used the wireless transceivers, the microcontroller, and the laptop connected as shown in the subsystem block diagram in Figure #. The group tested the chip antenna first since the datasheet claimed that it would have a range of 2.5 miles. One group member stayed at the bench by Jobst Hall with the laptop and one wireless transceiver. The other team members held the other wireless transceiver and microcontroller while walking towards the Bradley library. The connection started failing when the group members reached Bradley Hall, a distance of about 450 feet (estimated using Google Maps). When the test was repeated with the dipole antennas, the group members were able to get farther before the connection failed (just past Bradley Hall) at approximately 600 feet, but they were not able to reach the Bradley library. The Bradley University wifi system and the team's wireless transceivers both operate in the 2.4 GHz band. It is therefore possible that the Bradley wifi system could be causing interference, and may be the reason for the short range.
A second test was done near a field far away from Bradley's wifi system. The person with the receiver was stationary on one road, while the person with the transmitter drove away on a perpendicular road. The distance reached was 0.66 miles. After looking at the wireless transceiver configurations, the group realized that there was one more setting that could improve range. After turning on Forward Error Correction, the wireless range was 0.91 miles.

Microcontroller to Display Communication

Name	CAN ID	Minimum Value	Maximum Value	Label	Bits	Bit Offset	Units/bit
Engine Coolant Temp	525000	0	300	Fahrenheit	10	0	1
Engine Oil Temp	525000	0	300	Fahrenheit	10	10	1
Engine Oil Pressure	525000	0	100	PSI	10	20	1
RPM	525000	0	9000	RPM	10	30	10
Potentiometer 1	525001	0	12	Inch	10	0	0.1
Potentiometer 2	525001	0	12	Inch	10	10	0.1
Potentiometer 3	525001	0	12	Inch	10	20	0.1
Potentiometer 4	525001	0	12	Inch	10	30	0.1

Figure 5: CAN Communication

Since the AT90CAN128 microcontroller has a 10 bit ADC, that is the highest number of bits obtainable from the sensors. The 10-bit sensor data values collected by the AT90CAN128 are sent to the display using CAN. To accomplish this, the group found a package of CAN libraries and example programs from Atmel. After including the necessary files and headers, an additional config.h file was required to define values to reference the AT90CAN128 clock at 8MHz and to set a CAN baudrate of 250 kbps. The can_init function (included in the libraries) was run to reset and initialize assorted control registers, empty the mailbox of content, initialize the bit timing, and set a control bit in the CAN general control register to enable CAN. The can_init function required an input value, and the value was found to be a single control bit used to specify whether the program would automatically calculate the CAN baudrate or use a given fixed value.The control bit was set to 0 to initialize the program to operate with a fixed baudrate of 250 kbps.

Following initialization, a TransmitCAN function was written, using a function similar to one of the example CAN programs available in the Atmel reference libraries. The libraries set up a structure, st_cmd_t, which contains multiple variables relating to the CAN frame being sent. Rather than programming the sensor data directly into the CAN frame, the function instead sets up a separate 8 byte buffer, which is then referenced by a pointer in the st_cmd_t structure. Note that the example program being referenced set up the buffer as an array of eight uint8_t values, but for this program a single uint64_t variable was used to make it easier to insert the 10 bit sensor data rather than splitting it across multiple 8 bit variables. The address of this buffer was passed to st_cmd_t.data, and then the buffer was filled with the sensor info by inserting each sensor value, shifting the buffer left 10 bits, and repeating the process until the fourth value was inserted.

Once the data is inserted in the buffer, the details of the CAN frame are specified using the functions st_cmd_t.dlc, st_cmd_t.dlc, and st_cmd_t.id to initialize a message with 8 bytes of data, an extended 29 bit identification (ID) number, and to set the ID number (either 525000 or 525001) as an input to the function. Finally, st_cmd_t.cmd is set to CMD_TX_DATA (defined in can_lib.h), giving the order to transmit the data to the display.

After the TransmitCAN function was working, a larger TransmitAll function was written to receive one structure containing all of the sensor data (not st_cmd_t, but a separate structure to hold the eight uint16_t sensor values defined in the program). This function receives the sensor structure, then runs the TransmitCAN function twice, each time inserting the proper ID number and the four sensor values that correspond to the ID number.

Microcontroller to Wireless Communication

The RM024 wireless transceivers use 8 data bits, and is not re-configurable. Therefore, all signals will use a multiple of 8 for the number of bits. Since some of the signals need more than 8 bits to avoid loss of precision, all data was transmitted as 16 bit signals. Although the potentiometer and Engine Oil Pressure signals could use 8 bits without losing precision, the system is already faster than it needs to be(due to the slow nature of the mechanical inputs).Therefore it is easier to simply transmit all signals as 16 bit unsigned integers.

Once the sensor data is received, it is stored in a custom C structure containing eight uint16_t variables for all eight sensors. (The group also tested the system using dummy values in the sensors' ranges programmed into the microcontroller.) Before any data is sent to the transceiver, a separate function is run to initialize the UART registers on the AT90CAN128, enabling EIA-232 communication with a baud rate of 9600, 8 bit messages, no parity, and one stop bit, as well as enabling the TxD bit in Port E for transmission.

To send the data, a TransmitEIA function (referenced from an example in the Atmel datasheet) checks if the transmit buffer on the AT90CAN128 is empty, and once the buffer is clear, the given data byte is loaded into the UDR0 buffer where it is automatically transmitted through the TxD pin. After this function was working, a larger TransmitAll function (the same function from the previous section) was written to receive the one structure containing all the sensor data, then break each 16 bit value into two bytes, sending each byte through TransmitEIA in the proper order to be received by the RM024, ending the process by sending a newline.

A bug was found where LabVIEW would repeatedly stop and show an error saying "Memory or data structure corrupt", but this was traced to the newline character required by LabVIEW. The newline character, represented by 0x0A, is equal to 10 in decimal format, causing LabVIEW to read any sensor input of 10 as a newline character. This was fixed by converting any sensor values of 10 to 11 prior to transmission since the ranges for all of the sensor data would render such a minimal change negligible. Of course, other options are available if the data change from 10 to 11 would be unacceptable in other situations.

Wireless to LabVIEW Communication

LabVIEW parses the data received from the RM024 using the Instrument I/O Assistant block, shown below.

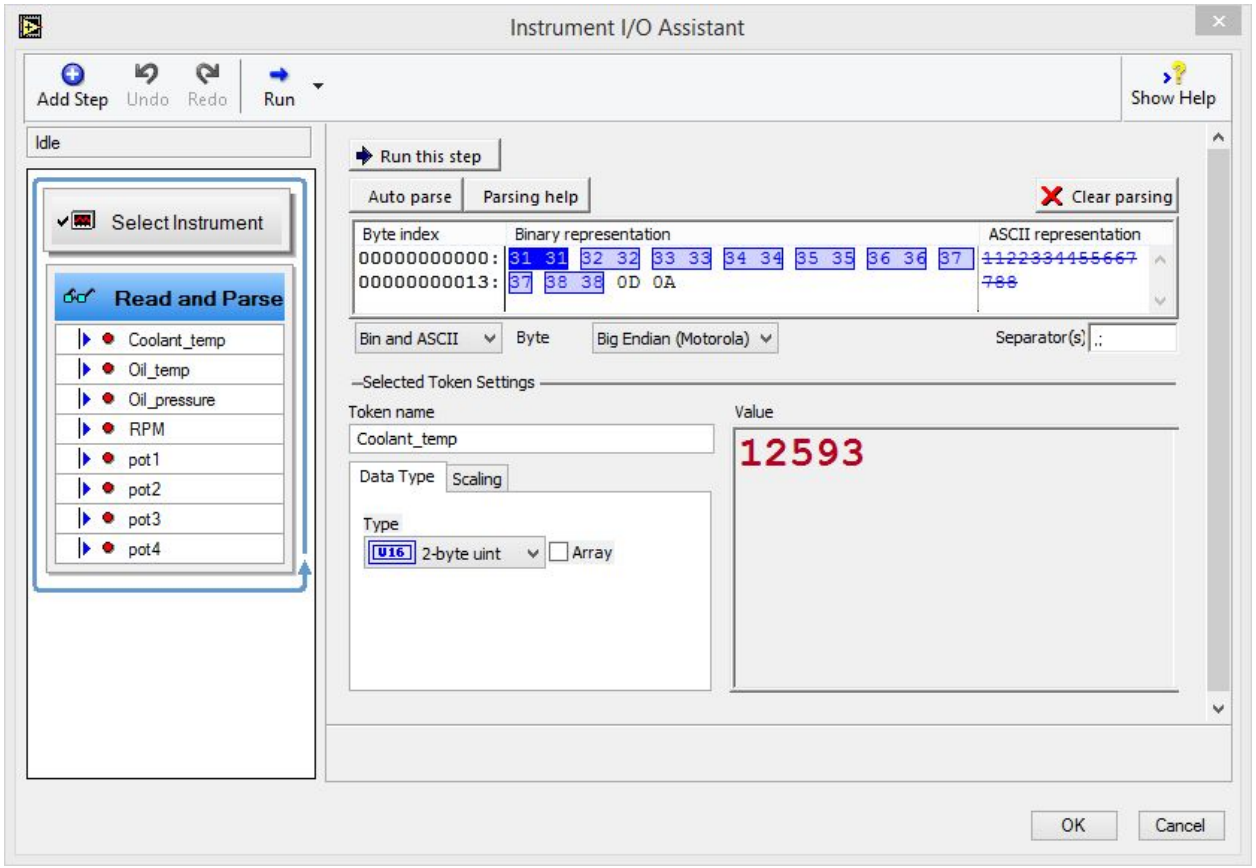

Figure 6: I/O Assistant ‘Read and Parse’ step

The figure above shows how the I/O Assistant parses the received data. Each input is two bytes, and has a Token name to identify it, and has a data type. To configure the Instrument I/O assistant block, any number of bytes are highlighted to create a variable. The variable can then be given a datatype and name, and the Instrument I/O Assistant creates that variable to be used in code.

*R*M024 Wireless Transceivers

The wireless transceivers transmit the data in “packets”. Packets are sent once they either reach their maximum size of , 239 bytes, or the 2.5ms interface timeout occurs. Because the transceiver automatically inserts header information into every packet sent, it is more efficient to send large packets instead of small packets.

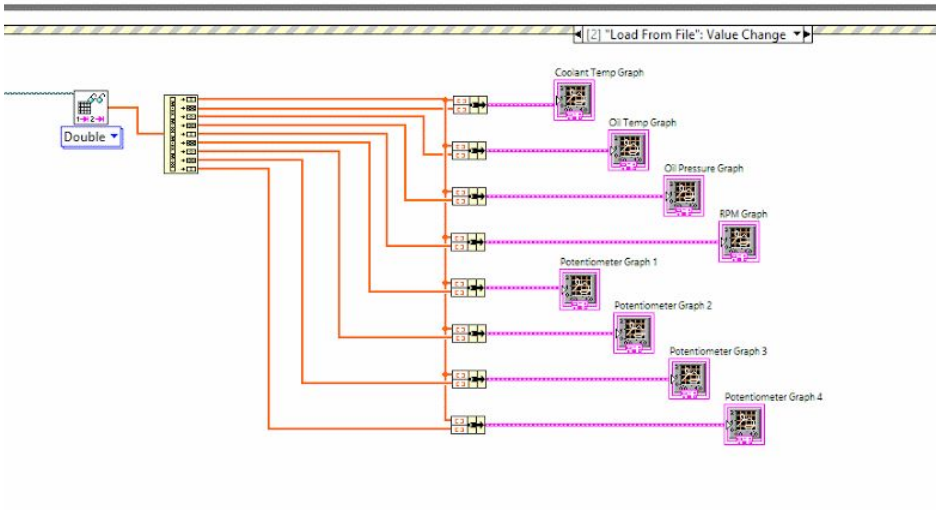

Figure 7: ‘Load From File’ button

This code runs when the ‘Load From File’ button on the ‘Graphs’ tab is pressed. It updates all of the graphs with data from the most recently saved file. The code works by opening and reading the file, separating the data into 8 data arrays, and then sending the data to the 8 graphs. For more information about using LabVIEW, see the LabVIEW Tutorial in Appendix A.

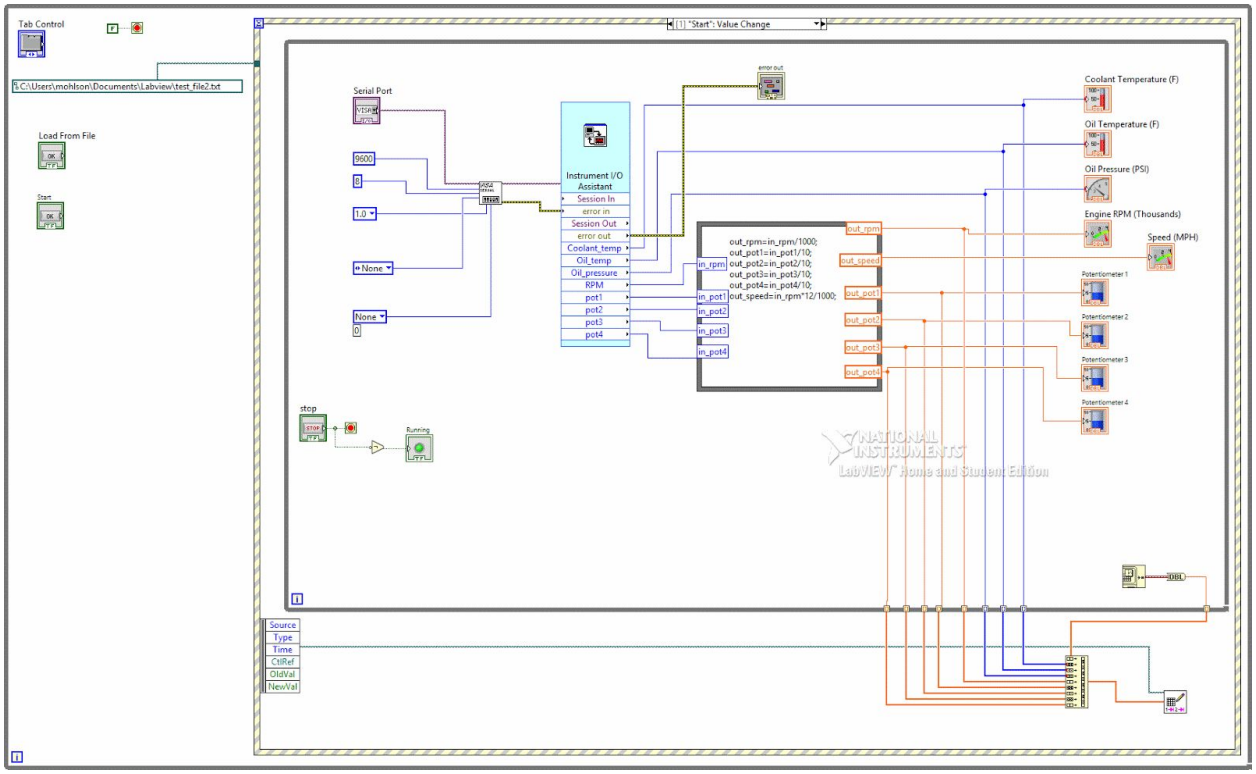

Figure 8: ‘Live Mode’ code

The figure above shows code that is run when the ‘Connect’ button is pressed on the Live Mode

page. Note that this code assumes that a Serial Port has been selected. The functional blocks shown in Figure 8 will receive data from the wireless transceiver, process that data, display it on the gauges and indicators in the LabVIEW GUI, and save the data for later viewing.

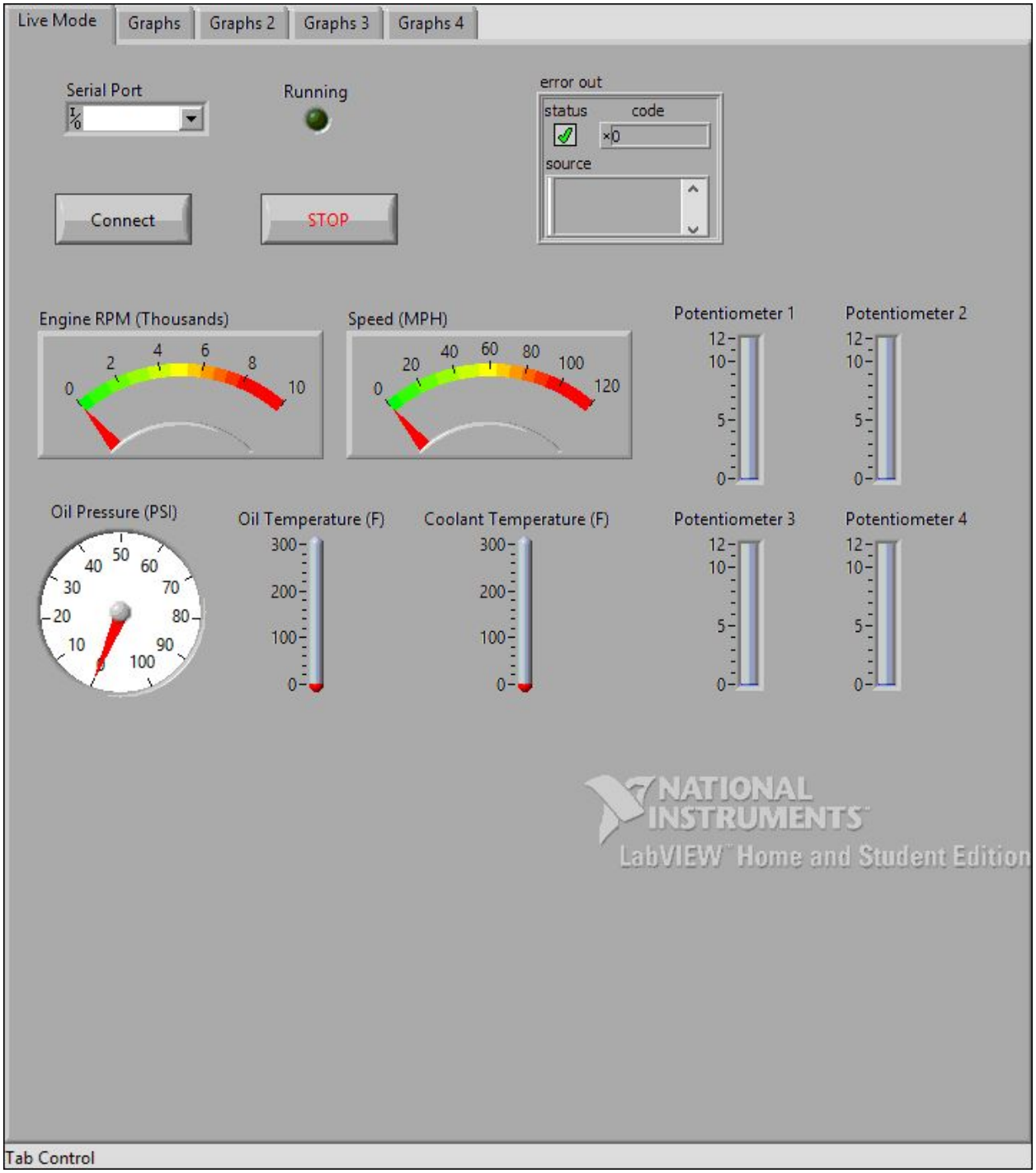

Figure 9: Laptop GUI 'Live Mode' page

This is the page the user sees while using Live Mode on the laptop computer. The user chooses a serial port, and then presses the connect button. While it is connected, the green 'Running' indicator will be on, and the graphs and indicators will display the received data. When done

collecting data, the user presses the ‘Stop’ button.

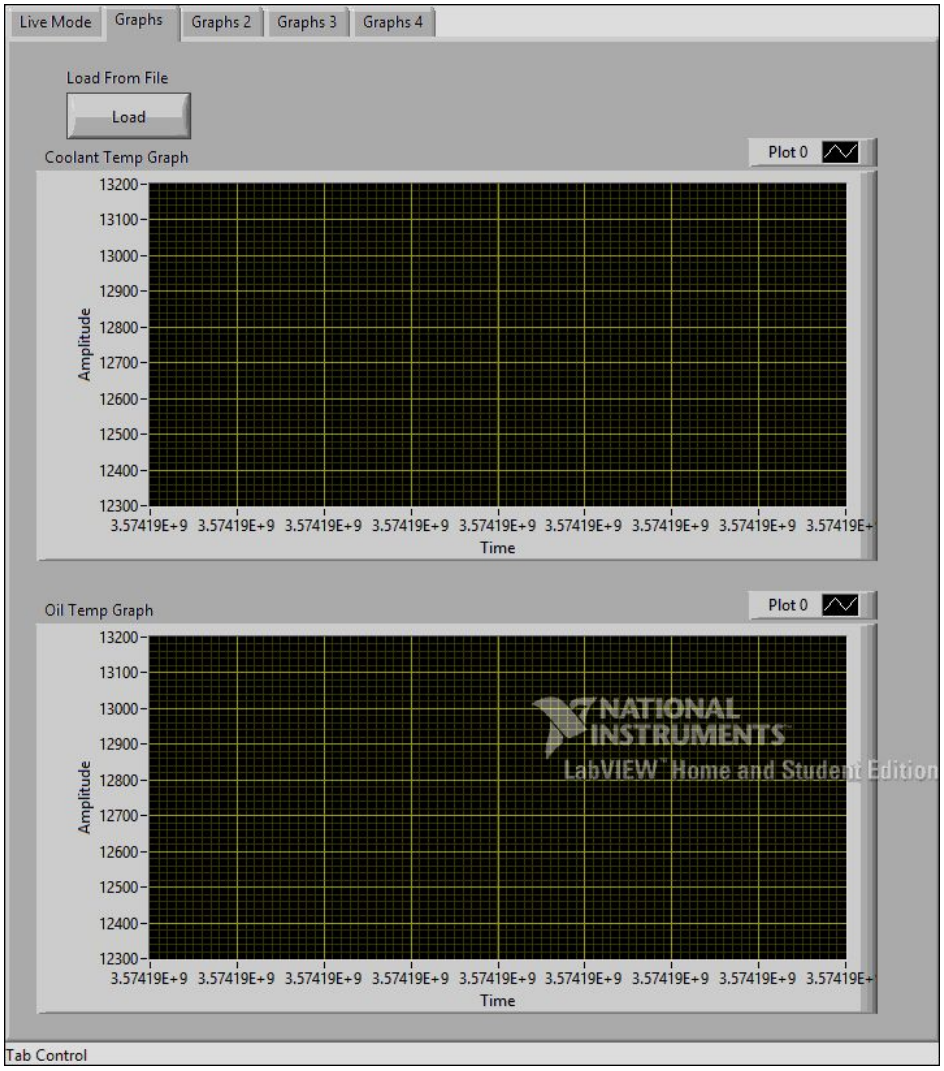

Figure 10: Graphs Page

The ‘Graphs’ page has the ‘Load From File’ button to read the saved data and display it on all of the graphs on the four graph pages. The ‘Graphs’ page is the only one of the Graph pages with the button.

A Parker IQAN MD4-7 display, an XA2 expansion module, and all the cables necessary to connect peripherals to the devices was donated to the SAE Formula Car Display and Data Acquisition System team by the Parker-Hannifin Corporation. However we decided to not use the XA2 expansion module, but instead used an Atmel AT90CAN to receive the sensor data, send CAN messages, and send EIA-232 information to the transceivers.

Parker IQAN MD4-7

Figure 11: IQAN Touchscreen Display

System Overview

The master module, IQAN-MD4, is the central unit in the system, or in the case of a multi-master system, one of the central units. The IQAN-MD4 has four CAN buses and two ethernet ports. The CAN buses support ICP and are able to control IQAN expansion units. SAE J1939 and Generic CAN protocols are also supported on the CAN buses and provide the possibility to interface to 3rd party units. A touch screen in combination with a graphical display makes system feedback with user interaction possible. The display in the module has very high optical performance across a wide operating temperature range and over a wide range of ambient light. The IQAN-MD4 has voltage and digital inputs that are designed to be flexibly configured using IQANdesign software. The unit also has four low side digital outputs. All I/O ports are EMI filtered and protected against short circuit to -BAT and +BAT.[1]

[1]

System Design Process

When starting this project, the design team obtained the IQAN MD4-7 display through a donation from the Parker-Hannifin Corporation. The IQAN MD4-7 display is restricted to 2 voltage inputs, 8 digital I/O pins, 2 ethernet ports, and 4 CAN buses. However, the RM024 transceiver used by the team is only able to communicate over EIA-232 serial communication lines. Therefore, the team needed to add another component that would be able to communicate with the display and the transceiver. Since all members of the team were familiar with Atmel processors and IDE, the team selected an Atmel product with both EIA-232 and CAN communication. The AT90CAN microcontroller was selected because it is able to support the 8 sensors, display, and transceiver as required for this project.

Messaging Spreadsheet

When trying to communicate with the microcontroller, it must be clearly understood what each component is sending and what the other component is expecting to receive. The microcontroller the ADC is 10 bits, and 10 bits of information will be sent for each sensor data point. In CAN communication, information is sent by frames as seen in the figure below. Each frame has a specific identifier and can only contain 8 bytes of information. This system will have 8 sensors with 10 bits per sensor. Since 80 bits of data will not fit into one CAN frame, the team must use 2 CAN frames to permit transmitting all 80 bits of data. We split the two CAN frames up by location on the formula car. Four sensors will be transmitting engine data and the other four sensors will be transmitting suspension data.

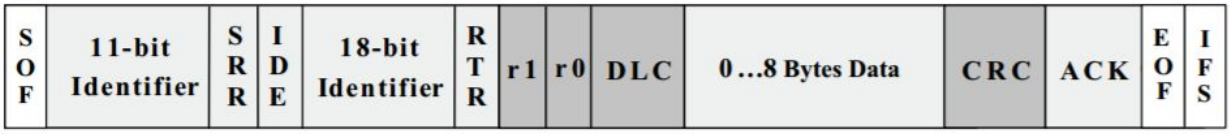

Figure 12: Example: CAN Frame

https://www.parker.com/literature/Electronic%20Controls%20Division/Literature%20files/IQAN-MD4_instructionbook_HY33-8408-IB_UK.pdf

Name	CAN ID	Minimum Value	Maximum Value	Label	Bits	Bit Offset	Units/bit
Engine Coolant Temp	525000	0	300	Fahrenheit	10	0	1
Engine Oil Temp	525000	0	300	Fahrenheit	10	10	1
Engine Oil Pressure	525000	0	100	PSI	10	20	1
RPM	525000	0	9000	RPM	10	30	10
Potentiometer 1	525001	0	12	Inch	10	0	0.1
Potentiometer 2	525001	0	12	Inch	10	10	0.1
Potentiometer 3	525001	0	12	Inch	10	20	0.1
Potentiometer 4	525001	0	12	Inch	10	30	0.1

Figure 13: CAN Messaging

IQAN Design

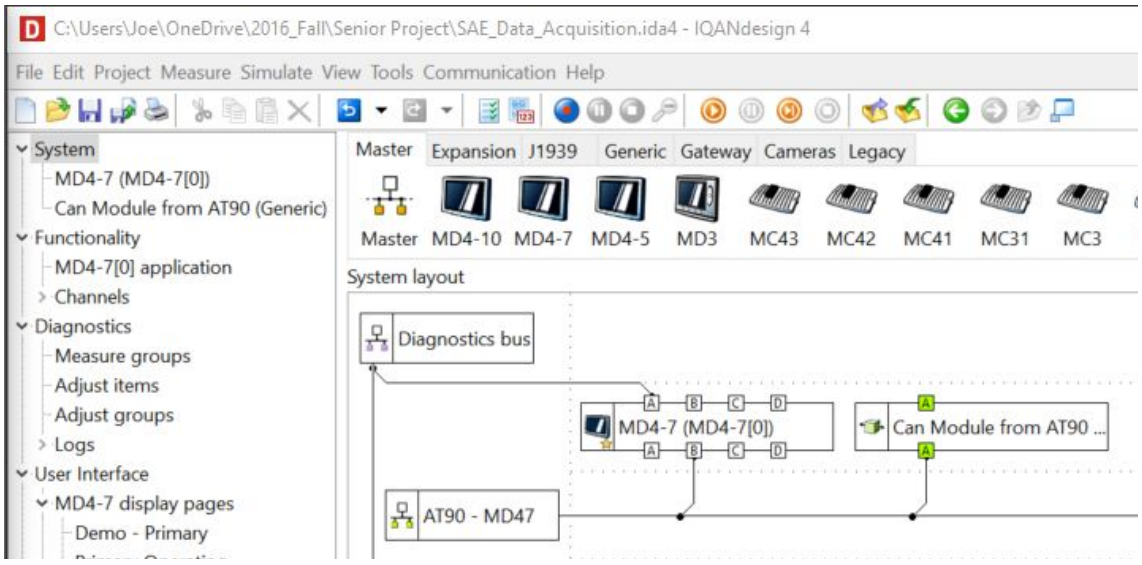

Figure 14: IQAN Design Top Level

When creating an application in IQAN Design, the user must first start at the system level as seen above. The user can drag different modules and their communication schemes, and then show how they are connected. Figure XX above shows how the MD4-7 is connected to the AT90 - MD47 CAN bus. Attached to the CAN bus is the CAN module from the AT90. Many different modules can be added to a system, making this development tool very versatile.

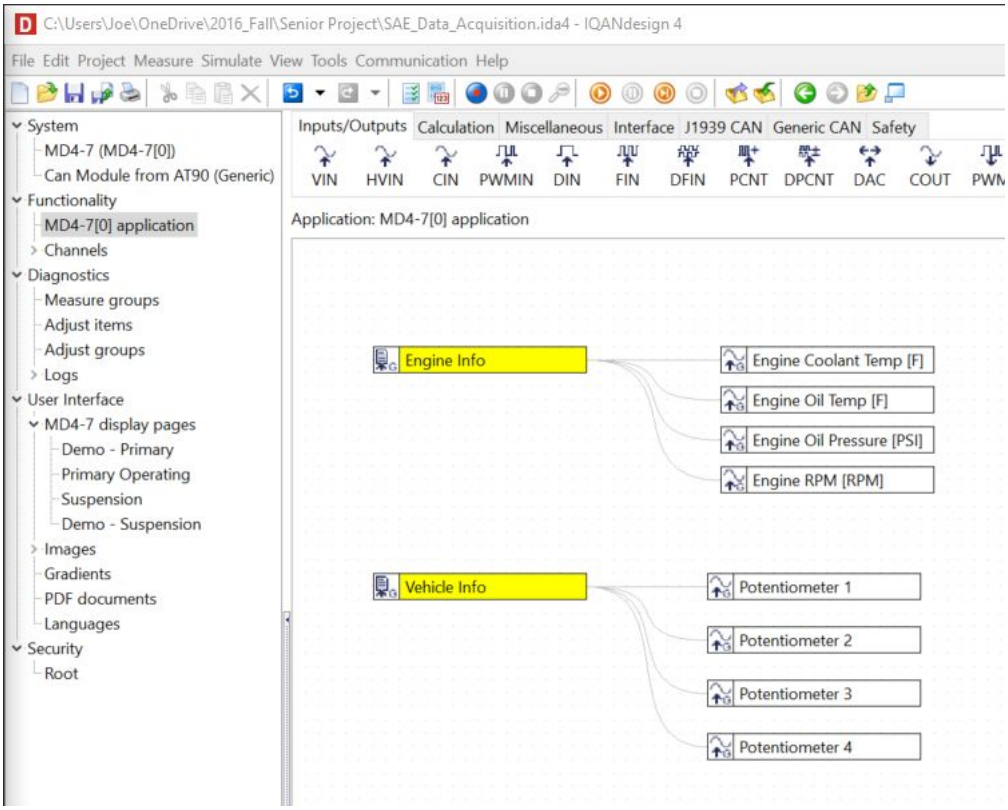

Figure 15: IQAN Design CAN Frame

After the high level design is completed, the user must define what signals (external and internal) will be used in the project. For this project, the team had to use two CAN frames. As can be see in Figure 14 above, there are two CAN frames shown that are then broken into the 8 sensor data bits.

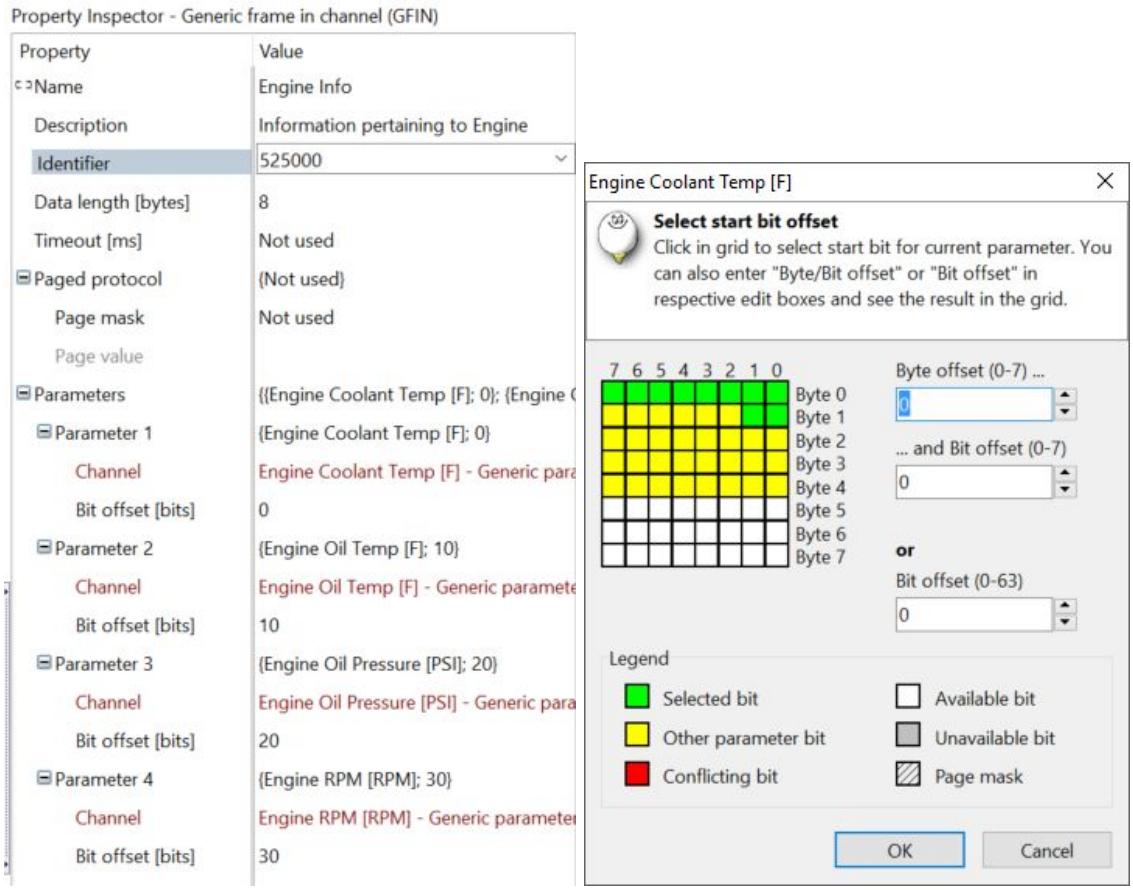

Figure 16: CAN Frame Creation

To define the CAN frame once it is added to the project, the user must show what bits of the Frame are for what messages. IQAN Design has an interactive tool to allow the user to do so on the right side of the screen. The IQAN Design software also enables the user to tell how to interpret the values. In this case, all of the values were unsigned 10 bit numbers.

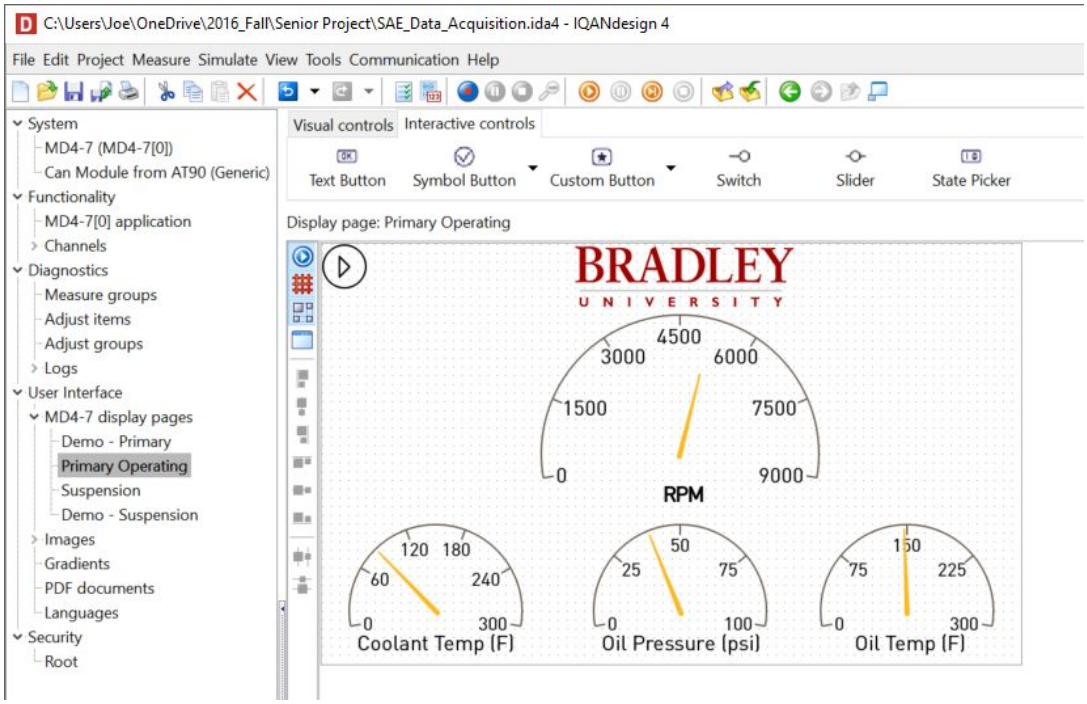

Figure 17: IQAN Design Display Page

Once useful data is defined in the project, data must be displayed in some fashion. To do this on the display, the user must create display pages. IQAN Design has many built-in display gauges so it is possible to simply drag and drop, and then resize the gauges. The user must also add the labels and connect them to the correct signals. Interactive controls are available on the display to go between 2 separate pages.

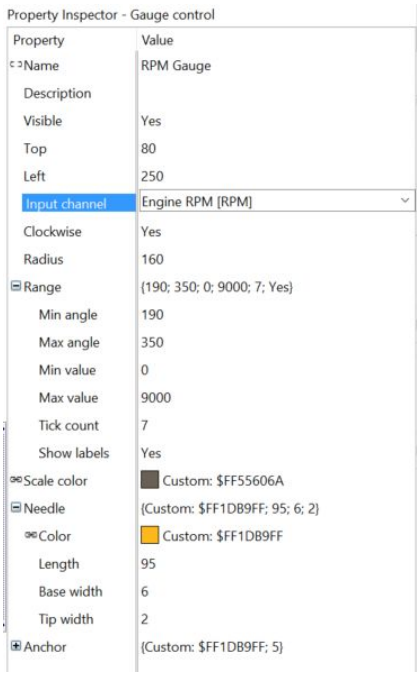

Figure 18: Gauge Input Signal

To link the gauge to the signal, simply set the input to the correct channel as shown in Figure 17 above.

Display Pages & Control

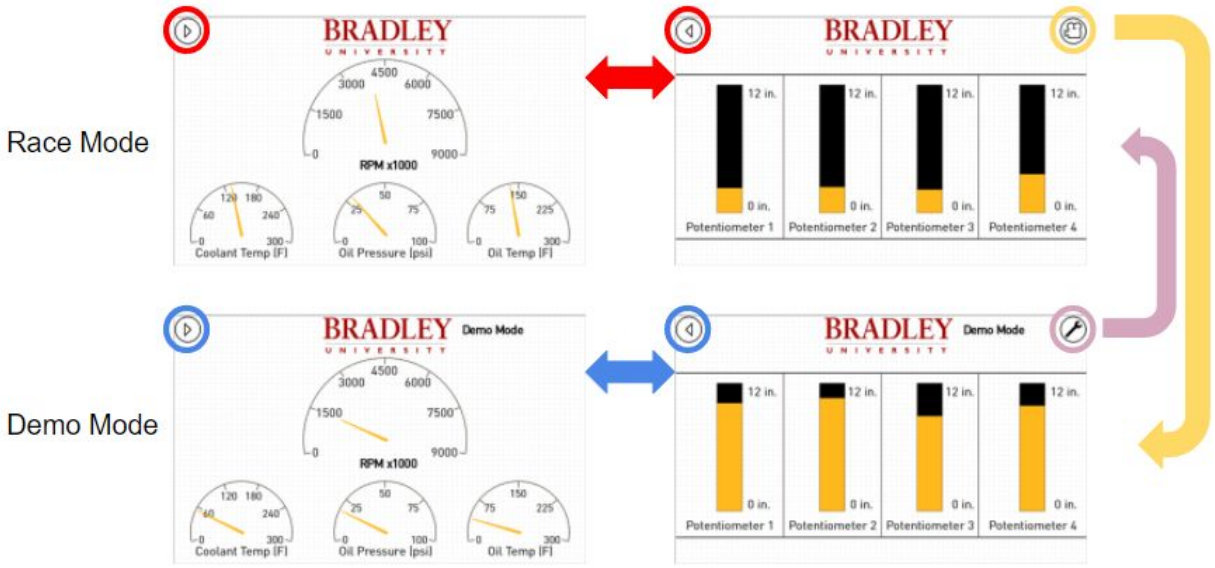

Figure 19: Display Pages

As can see from Figure 18 above, the systems is designed with a race mode and a demo mode. The race mode uses the CAN information from the microcontroller to fill in the gauges' information. However, the demo mode uses fake data generated from the display itself for judging during the SAE design competition.
The circled portions of the screens show how the user gets from one screen to another.

Demo Mode

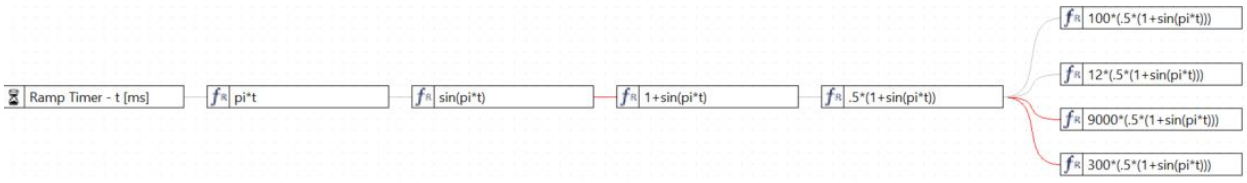

Figure 20: Demo Mode Signals

Similarly to how Simulink uses blocks, IQAN Design uses blocks to create its internal signals. As can be seen from Figure 19 above, the design sequence starts with a ramp timer of 1 ms. It is then fed into a multiplier of pi which is then input into a sin wave which must range from 0 to 1. And from the sine wave values ranging from 0 to 1 must make the values vary from the minimum gauge value to the maximum gauge value. Oil pressure ranges from 0 to 100 psi, suspension ranges from 0 to 12 inches, RPM ranges from 0 to 9000, and coolant and oil temperature range from 0 to 300 F.

IQAN Simulate

Finally, before downloading the program to the MD4-7, the user can simulate CAN messages and the display itself on a personal computer. To do so, open IQANsimulate and open the same file created in IQANdesign, and a simulated display screen will appear on the computer screen. The user can also send any desired CAN message to the simulation. This helps the user figure out what needs to be changed on the display before even initiating CAN communication with the display.

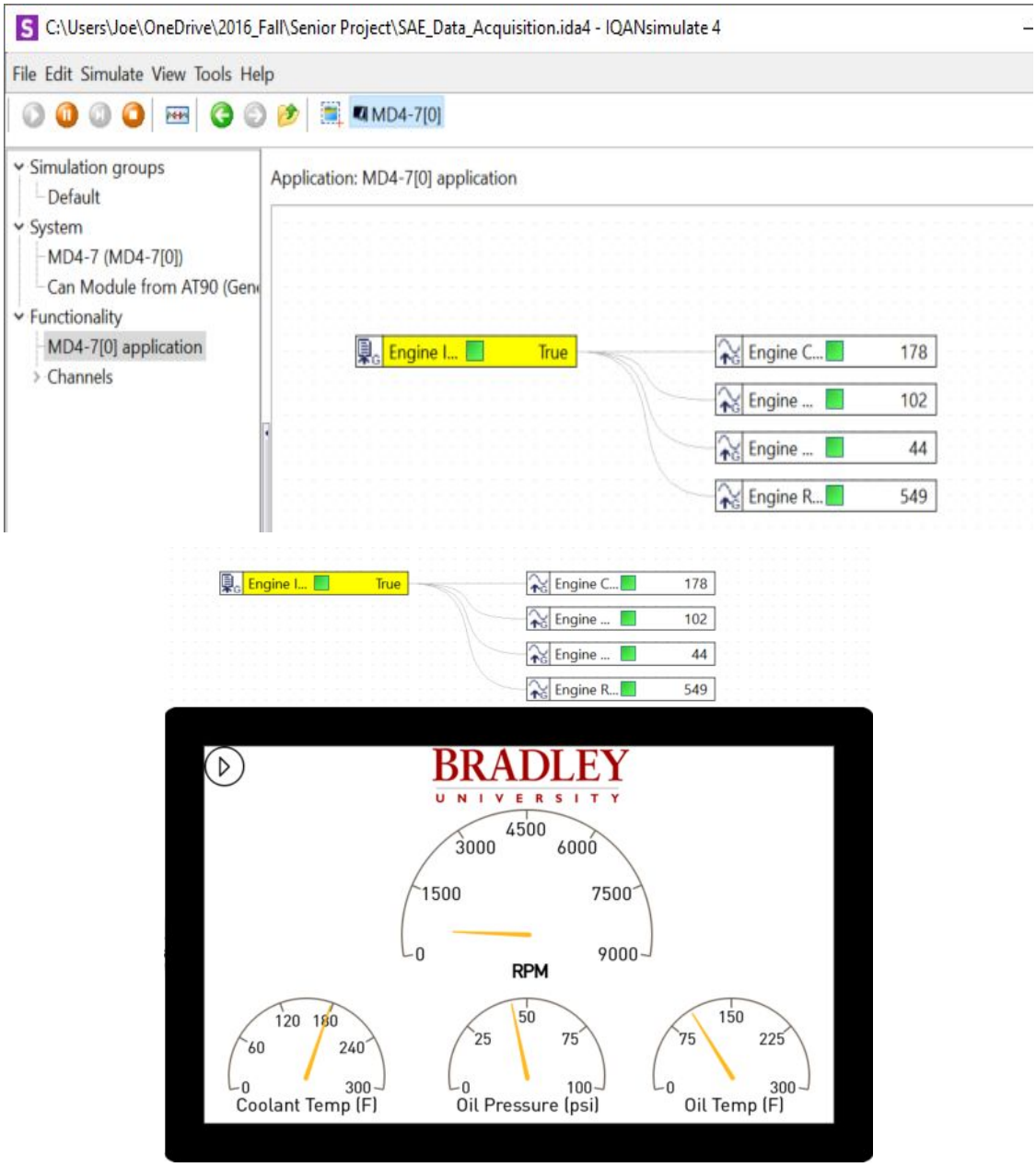

Figure 21: IQAN Simulate

Downloading Project to Display

One thing that makes IQANdesign a versatile tool is the variety of ways the user can download the project to the display. If this was on a large machine with one central CAN bus, simply tap into the CAN bus on the computer and download the project over CAN. The project can also be downloaded over ethernet, USB, and even over the internet if it has a connection. Once the user has selected the transfer protocol, simply press “Send Project” to send the program to the

display.

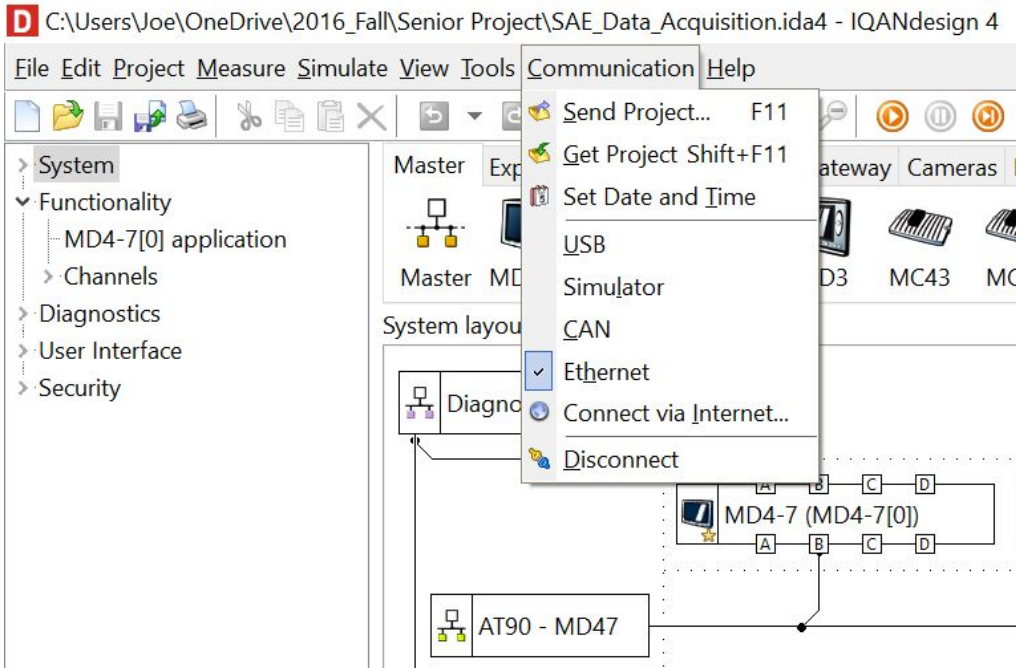

Figure 22: Downloading a Project

Discussion and Future Directions

Ideally, at the end of the project's completion, the team would like to implement this system on the mechanical engineering department's SAE formula car. Unfortunately, shortly after this project was started the Bradley University Mechanical Engineering Formula SAE team disassembled the vehicle that was originally specified as the target for the finalized system. Therefore, the completed data acquisition system could not be tested on a Formula SAE car

In terms of overall systems, the number of sensors could be increased significantly to include g-force sensors, seat-belt sensors, etc. The ideas are almost endless based on available sensors implemented on other automotive applications.

For the future, it is possible to envision adding a communication link with the engine and even adjusting engine parameters from the display as is common practice in many vehicles. For example, if the driver wished to change the idling RPM of the engine from the driver's seat, the driver could navigate through the touch screen display to an appropriate screen where a value could be entered and the engine would respond accordingly. This would require the engine to also communicate over CAN which is entirely possible. However, this would require a great deal of coordination between the mechanical engineering department and the electrical engineering department. Also, if that communication between systems was set up, the diagnostics for the

engine could also be sent to the display. As an example, if a cylinder was not firing properly a message could be displayed informing the driver and crew.

Finally, although this system was designed with an automotive application as the target, the basic system components could be used for many other applications, including remotely operated robots.

Appendix A - LabVIEW Tutorial

Create a Project

The first step to using LabVIEW is to press the Create Project button.

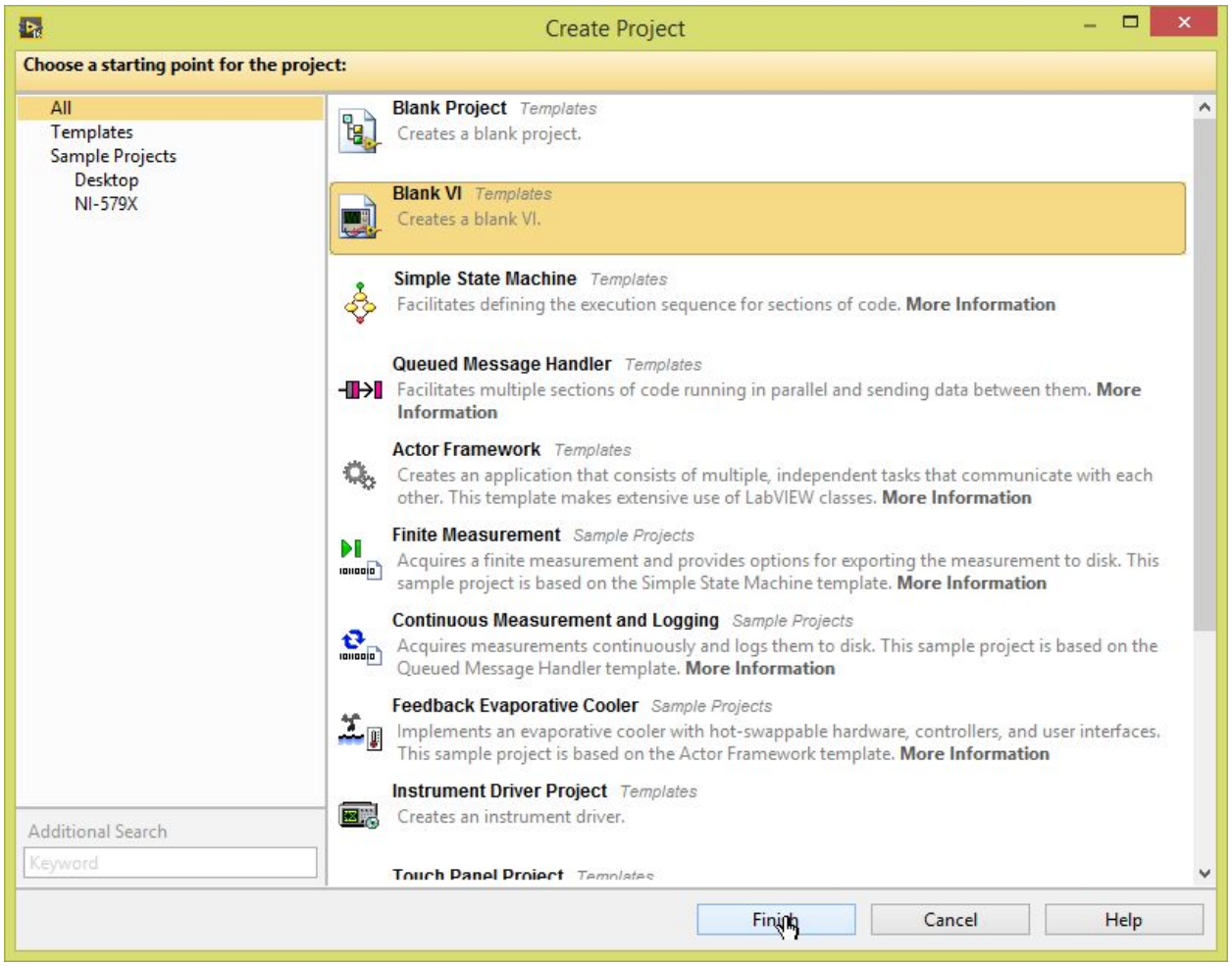

Figure 23: Create Project Window

After pressing the ‘Create Project’ button, this window will pop up. Choose the ‘Blank VI’ template to create a GUI project.

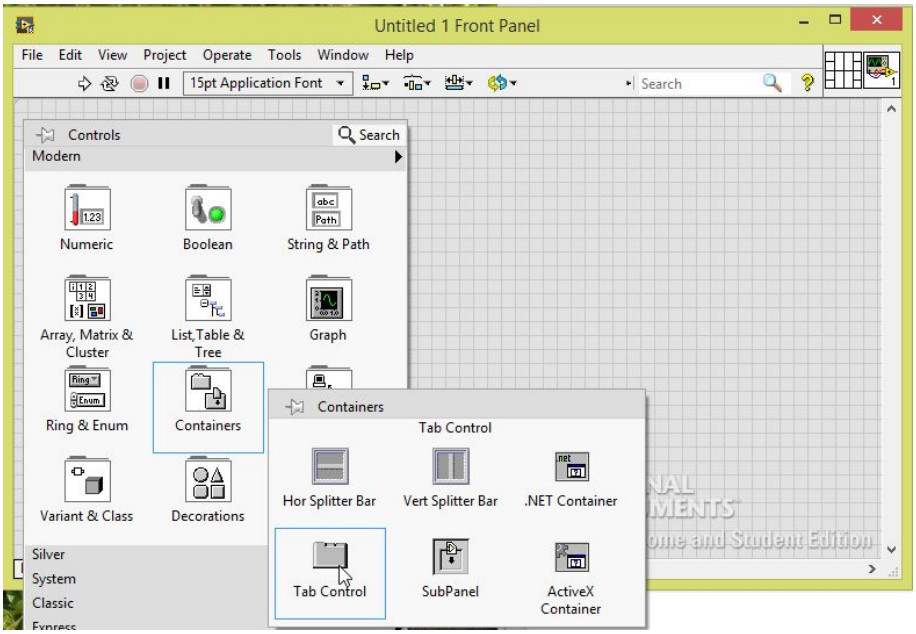

Figure 24: Insert Tab Control

The front panel starts out blank. First, add a container, such as the ‘Tab Control’, by right clicking anywhere in the grid and selecting a container. This method can be used to also add anything else to the GUI, such as a graph, display, textbox, or button.

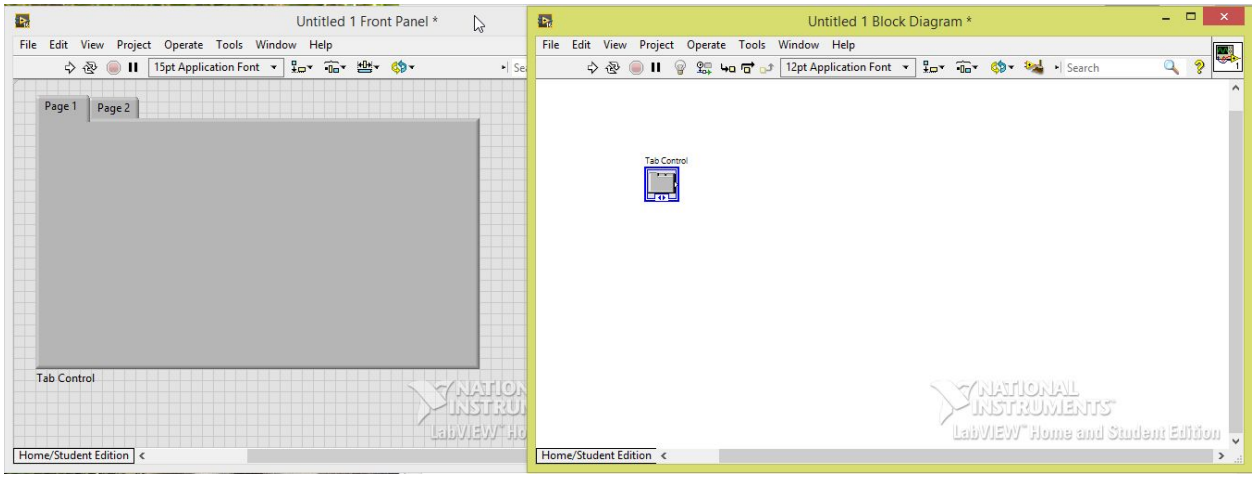

Figure 25: Tab Control

The tab control will appear in the Front Panel, and a tab control block will appear in the Block Diagram.

The Front Panel shows what the GUI looks like to the user, and the Block Diagram is used to program it.

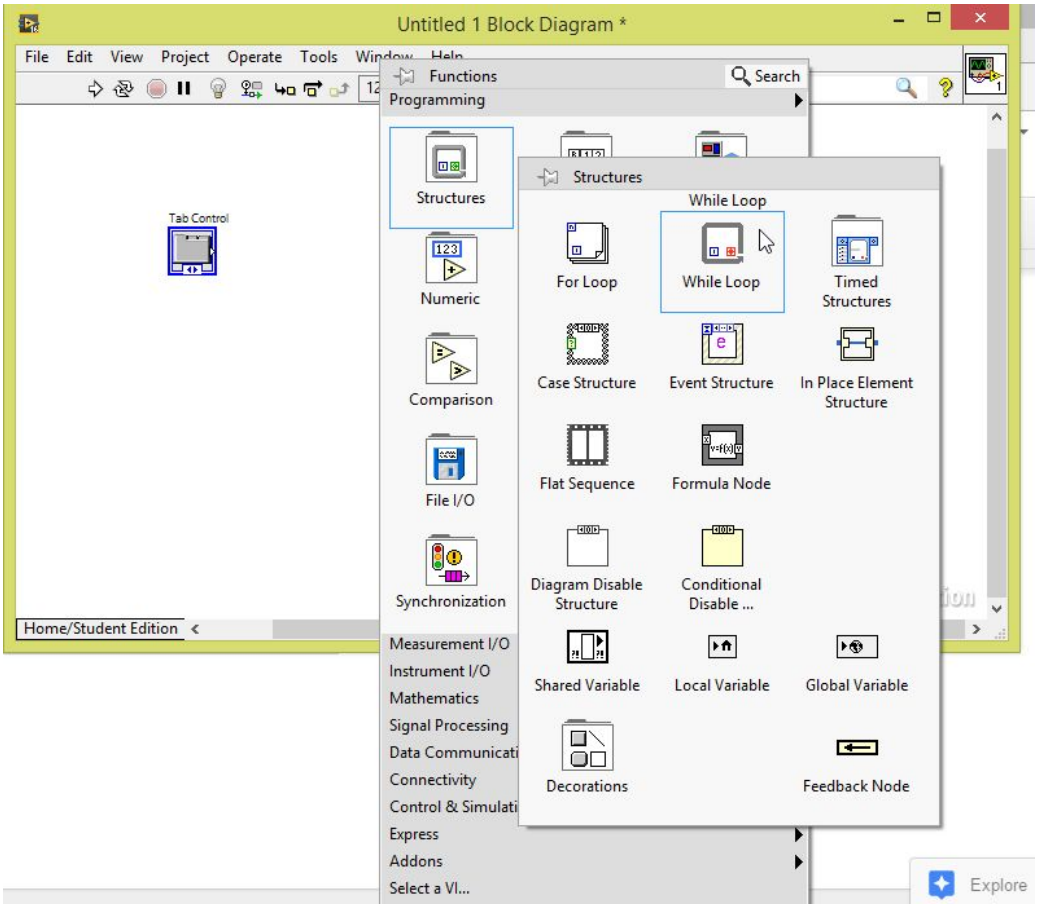

Figure 26: Insert While Loop

Blocks can also be added by right-clicking in the Block Diagram page.

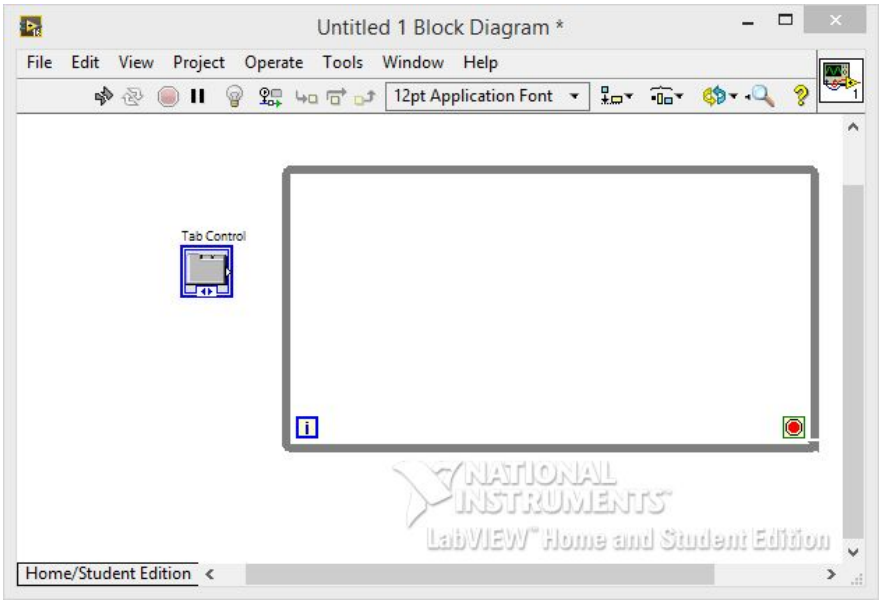

Figure 27: While Loop

After adding something in the Block Diagram page, it will appear there.

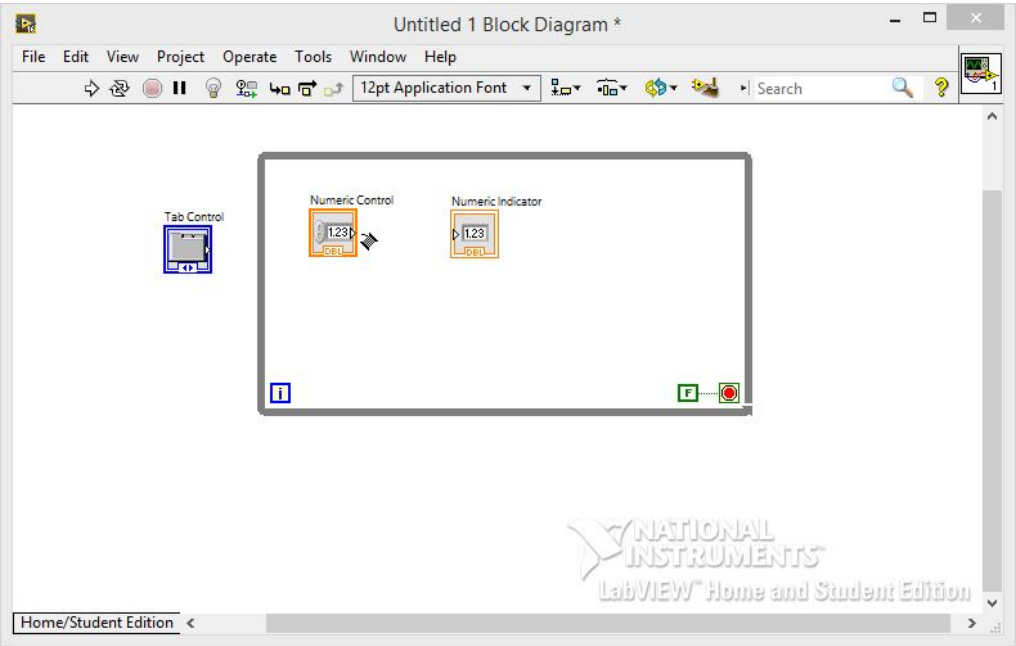

Figure 28: Connect Blocks

To connect blocks in the Block Diagram, hover over the port until the cursor changes to the one shown in the figure above. Then, click and drag to connect it to another port.

Run LabVIEW Code in LabVIEW

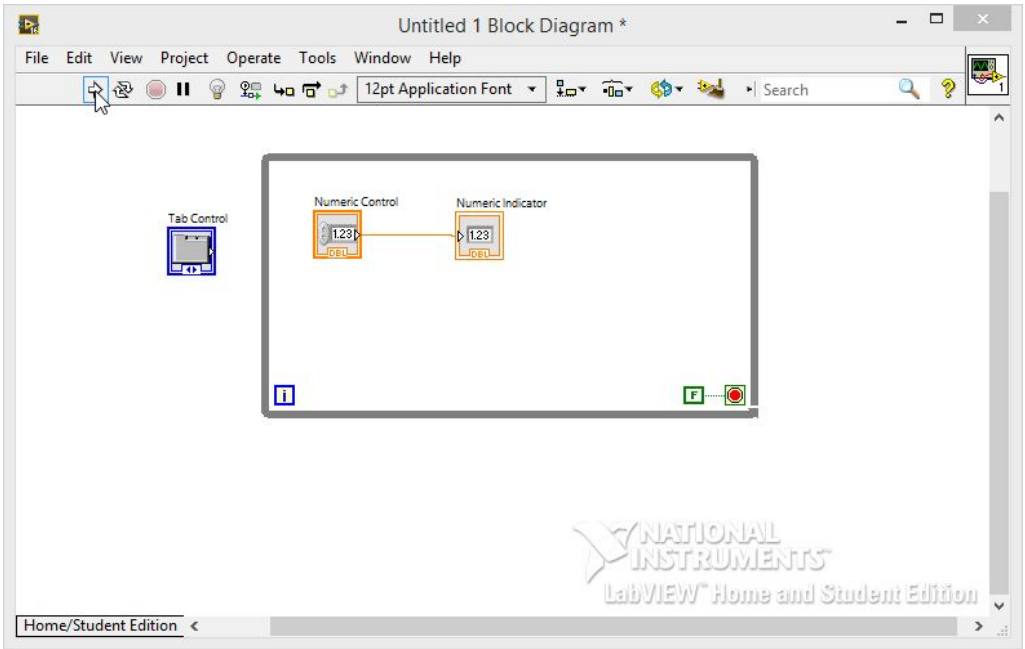

Figure 29: Run Program Button

To run the program, press the white ‘run’ arrow in the upper left corner of the screen, shown in

the figure above.

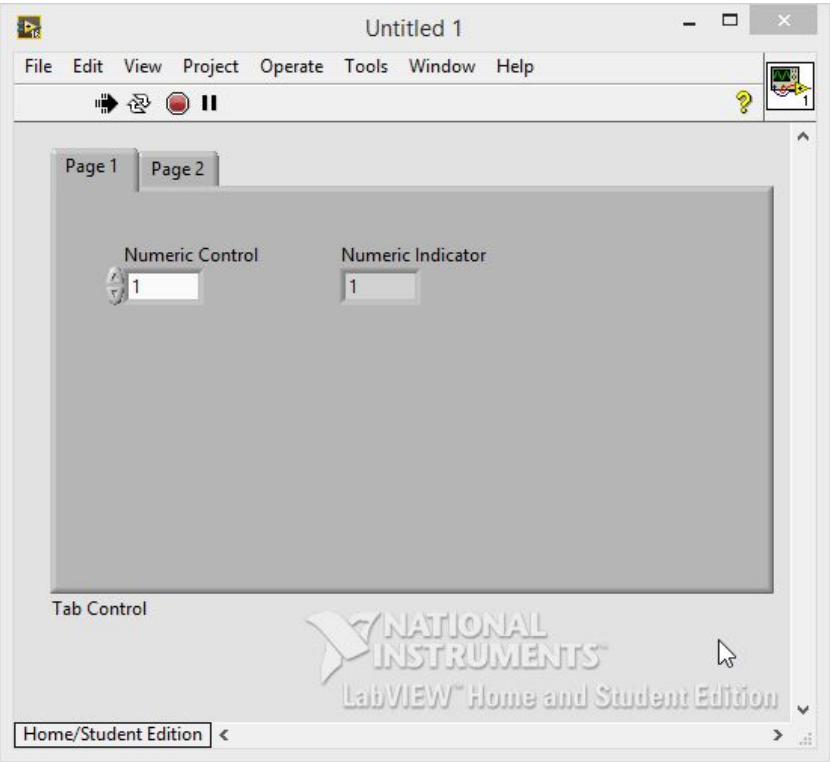

Figure 30: Running LabVIEW Front Panel

This figure shows the running GUI. This particular program is a simple program that displays the Numeric Control input in the Numeric Indicator box. Notice that the white ‘run’ arrow turned into a black ‘Running’ arrow. Press the red ‘Stop’ button to stop the program.

Compile LabVIEW Code

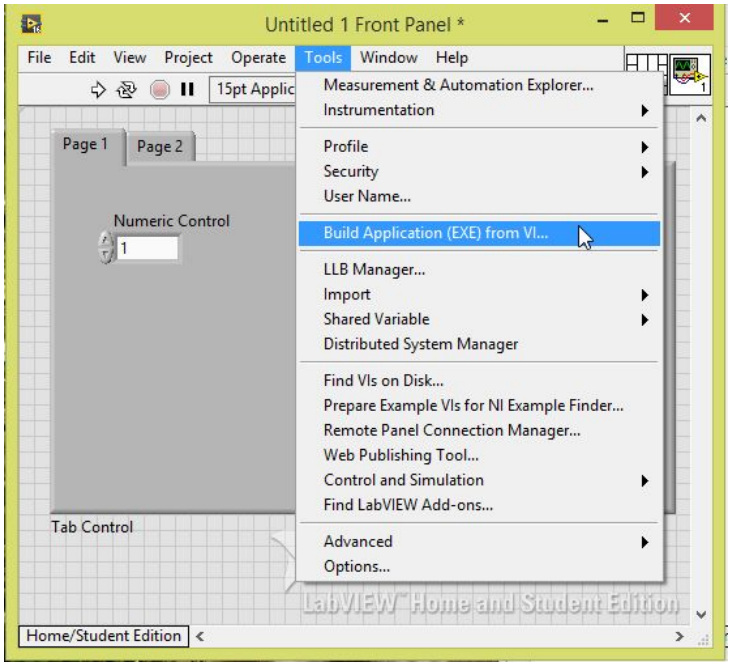

Figure 31: Compile LabVIEW Project

To compile your GUI into an executable file, press Tools->Build Application (EXE) from VI. This will create a Build Specification.

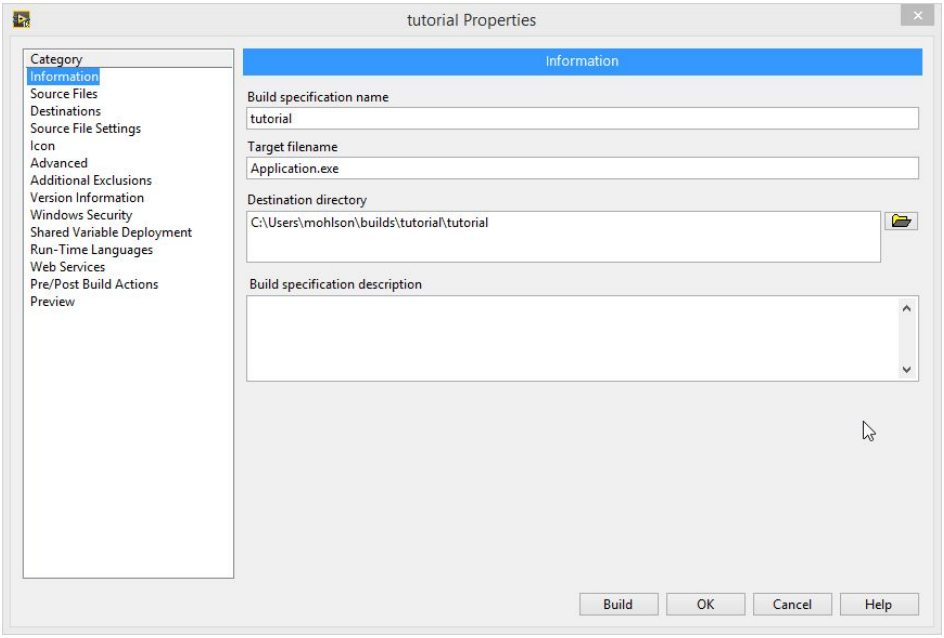

Figure 32: Build Specification Properties Page

Change directories or names as desired, then press the ‘Build’ button at the bottom to compile your LabVIEW project. These options can be changed later. It is also possible to make more

than one Build Specification.

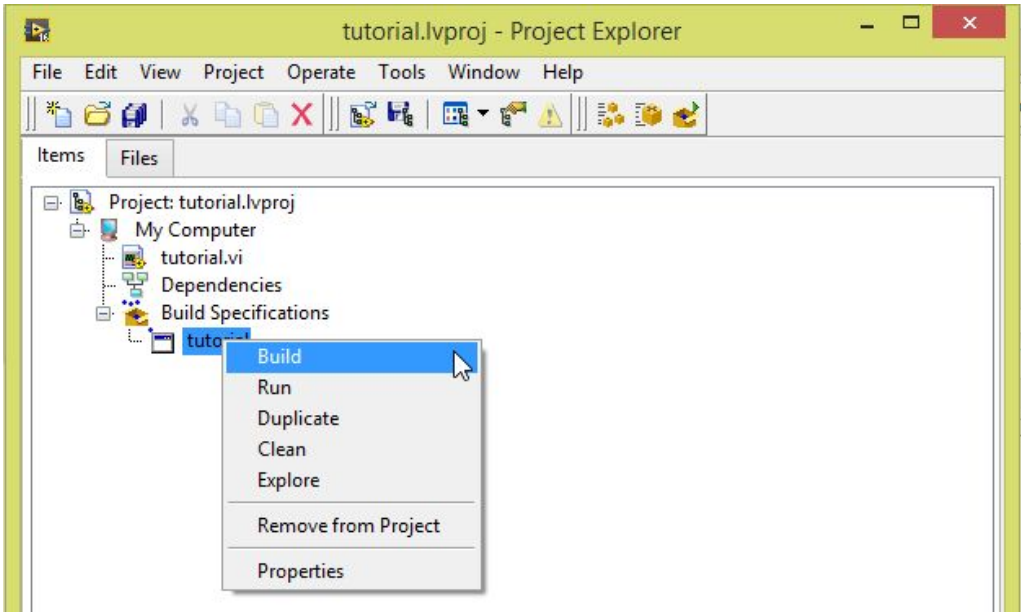

Figure 33: Build Specification

To build it again with the same options set previously, go the the Project Explorer window, and right click on the Build Specification created earlier. Press the ‘Build’ button to build, ‘Run’ button to run, and ‘Properties’ button to edit the build specification and change the options specified previously.

Graph Zoom

It is possible to zoom in on any part of the graph. The figures below show how to zoom in using the LabVIEW Graphs.

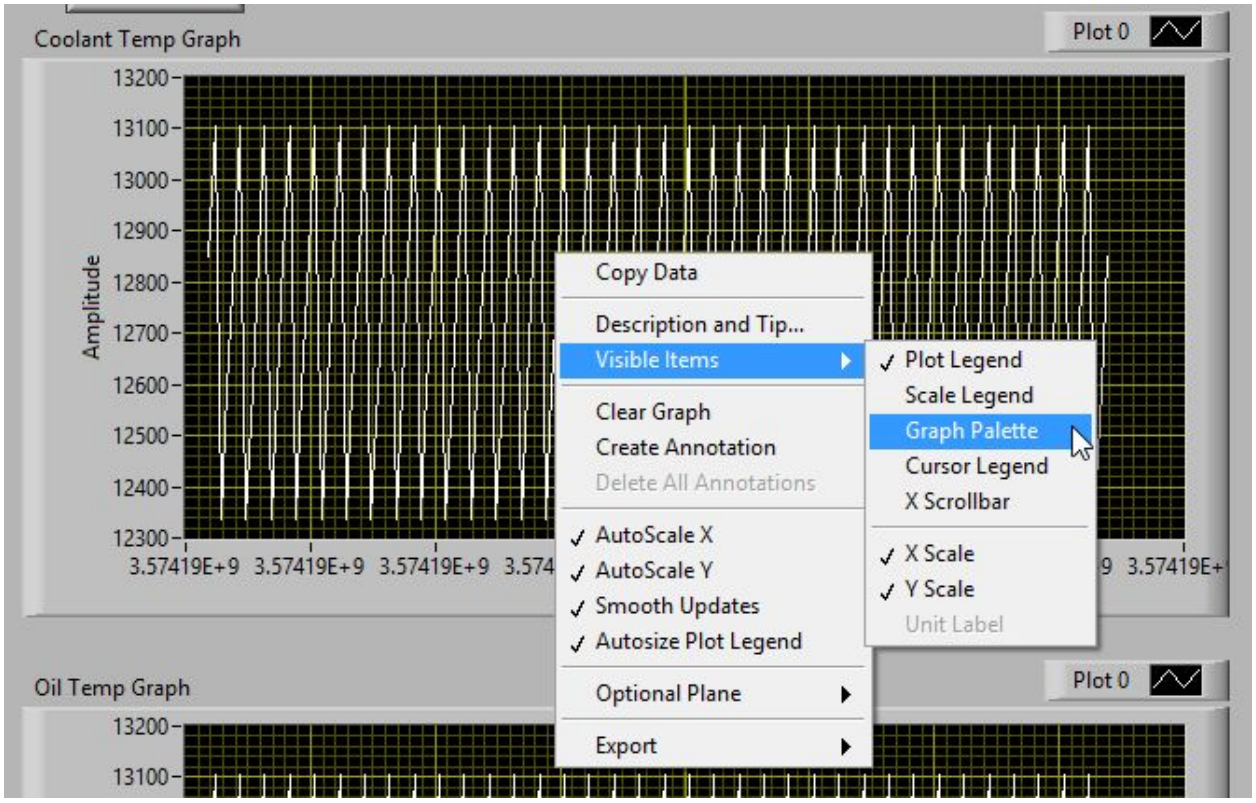

Figure 34: Graph Zoom

The first step is to right-click on the graph, and press Visible Items, and Graph Palette.

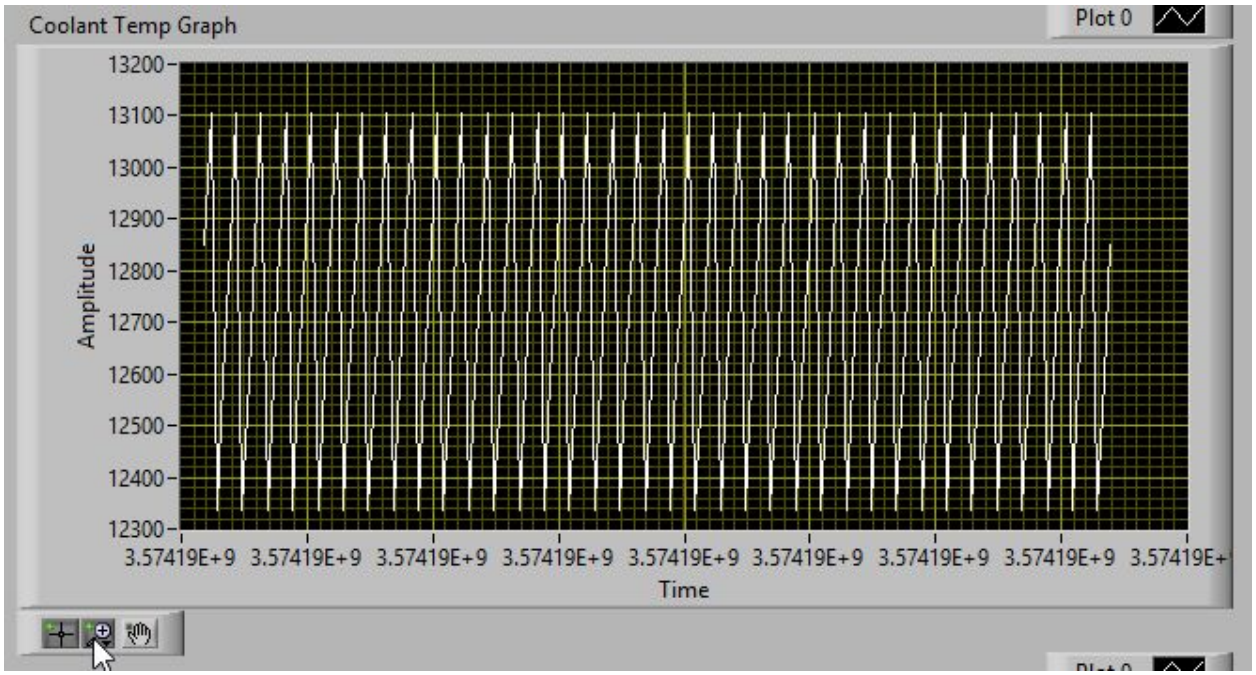

Figure 35 Graph Zoom

After pressing Graph Palette, three buttons will appear on the bottom left of the graph. Press the

middle button to use the zoom buttons.

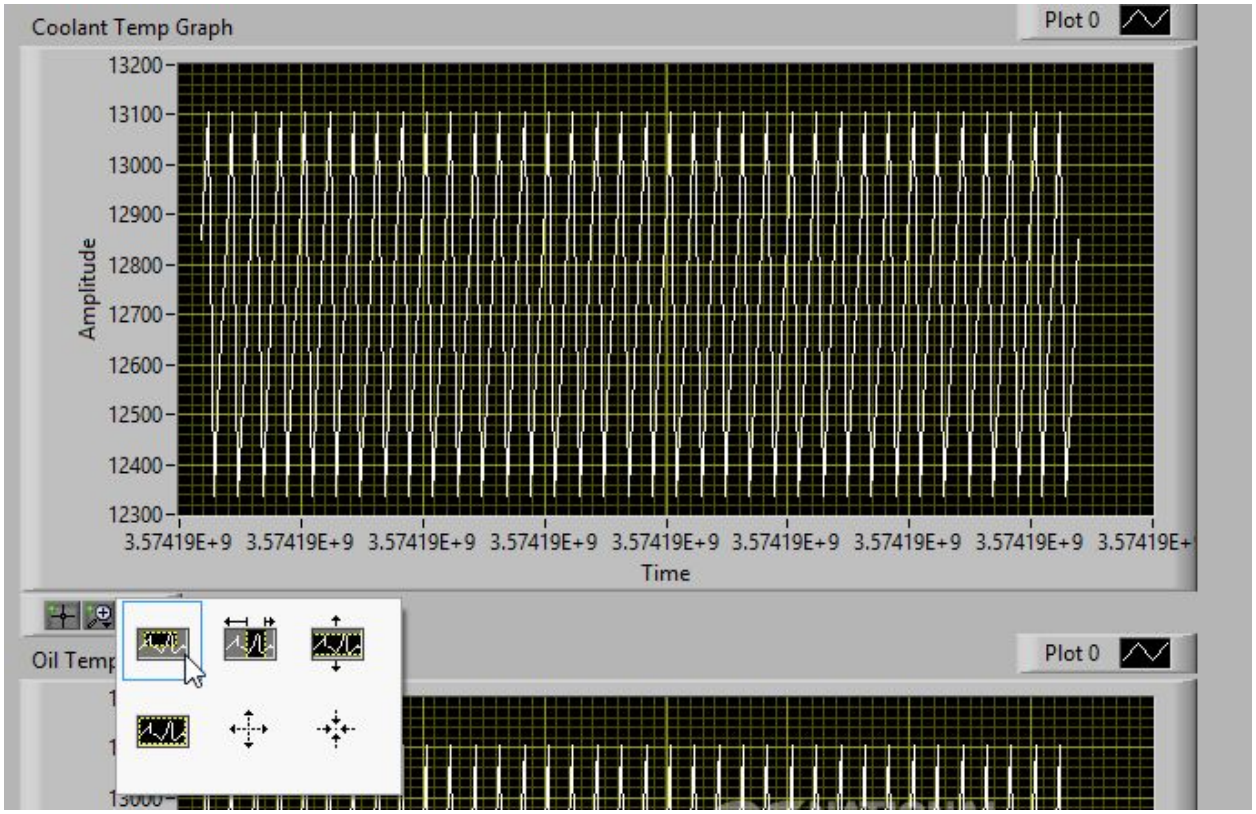

Figure 36 Graph Zoom Buttons

The top-left button is “Zoom to Rectangle”, used to zoom in on a rectangular area. The top-middle button is the X-Zoom, and the top-right button is the Y-Zoom. The bottom-left is the Zoom to Fit button. The bottom-middle allows the user to zoom in about a point, and the bottom-right allows the user to zoom out about a point.

Other Features

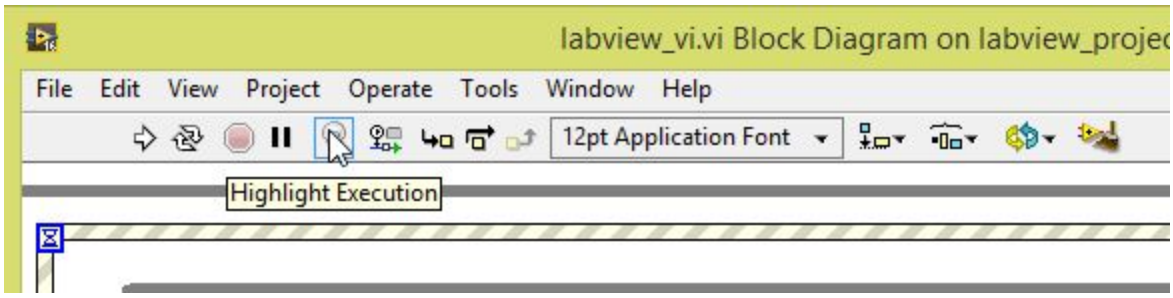

Figure 37: LabVIEW Highlight Execution Feature

The ‘Highlight Execution’ button in the Block Diagram page slows execution of the program, and displays all data transmitted between the blocks in the code. This feature is very useful for debugging purposes.

Appendix B - Microcontroller Code

main.c

```
/*
 * SAEFinalProgram.c
 *
 * Created: 5/8/2017 1:20:47 PM
 * Author : mhomler
 */

#include <avr/io.h>
#include <time.h>
#include "adc_drv.h"
#include "config.h"
#include "compiler.h"
#include "can_drv.h"
#include "can_lib.h"
#include "sensor_drv.h"
#include "at90can_drv.h"
#include "dvk90can1_board.h"
#define BAUDRATE 9600


typedef struct{
    uint16_t coolant_temp, oil_temp, oil_pressure, rpm, pot1, pot2, pot3, pot4;
} sensor_data;


void uart_init(){
    unsigned char baud = 51;
    UBRR0H = (unsigned char) (baud>>8);
    UBRR0L = (unsigned char) baud;

    PORTA = 0x00;
    DDRA = 0xFF;      // for LED testing

    PORTE = 0x00;
    DDRE |= (1<<PE1); // set bit 1 high to enable serial output

    //short int baudrate = 9600;      // equation to calculate baud rate value
    //unsigned int baud = CLKSPEED/(16*baudrate) - 1;

    UCSR0C = (0<<UMSEL0) | (0<<UPM00) | (0<<UPM01) | (0<<USBS0) | (3<<UCSZ0) | (0<<UCPOL0); //
bits 1+2 = 1 for 8 bits, no parity, 1 stop bit
    UCSR0B = (0<<UCSZ02) | (1<<RXEN0) | (1<<TXEN0); // bit 2 = 0 for 8 bits, set bits 3 and 4
to enable receiver and transmitter
    UCSR0A |= (0<<U2X0);    // affects baud rate, if 1 set then effectively doubles value
}
```

```
void TransmitEIA (uint8_t data) {
    if(data == 10){ data = 11; }
    /* Wait for empty transmit buffer */
    while( ! ( UCSR0A & (1<<UDRE0)));
    /* Put data into buffer, sends the data */
    UDR0 = data;
    //PORTA = UDR0;
}

unsigned char ReceiveEIA (void) {
    /* Wait for data to be received */
    while( ! (UCSR0A & (1<<RXC0)));
    /* Get and return received data from buffer */
    return UDR0;
}

void TransmitCAN(uint32_t id, uint16_t data0, uint16_t data1, uint16_t data2, uint16_t data3){

    uint64_t buffer;
    st_cmd_t message;
    message.pt_data = &buffer;
    uint16_t tempdata = 0xFF;

    buffer = 0;
    buffer = data3;
    buffer = (buffer << 10);
    buffer |= data2;                  // insert each sensor value into the buffer
    buffer = (buffer << 10);    // shifting buffer left by 10 after each entry to make room
for next
    buffer |= data1;
    buffer = (buffer << 10);
    buffer |= data0;

    message.dlc = 8; // verified - sets number of bytes being sent (8, entire frame) CANCDMOB
3:0
    message.ctrl.ide = 1; // before setting high: 11010011011111010111000001000001000000 10
    message.ctrl.rtr = 0; // high if requesting data from another node
    message.id.ext = id;

    message.cmd = CMD_TX_DATA;

    while(can_cmd(&message) != CAN_CMD_ACCEPTED);

    while (can_get_status(&message) == CAN_STATUS_NOT_COMPLETED);

    return;
}



void TransmitAll (sensor_data data){

    //Send data to RM024 using EIA-232
    uint8_t byte1, byte2;
    uint16_t eiarpm = data.rpm*10;

    byte2 = data.coolant_temp;
```

```
    byte1 = data.coolant_temp >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.oil_temp;
    byte1 = data.oil_temp >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.oil_pressure;
    byte1 = data.oil_pressure >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = eiarpm;
    byte1 = eiarpm >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.pot1;
    byte1 = data.pot1 >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.pot2;
    byte1 = data.pot2 >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.pot3;
    byte1 = data.pot3 >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    byte2 = data.pot4;
    byte1 = data.pot4 >> 8;
    TransmitEIA(byte1);
    TransmitEIA(byte2);

    //final newline char
    while( ! ( UCSR0A & (1<<UDRE0)));
    UDR0 = 0x0A;

    //Send data to Display using CAN
    TransmitCAN(525000, data.coolant_temp, data.oil_temp, data.oil_pressure, data.rpm);
    TransmitCAN(525001, data.pot1, data.pot2, data.pot3, data.pot4);
}

int main(void)
{
    can_init(0);
    //can_init(1);
    uart_init();
    sensor_data data;


    // ADC input will be voltage, 0-5 V - turns to integer 0-500
```

```
// coolant range: 0-300 F

data.coolant_temp  = 0;
int cooldir = 1; // using single ADC value as input for coolant temperature

//
data.oil_temp = 225;
int oiltdir = 1;         // ___dir values used to increase/decrease for simulation
data.oil_pressure = 25;
int oilpdir = 1;
data.rpm = 0;
int rpmdir = 1;
data.pot1 = 40;
int pot1dir = 1;
data.pot2 = 80;
int pot2dir = 1;
data.pot3 = 0;
int pot3dir = 1;
data.pot4 = 100;
int pot4dir = 1;

uint16_t vin; //testing ADC

while (1)
{
    vin = get_vin();
    data.coolant_temp = (vin*3)/5;
    data.oil_temp = (vin*3)/5;
    data.oil_pressure = vin/5;
    TransmitAll(data);
                // uncommented sections will cause variables to go from min to max
    /*                                  // rest are fixed values defined above
    if(cooldir == 1){
           data.coolant_temp++;
           if(data.coolant_temp >= 300){ cooldir = 0; }
    }
    else{    data.coolant_temp--;
                 if(data.coolant_temp <= 0){ cooldir = 1; }
           }

    if(oiltdir == 1){
           data.oil_temp++;
           if(data.oil_temp >= 300){ oiltdir = 0; }
    }
    else{    data.oil_temp--;
           if(data.oil_temp <= 0){ oiltdir = 1; }
    }

    if(oilpdir == 1){
           data.oil_pressure++;
           if(data.oil_pressure >= 100){ oilpdir = 0; }
    }
    else{    data.oil_pressure--;
           if(data.oil_pressure <= 0){ oilpdir = 1; }
    }
    */
```

```
        if(rpmdir == 1){
                data.rpm = data.rpm + 10;
                if(data.rpm >= 900){ rpmdir = 0; }
        }
        else{    data.rpm = data.rpm - 10;
                if(data.rpm <= 0){ rpmdir = 1; }
        }
        /*
        if(pot1dir == 1){
                data.pot1++;
                if(data.pot1 >= 120){ pot1dir = 0; }
        }
        else{    data.pot1--;
                if(data.pot1 <= 0){ pot1dir = 1; }
        }

        if(pot2dir == 1){
                data.pot2++;
                if(data.pot2 >= 120){ pot2dir = 0; }
        }
        else{    data.pot2--;
                if(data.pot2 <= 0){ pot2dir = 1; }
        }
        */
        if(pot3dir == 1){
                data.pot3++;
                if(data.pot3 >= 120){ pot3dir = 0; }
        }
        else{    data.pot3--;
                if(data.pot3 <= 0){ pot3dir = 1; }
        }

        if(pot4dir == 1){
                data.pot4++;
                if(data.pot4 >= 120){ pot4dir = 0; }
        }
        else{    data.pot4--;
                if(data.pot4 <= 0){ pot4dir = 1; }
        }


        /*
        data.oil_temp++;
        data.oil_pressure++;
        data.rpm++;
        data.pot1++;
        data.pot2++;
        data.pot3++;
        data.pot4++;*/

    }
}
```

config.h

```
//******************************************************************************
//! @file $RCSfile: config.h,v $
//!
//! Copyright (c) 2007 Atmel.
//!
//! Use of this program is subject to Atmel's End User License Agreement.
//! Please read file license.txt for copyright notice.
//!
//! @brief Configuration file for the following project:
//!             - can_spy_echo_example_gcc
//!
//! This file can be parsed by Doxygen for automatic documentation generation.
//! This file has been validated with AVRStudio-413528/WinAVR-20070122.
//!
//! @version $Revision: 3.20 $ $Name: jtellier $
//!
//! @todo
//! @bug
//******************************************************************************

#ifndef _CONFIG_H_
#define _CONFIG_H_

//_____ I N C L U D E S ________________________________________________________
#include "compiler.h"
#include <avr/io.h>
#include <avr/interrupt.h>
#include "at90can_drv.h"
#include "dvk90can1_board.h"

//_____ M A C R O S ____________________________________________________________

//_____ D E F I N I T I O N S _______________________________________________

       // -------------- MCU LIB CONFIGURATION
#define FOSC          8000          // 8 MHz External cristal
#define F_CPU         (FOSC*1000) // Need for AVR GCC

       // -------------- CAN LIB CONFIGURATION
#define CAN_BAUDRATE   250          // in kBit
//#define CAN_BAUDRATE   CAN_AUTOBAUD

       // -------------- MISCELLANEOUS
       // Using TIMER_2 as RTC
#define USE_TIMER8    TIMER8_2
//#define RTC_TIMER          2      // See "board.h"
//#define RTC_CLOCK          0      // See "board.h"

//_____ D E C L A R A T I O N S ________________________________________________

//______________________________________________________________________________

#endif  // _CONFIG_H_
```

Necessary Include Files

```
adc_drv.h
adc_drv.c
at90can_drv.h
can_drv.h
can_drv.c
can_lib.h
can_lib.c
compiler.h
dvk90can1_board.h
sensor_drv.c
sensor_drv.h
```

These files were used from Atmel's CAN library without modification.