# Senior Project 2005

The senior project for Colby Whitney, Brandt Deakin, Scott Halliday, and Brian Flint, was a vehicle diagnostics and monitoring system. The system consisted of a low power motherboard and processor, OBDII (vehicle interface protocol, and connection) to RS232 converter and a monitor with touchscreen.

We purchased a Via Epia 800, low power motherboard and processor, RS232 converter, monitor and touchscreen. We designed and built the power supply for the computer, smart switch to control the power supply, touchscreen controller, and custom housing for the monitor and touchscreen for our hardware portion of the project. For the software portion of the project we modified freediag, an open source vehicle monitoring system, wrote a server program to download and display log information from the system in the car, and wrote the embedded software to run the power supply, smart switch, and touchscreen controller.

The power supply was designed to meet the specifications of a Via Epia 800. In meeting the specification for the Epia 800, the ATX specification had to be closely adhered to as well. ATX is the standard for all desktop computers, and the Epia required this type of power supply.

Our power supply did not have to meet all of the requirements of the ATX spec for two reasons. In the specification it is assumed that 120 AC voltage will be used for the source, we had a 12-14 volt DC voltage for our input. The second difference between our power supply and an ATX standard is the current output requirement. The Epia is a low power board that enabled us to design for 30 watts, instead of the hundreds of watts required for true ATX compliance.

In addition to the mainboard and processor, we had a 2.5 inch hard drive, touchscreen controller, smart switch, and LCD to power. The easy parts were definitely the embedded processors, LCD, and hard drive, because they were low current requirements.

The LCD required 12 volt DC at 0.5 amps. For this we chose a LM7812, for several

reasons. The 7812 is rated to regulate an input from 12 volts to 35 volts DC. There was no concern for exceeding that limitation of the 7812 because the data from our tests never showed a voltage higher than 18 volts from the car battery. The instance of 18 volts occurred moments after the car started the alternator and there was a voltage surge. Secondly, the 7812 was rated for output current in excess of one amp. This was more than enough for the LCD, which on the bench power supply drew a rock solid 0.5 amps. The most important reason we used the 7812 was the fact that it was on the free parts list, which meant we did not have to pay for it.

Hard drives like the type used in this project only require 5 volts. We conducted some testing and found that about 0.5 amps were required to power the drive. An LM7805 was chosen to regulate the power for the hard drive for similar reasons for choosing the LM7812 for the LCD. The LM7805 is part of the same product line as the LM7812. It was tolerant of voltages in excess of what we were providing, the output current was more than sufficient to power the drive, and again this product was on the free parts list.

Our touchscreen controller needed to be powered by our power supply as well. There were two ADCs, one Motorola 68HC11E1 microcontroller, a crystal, other logic devices, etc. These all required 5 volts, and testing showed the current requirements in the milliamp range. Again, an LM7805 was chosen to regulate power for this device. The tolerant input range and more than sufficient current output capability in addition to the fact that it was free made this an easy choice.

The smart switch needed to be powered all of the time because it turned the power supply on and off, so we decided to give it a standalone source. The same microcontroller was used on this device as the touchscreen controller, so we decided to use another LM7805. We also decided to put the regulator on the smart switch board to eliminate the need to run wires back and forth from the power supply board to the smart switch board.

Designing for the motherboard and processor was the most challenging part of building the power supply. The current requirements although low by other computer standards, were larger than we were used to dealing with. According to the operating guidelines document we received from Via, we needed to provide:

- 3.3 volts at 2.43 amp
- +5 volts at 2.07 amp
- +5 volts standby at 0.52 amp
- +12 volts at 0.11 amp

The operating guidelines document mentioned the requirement for -12 volts but did not indicate the current requirement. For this information we found a PDF for the ATX standard (www.formfactors.org). According to the standard, a maximum of 0.3 amps were required for the -12 volt output.

For the 3.3 volt we used an Anachip AP1501-33. This regulator is a 5 pin 150KHz pulse width modulated DC/DC converter. The regulator was capable of input voltages between 1.23 volts to 37 volts and could source up to 3 amp. For implementation, two capacitors, one Schottky Diode, and one inductor were necessary. We ordered several, tested them, and found there performance to be sufficient.

The decision was made to try to stick with the parts we knew and were free. We used three LM7805 regulators in parallel for the 5 volt input to the motherboard. One regulator can only support 1 amp, and we needed more than 2 amps, so we chose to use three regulators to provide up to 3 amp max. The 5 volt standby current requirement was relatively low at about ½ amp. We again used another LM7805.

The 12 volt current requirement was fairly low. The motherboard only required 0.11 Amp max. The LM7812 met all of the requirements, and for the same reasons we have used the

other LM78xx regulators mentioned above, we decided to use it.

The -12 volt was the most complex chip on the power supply, but still not difficult to implement. We used a Maxim MAX765 inverter, which comes in a 8 pin DIP package. The chip takes in between 3 and 16 volts, and can source 250 milliamp. This is 50 milliamp less than the maximum rating of the ATX spec, but because our system required far less current on all of the other voltages, we bet that this part would be sufficient. For the whole circuit we used 4 capacitors, 1 inductor, and 1 Zener Diode.

When it came to design we wanted to keep it simple. The smart switch flipped a relay which provided power to the power supply, which in turn powered the motherboard, hard drive, LCD, touchscreen controller, and fans. There was also a connection from the power supply to the car battery which bypassed the relay and smart switch. This was to provide the input voltage to the regulator which was then providing the 5 volts standby to the motherboard. This voltage has to be on all the time, so the second input to the power supply was the most simple solution we came up with.

Another aspect of design was, where to place everything on out perforated board. We had many of the same parts, which were simple 3 pin regulators. We lined these up and connected the input to all of the inputs of the regulators using a stripped wire as a rail. The ground connection was done the same way. Then the outputs that were shared for current requirement reasons were tied together and the other outputs were left alone. The other, slightly more complex regulators, received a wire from input and ground, accompanied by the other required capacitors, resistors, and inductors.

The last aspect of design was the connection from the power supply to the devices it would be powering. We acquired all of the ATX standard connectors and their cables from a dead power supply. There were many wires which all required the same voltage, so for these we

connected all of them to the output of the regulator(s). Once all of the connections were made, we were ready to power it on.

When we powered it on for the first time it ran for just under a minute and then shut down. After some debugging it became very clear that the regulators would require heat sinks. After another unsuccessful test, but better results, a fan was added. With the heat sinks and active cooling, the power supply powered the whole system. The next power consideration is to know when it is safe to turn on.

The purpose of the smart switch is to protect the system from startup and shutdown power spikes that occur in an automotive environment. It also controls the startup and shutdown sequences of the operating system and power supply.

Over the summer, we discovered that when the car is started it does not always give the constant 12 volts that we needed to run our power supply. Some cars drop to less than 8 volts when they are starting. To prevent this problem we had the idea of waiting 3 seconds after our system receives power before starting the computer. We also needed a way to detect whether the car was on or not. We solved both of these problems by creating the smart switch.

To create our smart switch, we used a 68HC11 Motorola processor programmed in assembly code. We programmed the code into the eeprom. We used ports E and C for input and port B for output.

The program itself is simple. Using a signal wired from the ignition of the vehicle to port C on the 68HC11, the smart switch is able to determine the state of the car, whether it is on or off. When the car turns on, the smart switch begins a startup sequence. First the switch waits three seconds, to make sure the system is stable. Then, it activates a high power relay, through port B, that connects a relay which supplies power to our power supply. Once the power is applied, the smart switch tells the motherboard, also through port B, to turn on. The smart

switch receives a signal from the motherboard, through port E, that tells the smart switch its state, whether the motherboard is on or off.

When the vehicle is turned off, the smart switch performs the above sequence in reverse order. The smart switch waits 3 seconds, to make sure that the signal didn't spike, then tells the computer to turn off. When the motherboard signals that the system is ready to be turned off, the relay is disconnected.

The enclosure for the touchscreen and monitor was made by Colby as a project in another course. CS5963, Advanced Manufacturing, taught the use of Alpha1, importation of solid models to FeatureCam, NC code generation using FeatureCam, and actual manufacturing of designs from raw materials.

The Alpha1 system is an advanced research software base, supporting geometric modeling, high-quality graphics, curve and surface representations and algorithms, engineering design, analysis, visualization, process planning, and computer-integrated manufacturing. Alpha1 was used to create solid models of the enclosure and lid for the purposes of manufacturing.

For our purposes FeatureCam was used to import the solid model created using Alpha1, and generate the NC code (code that the milling machine will use). First the model was imported with stock defined, then the tools were selected, the order of operations defined, and simulations run to ensure things were in order. Once the simulations are done and we were satisfied, FeatureCam was used to generate the NC code. This code was then input to the milling machine, the aluminum blocks placed inside the machine, and the machine did the rest.

The touchscreen and touchscreen controller have a critical role in our project. They add an obvious cool factor to the project as well as simplify the user interface with our system. As designed, the touchscreen is the only way the user has to interact with our system. Given a tap on

the screen we change from car monitoring to system control and even shutdown. The major components of the touchscreen are the touchscreen overlay, a microcontroller, analog to digital converters, and a RS232 serial driver.

The touchscreen overlay that we chose was manufactured by 3M Touch Systems. There were two factors in determining which brand/model we were going to use in this project. First, it was based on who could provide a size that was closest to the size of our display. Second, was accuracy and ease of use. There are a few choices. Matrix style touchscreens divide the screen into large blocks typical eight rows of eight. The disadvantage is that they do not allow buttons to be anywhere on the screen, and they must fit within the block. The capacitance styles of touchscreens are very difficult to implement, plus they can easily trigger by accident if you get near them. Analog Resistive seemed to be the a natural fit as they can be very accurate depending on how many bits your analog to digital converter is, and they are relatively simple to implement.

The resistive touchscreens are so named because they are basically resistive voltage dividers. They're composed of two resistive sheets separated by a very thin insulator usually in the form of plastic micro-dots. When you touch the screen, you deform the two resistive sheets just enough to make electrical contact between them.

The trick is to not measure the resistance created by the connection, but to measure the division of the voltage when the connection was created. To do this, apply a voltage across one plane of touchscreen, then measure the voltage on the other plane. If there is no current flowing across the measured plane, then the resistance on that plane does not affect the voltage levels seen at the connection point. The measured voltage represents a ratio from the beginning to end of the plane that the voltage was applied across. For example, if the supplied voltage was 5 volts and measured was 4 volts, then the position on the screen, in the direction of the applied voltage,

is 4/5 of the entire screen or 1/5 down from the source. Notice that this only gives the position on one axis, in order to get the position on the other axis, the same operation must be performed again, but with the opposite plane excited.

The touch overlay used in this project is an 8-wire resistive FG or film on glass technology. The 8-wire has a couple advantages over the standard 4-wire. The 8-wire allows the designer the ability to separate the 'excite lines' from the measurement lines; as well as giving you the ability to average the measurement from both sides of the screen. Also, 8-wire is not as susceptible to temperature changes and drifting.

The measurement is taken by 8-bit analog to digital converters. This will allow 256 unique values for each axis of measurement, from top to bottom or left to right. The analog to digital converters used in this project were National ADC0809, which uses a successive approximation model to determine the digital equivalent. They are somewhat slow, as compared to a flash converter, around 100 microseconds for one read; but they are very accurate. Internally they use a full 256R ladder, instead of the typical R2R ladder, this helps prevent load variation on the reference voltages. In our application, we needed two separate voltage references, one for the front plane and one for the back plane of the touchscreen. Because two Vref+/- were required, two ADC were used, even though the ADC0809 can handle up to six analog inputs. Both ADC start and end-of-cycle pins are tied together, so that when the ADC is finished with a measurement, it immediately begins taking the next. Both ADC 8-bit digital outputs are tied together, so only one ADC's output can be enabled at a given time. The ADC's digital bus is analyzed by a microcontroller.

The device that controls the excite lines of the touchscreen, as well as the output enable of the ADC is a 68HC11 microcontroller from Motorola. It is also responsible for the interpretation of the data given to it from the analog converters, and the communication of that

information to our main computer in the car. The microcontroller has a small program that is written in its own assembly code, that takes care of this communication and control.

The program on the microcontroller polls portC, which is the input from the ADCs. If portC remains in its default state, the program will do nothing. The default state is all zeros, meaning that no connection has been made between the touchscreen planes. When a touch happens, the left/right plane is measured by writing 0x60 to portA. This switches a multiplexer in such a way to cause 5 volts to be applied to the right side of the touchscreen and 0 volts on the left. A nice side affect to using the mux to control our power lines is that while the mux is set to power one axis of measurement, the other axis is floating, which is the desired result We also enable the output of ADC#1 at the same time. With a short delay, the X-coordinate of the touch is ready on portC and is stored. Next we need the Y-coordinate of the touch. This is done by writing a 0x10 to portA, thus changing the mux such that top/bottom planes are excited and ADC#2 is enabled. One of the limitations of a touchscreen is that the two conductive planes create a capacitance between them, so it takes a little time to fully charge. The next part of the program delays while this charging is taking place. On a side note, if you don't wait long enough for this charge to happen, debouncing will not work, you get some weird measurements from the ADC's. Now with the top/bottom plane excited, the Y-coordinate is stored. The next part is due to a complication with the chosen microcontroller. If the Rx/Tx pins of the serial port are driven high on reboot, then the microcontroller goes into a special state (bootstrap mode) that prevents the code from running. To solve this problem, on reboot, the Rx/Tx lines are disconnected by default and must be reconnected to transmit. Another mux is used to create the high impedance affect during reboot. Now that the system has measured both X/Y coordinates, the serial line is reconnected by writing a '1' to bit7 of portA. Four ASCII characters are transmitted, an Ω symbol followed by the X-coordinate, then an ∞ followed by the Y-coordinate.

The ohm and infinity symbols were chosen because their value in hex exceeded any value that we would ever see from the touchscreen. This would make it simpler to parse on the PC side of things. The last step of the program is to switch configurations back to default, disconnect the serial line, and then wait for the connection on the touchscreen to be broken. The program determines the touchscreen connection has been broken when the ADC returns a value of zero, which can only happen when no connection is made between the front and back planes of the touchscreen. The touchscreen controller then waits for another press.

Our client uses the standard C libraries to initialize and use the serial port on the computer to receive the button press information. Our client computer originally was only equipped with a single serial port, which caused something of a problem. The system used to communicate with the OBDII bus requires a serial port with a real UART, as does the method we use to receive raw information from the touchscreen controller. Most inexpensive USB to RS232 converters use a chipset that does not support this type of communication, but this was not learned until we had already purchased two of these converters.

To overcome our shortage of resources, we purchased a PCI card with a real UART on it. It had a pair of serial ports which more than covered our needs. After installing the card and starting the machine up, it was obvious from the system log that the machine saw the PCI card and it's two serial ports. It took nearly an hour to realize that they had been initialized as /dev/ttyS14 and /dev/ttyS15 for some reason. Once this had been learned, we were able to communicate with both the vehicle and the touchscreen controller at the same time. The client software includes code that simply waits in a loop for new information to come in via the serial port. When a location on the touchscreen is pressed, a packet containing 4 bytes is sent to the host. The first byte is the number 252, which was used to tell the host that the next byte is the X coordinate. The third byte is the number 253, which was used to tell the host that the final byte

was the Y coordinate. We formed the packet this way so that we could avoid potential problems with timing should packets get broken up or misinterpreted.

The user interface program is written in C++ and uses the FLTK libraries for buttons and callbacks. Our client program runs inside the X.org windowing system on top of a modified Fedora Core 4 installation.

Fedora Core was selected because of the familiarity that we had with it. We were able to install it without any major problems by hooking up an optical drive up to the computer and using the install media. The optical drive wasn't part of the final install and as such was only temporarily used. We parted from the default install in that we used traditional partitions on the hard drive rather than using the soft partitions that are more common now. We ended up uninstalling several packages that we knew would not be needing allowing bootup time to be faster.

The install was performed using a normal CRT for video output, which was also used for most of the debugging process as well. To prepare X.org for the LCD we used we had to do quite a bit of experimenting to find a proper setting. Because our LCD used a composite signal via an RCA connector, the video quality was underwhelming. The native resolution of the screen actually turned out to have a lower image quality than 800 x 600 with large fonts. Using the CRT also worked quite well during debugging as it was a much more comfortable resolution to work at than the native resolution of the LCD.

We used FLTK for our buttons and callbacks because two of the team members had previously used it in other classes. FLTK stands for "Fast, Light Toolkit", and it is all those things. It is intended to be used to write interfaces for programs in a rapid development environment, allowing the developers to focus on making the features work right, and not just look good. Our experience with FLTK allowed us to leverage the framework from one of the

assignments in a previous class as a starting point for our client interface. Nearly all of that code was dropped quite quickly as our code matured, but it helped lay the foundation. We chose FLTK 1.1 because it's more simple than the later revisions. We downloaded the source for FLTK, compiled it, and installed it and soon found that it had issues with libpng. Libpng is used as a library for rendering and saving png image files. We had planned on using the png file format for the images used in the GPS portion of our interface and fell back to the gif format. This ceased being a problem when time ran out and we couldn't incorporate our GPS interface.

Our client interface was written to be friendly with the limitations of the small nature of the LCD and the nature of the touchscreen we chose to use. We chose to use the top 1/6$^{th}$ of the screen for a series of buttons that are used to select between a number of panels corresponding to the different features of our project. In the first panel, we placed the information that is extracted from the engine computer. We chose not to display all of the information that we had available because of the limited space available on the screen. We chose the most valuable 9 fields of information and formatted it for Imperial/English units. The next panel was going to be the GPS information. Originally, the left 25% of the screen was saved for a number of buttons allowing the on-screen image to be manipulated. It was then decided to have "hot" spots on the image and allow it to use the whole panel. This would have allowed more of the image to be shown. This also permitted us to show more detail of the area that the user desired to view as more space was available.

The third and forth panels of the interface allowed the user to perform a number of more advanced functions. The third panel was titled "SysControl", and gave the user the opportunity to turn off the computer manually (as opposed to waiting until turning off the car.) It also provided the user with a button that triggered a script that uploaded the logs and such to the computer that was configured as the server. The first button simply called the system program,

“poweroff”, which brings the system to init level 0 and tells apm to turn off the computer. The forth panel has a large text area where all of the DTC’s, monitored as well as those that have been thrown. The forth panel also has a button which calls a script telling freediag to clear the thrown DTC’s.

The client software ends up having a very spartan, easy to understand interface. The user is presented with the important information, with the unnecessary details saved for later review on the host machine using the server-side software. This design does limit the amount of information that is available to the user while in the vehicle, but it also limits our legal liability should the user spend too much time using the computer rather than watching the road.

The loop that was used in the client software was one of the most power consuming features of our project. It caused our motherboard to increase the power consumption by 800 milliamps. This is more than the hard drive, or any of the fans. It was necessary to have the loop function this way because the 4 bytes sent by the touchscreen controller didn’t have any handshaking or flow control. If the host didn’t pay attention to every byte that came down the line, it could be lost. Various small sleep times were investigated, but it was found that even a small idle time would prevent our system from functioning properly.

Aside from the challenges in communicating with the hardware through the serial interface, we also had a few challenges configuring the program to decode the locations accurately. One of the causes of this challenges was that the overlay for the screen enclosure matched the opening to the LCD and not the touchscreen overlay. This was the right thing to do, as the appearance would be lessened by seeing the filler material behind the touchscreen overlay which is transparent. The reason that this caused a problem is that the readings sent back from the hardware’s ADCs correspond to the whole overlay, and not the approximately 60% that was used.

This was overcome by some fairly extensive testing and configuration which allowed us to get the coordinates for the corners of the area that was used. By so doing, we were able to re-calibrate our program to accurately decode location based on the smaller area. We were not able to maintain our 8-bits of accuracy from the ADCs, but we found that it wasn't an issue given the size of our buttons on the screen.

Once we have logged our data on the client side (car), we needed a way to display the logs over time. We created a server to display the data we have recorded while driving the car. For example, after driving around for a while in the car with our device on, the user decides to pull into his garage. He presses the sync with host button on our touchscreen and immediately uploads the file from the current trip to the server.

We bought a wireless access point to communicate from our car computer to the server and send our files. We did not fully implement this part of the project because we didn't get the access point we had planned for. The new access point required a different voltage than we were prepared for. So we took a USB flash drive and transferred our files manually to the server to show our data.

After our server receives the files from the client, GD takes the output file from freediag and parses out the data for Intake Air Temperature, RPMs, Engine Load, and MPH. This data is created into 4 different graphs over time. These graphs display all data from the previous upload until the current upload.

When the server program is first run, it makes a call to scripts that are responsible for creating the images that will be displayed. These scripts parse the log files from the client with the grep command looking for the lines that contain the information that will be needed for a specific graph. For example, the line, "grep "Vehicle Speed" output.txt > /tmp/rpm.txt" takes all of the lines that contain information about vehicle speed and places them in a temporary file.

Next, a while loop reads that temporary file line by line and takes out the numerical data. Awk is used to get the third field, using the command, "awk < /tmp/rpm3.txt '{print $3}' >>/tmp/rpm2.txt." This command adds the information to a second temp file. The third field also contains the characters "mph", such as "37mph", and the "mph" must be removed before the information can be given to the perl script that will call GD to create the graph.

Once the file containing the array of numerical values has been created, the main script for each graph concatenates the data from that file onto a header file containing the first few lines of the perl script. These few lines end in the opening descriptor for the array of values, so the file with values is then added at the appropriate place. While the file with values was being created, a second file containing empty descriptors for every value was created with the same number of fields. This is concatenated on right after the values in the perl script that the main script is building. After this is done, the final portion of the perl script is added to it, and then it is run. The perl script makes a call to GD, passes it the values as well as the labels for the graph, and GD will return the graph. The perl script was adapted from a CGI program with a similar function. The bash scripts that build the perl script were written from scratch for this use.

Our server program, written in c++ with FLTK libraries, has 3 main buttons. The program is written in a very similar form to our client program. The first button shows the 4 graphs made by GD. These graphs are loaded and displayed to the screen from four separate jpeg files for MPHs, RPMs, Intake Temperature and Engine Load.

The second button displays the entire freediag output file as text to the screen and allows the user to be able to scroll through it. Each entry is taken from freediag every 3 seconds. This log file can become quite large, in which case our server program takes some time to start up. We felt it important to add this feature as there are several bits of information that we did not show in the graphs. It also allows the user the match data up with a precise time that the data

was gathered from the car.

The third button displays a maintenance log, that is updated to a file every time a new line is added. There is a user input line and a button to add the entry to the file. When the button is pressed, the inputted line is immediately appended to the file and on subsequent runs of the program, the new data is present.

The server software performs several functions that can not be easily accomplished on the client side due to interface constraints. It also offers a way for the user to store the logs for later comparison or review. The server software could easily be modified to have any additional functions or graphs that the user desired. It has been designed in a straight-forward, extensible manner.

If you ask any of us about any part of the project we would probably say we would do it differently now. As the saying goes, hindsight is 20/20. This is definitely not a bad thing. We all learned a tremendous amount about real hardware and software engineering, setting goals and deadlines, completing a project, and group dynamics in a project.