ABSTRACT:

Aces Racing, a team of mechanical engineers at the University of Evansville, is looking for ways to improve performance and reliability on their car. Using a simple microcontroller to gather telemetry data in real time, the data will be sent back to a computer to monitor the car during operation. This project will also make use of databases to store the information collected during a run.
ACKNOWLEDGEMENTS

The project engineer would like to give his best thanks to the Aces Racing Team for their support and willingness to work with him on the project. To Dr. Don Roberts for his knowledge and experience in design software applications and insight in working with embedded devices. Special thanks to Professor Randal for his expert advice with knowledge of the CAN communication Protocol, use of his simulator, and advice on serial communication.

LIST OF FIGURES

Figure 1.A represents a successful noncorrupted message under the protocol used for the device. 1.B represents a non-successful noncorrupted message under the protocol used for the device.

Figure 2 a graph the base graph has been conditioned with a moving average conditioner.

Figure 3 represents an example of how the interface may be able to plot the track.

Figure 4 shows a screen shot of the interface while in operation of the data collected from the Arduino.
PROBLEM STATEMENT AND BACKGROUND

Aces Racing is in need of telemetry to monitor their car during operation. The team currently does not have any way of monitoring the car in real time or post-run and does not have sufficient funds to get the equipment to measure such data. With real-time telemetry data, many sub teams on the car will be able to better understand how the car is performing or if the car is breaking down. The team also needs some way of replaying this data, so a database would be helpful to analyze the runs of the car to compare performance and reliability over several runs.

Aces Racing is a student-run organization that competes in the Society of Automotive Engineers (SAE) Formula Race Competition. The race competition includes various events such as a drag strip competition, endurance, and even wet track conditions events, each designed to test the limits of the car under each condition. The race team is a not-for-profit organization that asks for sponsorships from companies small and large across the state of Indiana to fund the building of the car each year. The cost to get similar telemetry systems can run up to several thousand dollars, but the team believes that similar results can be done with cheaper hardware, and the results will be just as good for the accuracy in data that they require.

The team in the past has sponsored University of Evansville electrical engineers to develop sub-systems for the car that have used various devices to measure and read data components about the car, but most were more focused on the dashboard components such as the dashboard and being able to display speed and RPM. The previous electrical engineers have not transmitted the data back to a monitor application to watch the data in real time from a computer for further analysis.
Telemetry is important for multiples of reasons, but performance and monitoring of component health are the most important. Today, cars can monitor their engine temperature, engine running condition, and even tire pressure. Based off these inputs, the team can warn the users that their car may be in critical condition or if the car is not running correctly. In racing, knowing that the brakes were applied even a second early has been shown to cost the car valuable time in a lap, and being able to analyze the data from a set of previous runs will greater improve the efficiency of diagnosing performance or reliability problems.

With a front-end visual monitor, the race team would be able to see the car’s real-time performance while on the track. The ability for the race team to monitor the car’s condition in real time is beneficial because it will allow them to monitor their engine temperatures, fuel to air ratio, and RPM range, which are critical components to watch during tuning because they can cause engine and other mechanical failures. It will also be able to save the data received by the car so that eventually a playback function can be implemented or can be accessed and plotted with other tools.

REQUIREMENTS AND SPECIFICATION

This project must be able to display data that is received from an on-board telemetry device and display and save the data to a local database so that it can be read by humans. The program must be able to display various readings that are given off by the car via a CAN (central area network), accelerometer, and GPS (global position system) sensors. Each of these devices will be packaged by the microcontroller and sent back to the monitoring computer, where it can be displayed. The data will also be able to be saved to a database, where it can be analyzed later for more analysis.
The front-end application will graphically display the data from these sensors in a way that helps users to easily read all the data values. Methods of displaying data should include graphs, labels, gauges, and other visual aids that help the team indicate the car’s condition and performance.

A microcontroller that has ample ports and power will be used to read the sensors and packet the data to be sent to the computer monitor. The method used for communication will be a serial bus, over a set of 2-way radios capable of sending the data a great distance. The serial bus is the easiest way to send data to and from the microcontroller to the computer monitor; it eliminates the need to set up a network or do any kind of interpreting that may be required of other protocols.

The car’s accelerometer should be able to read the g-forces on the car and be displayed graphically on the screen. The accelerometer should also be able to measure the z-axis of the car, and this should be able to give the team data about how the car is handling bumps on the road. It is also expected that the data be enhanced from its original raw state so that it filters out any amount of noise that may occur during operation of the vehicle. In the case that the sensors already have some degree of filtering, the option to filter the data should be able to be disabled.

The interface will consist of various displays and configurations for the users to watch various parts of data. Current highlights to the interface will include a wave-form chart to be able to watch the accelerometer data that is being received by the car so that the technicians can visually see the change in accelerations that the car is going through in real time. Gauge clusters will be able to quickly identify the speed and RPM of the car. Gauges can also be highlighted to show when items are at a critical level or giving out. A real-time map will provide a trace of the car’s past movements and inputs from the driver. A fuel-to-air ratio display will show whether the car is running on too little or too much fuel.
All the data sent from the car to the computer will be stored in a local database only when the technicians have pressed the indicated record button on the screen. The record button will stay active until a technician has pressed the button, which will stop the recording and then allow the technician to specify a name for the file and let them save it. The data should be able to be loaded for a later time so that it can re-opened and the data can be replayed to the screen.

DESIGN:

The program will be mainly event-driven on a forms interface from Visual Studio. Once the user has connected to the microcontroller, the interface will begin to display all the data received to the screen.

The main interface will get updated every time that the program has received a message from the device. A message will consist of a ASCII character from our alphabet in lower or uppercase form, followed by a list of integer numbers that can contain a ‘-‘ sign at the beginning to symbolize that the value is negative or a ‘.’ to symbolize that the message is to be parsed as a double and will end with ‘*’. If any other ASCII character is found in the string, the message is not accepted. When a message is received, it is parsed into two components: the leading character and the value. The leading character is the key to a dictionary relating to the parameters of the message, such as the range of values and whether the value can be a double. If the message is accepted, the value mapped to the leading ASCII is mapped to screen. If the message is not accepted by the parameters, then the message is discarded and the interface is not updated or saved.

\[ A \rightarrow a.23*B-.75*D45*... \]

\[ B \rightarrow a.89*g.7532a3s... \]
*Figure 1.A represents a successful noncorrupted message under the protocol used for the
device. 1.B represents a non-successful noncorrupted message under the protocol used for the
device.

In the example above, Figure 1.0.A contains three messages that are all contained and all
accepted. In Figure 1.0.B, a.89* is accepted into the language but g.7532a is not accepted
because it does not end with a ‘*’, and ‘3’ is not the valid start of a message, so it is discarded,
and ‘$’ is not a valid start, so it is also discarded. When the message comes in and is accepted, it
is given a time stamp, and if the save feature is active, it will be saved to the database.

The communication will be done through Microsoft’s serial port class in their framework. The
serial port class allows the program to utilize its data received event, which can be used to read
the serial port whenever an object detects that a message has been sent to the program. In the
design of the program, it will use the redline method to read in the message. The message’s first
character will be looked up in a dictionary, where the parameters include the range of values and
whether a number can be a double. When the message is accepted, it can be saved both to the
database and to the screen.
The data that is collected from the car can be quite noisy, with spikes and troughs in data values due to quality of the hardware collecting the data and noise from the vibrations of the engine and suspension as the car is in operation. However, using smoothing functions such as a moving average enqueues the newest value, dequeues the last value, and takes the average of the current values in the queue over the last couple of samples. The filter should only slow down the program by fractions of seconds because of the incredibly low amount of operations that it has to run. A moving average holds some previous amounts of elements in a queue, and the average of the numbers in the queue is plotted on the graph instead of the actual value read by the sensors. The program will use Visual Studio graphs and will get updated with new values when the system goes through a cycle.

With the use of the program, the user will be able to see the car’s position on the screen from the onboard GPS module attached to the microcontroller on the race car.
This will be done by using the Web Browser Control in the Visual Studio Designer Controls, where we can load a script to the control and run it at a fast rate. The script will be written in HTML and JavaScript and will be invokable from the main method. The user will be able to zoom in and out on the control, as well as change the view setting to get a better idea of the scope of the run.

To record the data from a run, the program will use a SQLite Database because it is known to have fast query speeds and will have a low footprint on the program. The table will consist of three columns: the time stamp, which will be from the year all the way down to the millisecond; the sensor char that was read; and the value that was associated with the sensor reading.

The interface should be robust enough so that if the team were to implement playback into the program, a programmer would easily recognize that the method that displays the data from the serial stream is robust enough to be used to also play the data back screen as well. By using a common method for this operation, it can significantly reduce the size of the code structure.

Aqua Gauges will be used to represent the RPM, as well as temperatures of the engine and other sensors on the car. The values for these gauges are loaded with the values from the car’s sensors. Aqua gauges are extremely easy to implement and have a variety of different looks that will help the users easily read the values and identify when values are not correct.

To get all this data, the team will be using an Arduino, which has several libraries associated with the sensors previously mentioned. The Arduino is the easiest microcontroller to use, not only because the use cases are well documented, but also because all the sensors are running independent protocols to collect their information so that none of them interfere with each other. While the sensors can poll data at an extremely fast rate, it would be impossible for a human to
see all of the data that it is gather, so the data transmission will be timed out so that it doesn’t poll so often that there is wasted space in the database.

CONCLUSION/IMPLEMENTATION:

The scope of the project was much bigger than I previously thought it to be. Microcontrollers and electronic devices in general can be very tricky to work with, especially without the guidance of someone who is experienced with working with them or when there is very little documentation on some aspects of the components. The C# interface worked very well to represent data, organize code, and utilize the SQLite database.

I was able to display the waveform graph easily by setting up a chart control in the Visual Studio Designer, and to make it look as if the data were more fluent, I opted to make the maximum and minimum values as well as length values a fixed size. By doing this, it keeps the chart control from re-sizing, which can lead to longer update rates. To make the chart look waveform when it reaches 100 elements, I removed the first element of the chart and appended the new average of samples to the end of it. To save space in the main form code, I passed the chart through a class I made. This class would put the element into a queue and de-queue an element. If the queue length was greater than the sampling size, the queue elements would then be averaged, and the new sample would be appended to the chart control so that the noise is taken out of the data.
Figure 4 shows a screen shot of the interface while in operation of the data collected from the Arduino.

C# contains a Serial Port Class that has multiple methods and events that can read coming off the Arduino and read in the given data to the C# program. While the data received event is enabled anytime that the Arduino sends data to the serial port it is connected to on the program, the program will fire an interrupt that will do a serial read line to read in the buffer until the newline character is reached. When the line has been read in the Message Parser class will ensure that the message is valid. A valid message will have to be within the constraints that are defined by the program. The constraints can be that the value can be a double, and within a range of acceptable values. These parameters are stored in a dictionary so that they can be accessed in constant time by the parser so that if more constraints or controls were to be added, the lookup time of the constraints would not change. When the message passes through the parser and is valid, it will get mapped to its control by the identifying character of the message and then will pop an invoke with the value to load to the control. The invoke of message ensures that while updating the control, it is thread safe operation so that the program does not fault.

If the user has declared a file name to save to, the user can now press the record button on the interface to record the data to the database. The identifying character of the message and the value gets saved to the database. The timestamp is assigned to the message when it is stored in the database. To assign the database file name, the control will open another form that will allow the user to specify a file name and click the done button. After some data has been recorded by toggling the record button by the user, the user can ensure that the data base has successfully saved to the database. The SQLite Database is set up with a basic table of the timestamp, identifying char, and value. The database was programmatically created inside another class so that adding or reading data from the database could be simplified for the programmer. To view
the data, the user can click a button on the Main Window when there is data in the database and the database is not recording. This button will open a form that contains a list box control from Visual Studio that will allow the user to scroll through the data and check to see what is in the current database.

To use the gauges on the interface, I imported the Aqua Gauge library and added the control to my main form with the design I liked. To adjust where the needle pointed, I simply set the value of the needle position to what the message value was from the parser. The histogram on the control allows the user to set how many samples they would like to display of the fuel-to-air ratio, which will allow the user to see if the engine is running lean or rich in fuel over a course of time. The histogram uses fast points to plot the fuel-to-air ratio value on the y-axis with respect to RPM on the x-axis.

The Web Browser Control was able to map the GPS location of the car via loading the application with a web-based script based on html and JavaScript. The actions to run made scripts were able to be done by invoking said script through the Web Browser Control in the visual studio library. Bing Maps API was able to be used to plot markers on the webpage through a function call from the main form to the invokable script on the web browser. To make the display look like there is less noise, the polling rate of the GPS signal was decreased to every second rather than every 100ms because it made the map control seem cluttered and noisy. The Bing Maps API also allowed the interface the amount of extra external features that the base map had to offer, which helped on clutter. The current control allows the user to specify the amount of area they would like to look at and to drag and zoom in out of the control.
Figure 5: Shows an example of the web browser control showing points received from the GPS unit from the Arduino.
REFERENCES


Marks, Steve. *Google Maps V3 API.*

Thirugnanam, Ambalavanar. “Aqua Gauge.” *Aqua Gauge - CodeProject,*


Biography

Chris Johnson is a software engineer who has studied at the University of Evansville in Evansville, Indiana. He is currently a senior pursuing his bachelor’s degree in Computer Science and a part-time employee at Embry Automation and Control, where he does work on industrial systems, as well as work on algorithms to automate the lumber industry’s saw mills to increase efficiency. Chris has taken an introductory class on digital systems and plans to further his study in embedded systems.