Design of an engine controller interface and its role in industry

Second Semester Report
Spring Semester 2009
By
Josh Gabler
Saurav Joshee
Dean Kooiman

Prepared to partially fulfill the requirements for
ECE402
Department of Electrical and Computer Engineering
Colorado State University
Fort Collins, Colorado 80523

Report Approved: __________________________________________

Project Advisor

____________________________________
Senior Design Coordinator
ABSTRACT

The Drivven 2008-2009 senior design team worked developing an engine controller interface for use in the automotive industry. The current interface is a computer with specialized software that interfaces with Drivven’s controllers. Changing the state of the engine controller through the computer interface has proven difficult. The team goal for this year was to design and implement a robust and aesthetically pleasing self contained interface unit. The unit had to be easy to use, with 4 quadratures, an 8 position switch, and 2 push buttons for user input. A LCD screen was also provided to display the current status of the engine controller. The new unit incorporated all of the functionality of a traditional engine controller as well as the ability for the user to create their own display of variables.

Our work was broken up into two parts. Last semester, a large amount of work went into modifying the hardware and software configurations given to us by last year’s team. Because the project was a continuation project, it was necessary to determine what exactly was accomplished by the previous group. This involved examining many lines of coding, breaking it up into a more readable format and then testing the current functionality. There was also a great deal of research involved with hardware modification and correction. User manuals for the Digi board as well as more research regarding PCB design had to be conducted in order to first get a grasp on the material. The current operating system was NetOS. We used ThreadX and wxWidgets to implement the threads, interrupts, and GUI designs. Because C++ course were two hundred level courses for us, we had to review the language. The previous team developed a GUI display as well as ISRs that enable the controller to effectively display engine control variables. We created a more dynamic GUI and were on our way to getting the pins interruptible. A boost power supply was also designed to provide the voltage needed to supply the LCD. A finite heat element analysis was also conducted to get an idea of power dissipation and heat sink limits on a PCB.

This semester, our focus shifted drastically. Our design goal was to port from the NetOS environment used last semester to Labview’s Elemental IO. This required porting existing source code to a new compiler. Embedded LabView allows for better GUI displays, easier modification, and allowed the program to be placed on many different types of boards. The goals for hardware remained the same. We worked to complete a PCB design and to make a design that would support the current LCD display.
## CONTENTS

Abstract ................................................................................................................................. 2  
List Of Figures ..................................................................................................................... 4  
List of Tables ....................................................................................................................... 4  
Chapter I: Introduction ........................................................................................................ 5  
Chapter II: Previous Senior Design Overview ................................................................. 6  
Fall Semester overview .................................................................................................... 8  
  Chapter III: Board Design ................................................................................................. 8  
  Chapter IV: Power Supply Design (Boost Supply) ............................................................ 10  
  Chapter V: Finite Heat Element Analysis ........................................................................ 11  
  Chapter VI: Hardware I/O in NEtoS ............................................................................... 12  
  Chapter VII: Dynamic GUI Design ................................................................................. 15  
Spring Semester Overview .............................................................................................. 18  
  Chapter VIII: Implementation of PCB ............................................................................ 18  
  Chapter IV: Porting to LabView ...................................................................................... 22  
Chapter X: Marketability and Manufacturability ............................................................... 27  
Chapter XI: Project Management ..................................................................................... 27  
Chapter XII: Ethical Concerns/Issues ............................................................................. 28  
Conclusion and Future Work ............................................................................................. 28  
References .......................................................................................................................... 29  
Appendix A: Abbreviations ............................................................................................... 29  
Appendix B: Budget .......................................................................................................... 30
LIST OF FIGURES

Figure 2.................................................................................................................. 6
Figure 1.................................................................................................................. 6
Figure 3.................................................................................................................. 7
Figure 4.................................................................................................................. 8
Figure 5.................................................................................................................. 9
Figure 6.................................................................................................................. 10
Figure 7.................................................................................................................. 11
Figure 8.................................................................................................................. 11
Figure 9.................................................................................................................. 13
Figure 10............................................................................................................... 14
Figure 11............................................................................................................... 14
Figure 12............................................................................................................... 14
Figure 13............................................................................................................... 16
Figure 14............................................................................................................... 17
Figure 15............................................................................................................... 18
Figure 16............................................................................................................... 20
Figure 17............................................................................................................... 21
Figure 18............................................................................................................... 22
Figure 19............................................................................................................... 23

LIST OF TABLES

Table 1.................................................................................................................. 12
Table 2.................................................................................................................. 19
Table 3.................................................................................................................. 24
Table 4.................................................................................................................. 30
Table 5.................................................................................................................. 30
CHAPTER I: INTRODUCTION

The need for an engine controller interface is paramount in the automotive industry. Drivven’s commercial tool Cal VIEW allows a user to interface with their engine controllers through lab view. The engine controller interface, at this point, uses a keyboard and mouse controlled by the user. The keyboard and mouse, however useful in normal operation of a computer, are not optimal in the operation of engine controllers. It is easy for an operator to slip up and input a bad value or drop the mouse at an inopportune moment. The idea behind our engine controller interface is to take the mouse and keyboard functionality and represent them with 4 easy to operate quadratures, an 8 position switch, 2 push buttons and a LCD screen. The device must be light and robust as well as astastically pleasing. The device must run on 9 to 30 volts DC, this range is to allow for battery or wall operation. The end goal of this device is to provide a better interface to modify engine controller parameters to the operator. This year’s project is a continuation from last year’s design of an engine controller interface. Chapter II contains a critical review on the scope of work from last year’s senior design team.

This year, instead of the idea of using the Digi Connect Core 9P 9360 development board, we are designing our own PCB with the selection of the required components from the development board. The current problem involves designing a power supply that provides 9V to the LCD from the development board that has an input voltage of 3.3V. Thermal simulation of the PCB has to be taken into consideration for the design process of the board. Protel 99SE software is being used to generate the board layout design next semester. Once, the pre-layout is generated from the software, it’s going to be surface mounted to get the actual design. More detailed explanation is done in Chapter III about the PCB and hardware configuration.

One of our constraints for this year was that we had to use the Digi ConnectCore 9P 9360 development board that was used in last year’s project. Traditionally, the Digi board runs NetOS and ThreadX, a real time operating system. Last semester we developed GUI interrupt Service routines and threads under the NetOS environment. Detailed results from this development can be found in Chapter VI. Last semester, we also developed a more sophisticated GUI in NetOS using wxWidgets. A detailed report of this can be found in Chapter VII.

This spring, we came to the conclusion that the other operating system was not what we wanted in our final implementation. We were told that LabView would be better because it provided a better GUI and thread development system. A LabView implementation is easily portable between most Arm9 chipsets (excluding the Digi Arm9 chipset) whereas a NetOS/ThreadX implementation is constrained to products developed by DIGI. Because of this fact, our goal for this semester was to port LabView onto the Digi ConnectCore 9p 9360 even though it was an unsupported board. Refer to Chapter VIII for our guide and approach to porting.
CHAPTER II: PREVIOUS SENIOR DESIGN OVERVIEW

The 2007-2008 senior design team provided the box enclosure, hardware layout and basic source coding. Last year’s hardware layout is provided in Figure 1. This view shows the four quadratures, push buttons and LCD. Figure 2 shows the block diagram of how the interface will interact with an engine. The team built an interface that communicated over Ethernet using the TCP/IP protocol with a labview simulation of an engine.

Figure 3 contains a flow chart for their program. This program loop contained hard coded display routines and a network poll routine that allowed them to communicate with their Labview simulation. The program starts by setting up the Ethernet interface and then drops into a configuration loop. This configuration loop allows the user to choose from 2 display layouts. When the display layout is picked, the program drops into the main loop. This main loop displays variable changes through polling and updates the engine control simulation accordingly. The LCD configuration is not used because there is not a 9V trace on the development board.
The hardware provided a basis for our project, but the devices were not functioning properly. The green quadrature was broken and one signal on the rotary switch was inoperable. The chosen LCD required 9V which was not appropriate for the development board. The development board is designed to accommodate a 12V LCD. The I/O poll routines, however helpful, were not operating correctly. The GUI by the team was not as involved as the requirements specified. The team appeared to focus more on the hardware and did not have time to complete a GUI that was highly functional. While it was fairly easy for the user to operate, it lacked a lot of functionality that would be useful in industry. User’s were guided through a display and only given two choices for their layout. They were not able to decide on the variables used or the quadratures to control them.
Digi ConnectCore 9P 9360 development kit is being used for our actual board design. This year, the components that exist in the board are being sorted out to design a user interface Board.

Figure 4 is the development board consists of PoE connectors for PoE application board. Two user LEDs, Two user keys, one debug LED, USB host ports PCIe socket SPI header I2C digital I/O expansion (8-bit) with 9-pin header, Ethernet connector and a 3.3V lithium battery. The current measurement option that the board has is +3.3V, +5V. The test points are +3.3V, +5V and GND. The power supply is from +9/+30 VDC.
Figure 5 is a system block diagram of the general layout of the components on the Board. Note that these components might subject to change. A CAN port in the block diagram could be added for communicating with the ECU. The board has a power supply that supplies +3.3V to the board that is stepped up to a 9V trace to feed into the LCD. The LCD was measured at a value of 9 V at a current rating of .092A for the best resolution. However, at voltages higher than 9V the LCD had slightly better resolution; the current was fluctuating and was not stable. So, we decided to choose 9V as the base voltage of the LCD. Ethernet and the RS 232 are the serial communication ports used for communicating with ECU. The USB port is used for the software interface. The VGA is not shown in this block diagram but might be added to the board for debugging purposes for future operations. The four quadratures are used for operating changes in different parameters along with the push buttons that perform Boolean operations. The quadratures and buttons operate at a voltage of +5V from the board itself. This design is going to be implemented using Protel-99SE and given for surface mount operations.
The main goal in the design of the board was to get the LCD running. Other than that almost everything is a re-analysis of last year’s work. Last year’s team did not get the LCD screen to display the output. The development board had a trace for the LCD operating at 12 V and not 9 V. A boost supply was designed for this purpose. Boost supply also known as Boost converter steps up a low input voltage to a high output voltage.

**Figure 6**

Figure 6 shows a boost convert LM2735X. This boost supply operates at a frequency of 1600 kHz. L1 is the inductor, D1 is the diode, Rfb1, Rfb2, Renable are the resistors, Cin and Cout are the capacitors. The values of the inductors and capacitors shown are the specified values for the desired output. The input ranges of voltages for this design were 3.2V to 3.4V and the desired output voltage was 9V at an output current rating of .092A. The inductors along with the capacitors are used to offset any voltage ripple of the signal provided.

The switch when turned on, the inductor absorbs energy and increases the inductor current. When the switch is turned off, the inductor dissipates the energy through the diode and creates a difference in current that further creates different input and output
voltages. This is how the 3.3V input provided steps up to 9V output. This design was generated through the software at National.com.

CHAPTER V: FINITE HEAT ELEMENT ANALYSIS

Finite heat element analysis is done in the component of the PCB by thermal simulation analysis. This amounts the power dissipation capabilities of the board. It is always a good practice in board layout design to know the power dissipation of the components on the board for a longer life span of the board. PCB acts as a heat sink due to its large flat surface which is effective in transferring heat from components to the air. However to every heat sink, a limit exists. A common rule of thumb in calculating the total power dissipation is from a heat transfer function.

Power dissipation \( Q \) = Heat transfer coefficient \( h \) * Surface Area \( (SA) \) * (Maximum component temperature \( (T1) \) – Air temperature \( (T2) \))

The heat transfer coefficient is largely dependent on airflow speed. So PCB board at airflow of 20°C with 2000 linear feet per minute would give a power dissipation of 1.8 watt/m². For this senior design, a simple finite heat element analysis was done to get an idea of how much power components dissipate. This would help in knowing the tolerance of the PCB board for power when all the components are added. We wanted to carry out a FHEA for the design of power supply mentioned in chapter III but the input voltage was not in the specified range (has to be >14 volts) with the software we used. So, we defined a range from 14V to 20V and designed a power supply that had an output current of 2A with an output voltage of 3.3V to get a feel for FHEA.

Figure 7 shows the mounted components of the power supply on a PCB generated with the input values of voltages from 14V to 22V with an output voltage of 3.3V and an output current of 2A.

Figure 8 shows the thermal simulation of the board with the components with different degrees of heat dissipation. Table 1 shows the power dissipation of each component on the board.
Components | Power Dissipation (W)
---|---
C<sub>i</sub>(Capacitor) | 0.00149
C<sub>o</sub>(Capacitor) | 7.456E-5
D<sub>1</sub>(Diode) | 0.7484
Inductor(L1) | 0.2816
U1(transistor) | 0.7296

**TABLE 1**

The total power dissipated by the designed power supply would be 1.761W. This power is for a single section of a board layout. Power from every section of the board is considered to analyze the total power dissipation of the entire board.

---

**CHAPTER VI: HARDWARE I/O IN NETOS**

In the fall, one of our major improvements was the Hardware I/O. The main loop was altered to allow for a more dynamic approach to the graphic user interface display. The new loop is designed to allow for high speed interaction between devices and visible frames. This high speed interaction will come through the development of interrupt service routines (ISRs) and static data storage.

The general purpose input and output routines were modified from the previous year’s group. The eight position switch poll routine was redesigned to reflect changes in the main program loop. The quadrature and push button poll routines were first fixed and thoroughly tested. They were then adapted to interrupt service routines to be registered under ThreadX.
The selector switch routine had to be modified to return state values instead of incrementing a counter. The 8 position rotary switch does not require an interrupt service routine as this poll completes very quick. In addition this device only controls the frame that is displayed on the LCD and its output is not needed for any other methods. Figure 9 details the new 8 position switch poll routine.

```c
//ROTARY3STATE IS THE MOST SIGNIFICANT BINARY
//BCD = 213
unsigned int ret = 0;
if (Rotary2State == 1) {
    if (Rotary1State == 1) {
        if (Rotary3State == 1) { //111
            ret = 7;
        } else { //110
            ret = 6;
        }
    } else { //101
        ret = 5;
    } else { //100
        ret = 4;
    }
}
else { //011
    if (Rotary3State == 1) { //011
        ret = 3;
    } else { //010
        ret = 2;
    }
    if (Rotary2State == 1) { //001
        ret = 1;
    } else { //000
        ret = 0;
    }
}
```

FIGURE 9

The quadrature and push button poll routines were modified slightly to improve their accuracy. Debouncing was introduced into the push button poll routine to eliminate false positives on button presses. The development of ISRs for both of these devices has been extremely difficult. Although the routines have been developed and modify data, they are not being called by ThreadX. An example of an ISR for the blue quadrature is provided in Figure 10. The ISR takes a pointer to a parameter that is passed in by ThreadX. This ISR is triggered off a rising edge on the blue quadratures second signal. We then grab the current value of the quadratures first signal off pin1. If this value is non zero the quadrature has moved left, otherwise the quadrature moved to the right. We then modify static data storage to reflect this change. Figure 11 contains the registration of the blue quadratures ISR with ThreadX. We register the ISR and then print a meaningful result returned from the ThreadX kernel.
The development of threads under NetOS and ThreadX has been difficult. Currently the Graphic User Interface (GUI) thread runs in the root thread. This is fine because our program entry is after the root thread has already setup the supporting threads. The network communication thread has been removed from the old loop; the network thread is now created and registered using our thread helper class. Figure 12 shows how the network thread is allocated and registered under the ThreadX kernel.

```c
void BlueQuadratureISR (void * parameter) {
  unsigned int PinResult = 0;
  NAgetGPIO pin(APP_INPUT_PIN1_Blue, &PinResult);
  if(PinResult > 0) {
    QuadratureInterruptData[1]++; // Quad moved left
  } else {
    QuadratureInterruptData[1]--; // Quad moved right
  }
}
```

//Install Blue ISR
result = naISRInstall(APP_INPUT_PIN2_Blue, (NA_ISR_HANDLER)
BlueQuadratureISR), &parameter);
naISRInstallPrintResult(result, "Blue Quad: ");

The development of threads under NetOS and ThreadX has been difficult. Currently the Graphic User Interface (GUI) thread runs in the root thread. This is fine because our program entry is after the root thread has already setup the supporting threads. The network communication thread has been removed from the old loop; the network thread is now created and registered using our thread helper class. Figure 12 shows how the network thread is allocated and registered under the ThreadX kernel.

```c
void ThreadSetup::setupNetworkThread() {
  TX_THREAD networkThread = {0};
  CHAR * networkThreadName = "Network Thread";

  void * stack = malloc(NETWORK_THREAD_STACK_SIZE);
  printStackMalloc(stack, networkThreadName);

  UINT networkStatus = createNewThread(&networkThread, networkThreadName,
NetworkThreadEntry, stack, NETWORK_THREAD_STACK_SIZE);
  printTxThreadCreate(networkStatus, networkThreadName);
}
```

Information is shared between the GUI and Network threads through static data storage. GPIO changes on frames propagate through memory to the network thread where they are batched up and sent across the network to the ECU. Under ThreadX each thread runs in its own block of memory. The thread can dynamically allocate and release memory within its own block. However threads may not overflow out of their memory space, if this occurs an exception is thrown and program execution terminates. To implement threads under ThreadX the developer needs a pointer to the last block of memory that was allocated. The developer then passes this pointer as well as the size of the block of memory that the new thread will run under.
CHAPTER VII: DYNAMIC GUI DESIGN

All of our programming in the fall was done in the Digi ESP development environment. This involved the use of C++. The C/C++ Development Toolkit (CDT) is a set of Eclipse plug-ins that provided C and C++ extensions to the Eclipse workbench. Because none of our team members had ever programmed any form of graphics using C++, this task involved a lot of overhead. Luckily, a C++ book that we had from a previous class contained a little information about common C++ graphic designs and provided the source code for how they were created. We were able to use this to give us an idea of how we could create what we wanted. After a little research we were able to start programming.

When we first received the board from last year’s project, it was evident that they had spent a lot of time on developing the hardware aspects of the board. Very little time was spent actually programming the board to be a user friendly environment. The user was given a page and when he hit the enter button was taken to a screen that had four variable boxes, each linked to a specific quadrature. While this was a very good demonstration of the functionality of the hardware it did not provide much for the user.

Our goal for last semester was to get a dynamic GUI finished that allows the user to determine what type of layouts that would be used. This is to be accomplished using a rotary switch. Position zero on the switch is the configuration page and links to the other seven positions on the switch. It has the ability to determine what each of the seven other pages will look like, what variables will be used on each page, what quadrature will connect to each variable, and the sensitivity of each quadrature.

Our work started with just getting the rotary switch to be able to go from the configuration page to each of the other pages by moving its position. This was done by using an ongoing while loop and determining if a change had occurred to the position on the rotary switch. If a change had occurred it would set the page to visible and hide the other pages. We were able to complete this and got it working pretty well. Below is a UML diagram of the basic structure of the code:
The seven configurable pages were called generic frames and they could be modified based on the parameters that they receive. The generic frame class had a `SetFrame()` method which was able to set the page's current frame to a certain layout. This was accomplished through a series of switch statements that would first create the frame that was desired and then would set the frame to visible. Whenever the data was updated we used a method called `Poll()` that was called throughout the hierarchy until the correct frame was given the data. We kept the method named `poll()` even after we implemented interrupts because it kept our code constant.

The possible frames that the user has to choose from are a frame that contains 1-4 separate variables, a frame that contains a list of variables, and a frame that contains a multidimensional array of variables. Each individual variable can be controlled by a single quadrature. A list of variables requires two quadratures; one to move left and right between variables and one to change the value of the current variable. An array of variables requires three quadratures; one to move left and right, one to move up and down, and one to change the values of the highlighted variable. Because each page has at most four quadratures, only a finite amount of layouts could exist on a page. The person could have up to four individual variables on a page, two list, a list and two single variables, and array and one single variable, etc. This fact made programming rather easy.

After the ability to implement different layouts was created, most of the time was spent creating the graphic layout for each type of frame. This actually turned into being a
rather time consuming task due to the fact that many aspects overlapped and viewing current setup required a new build of the system. Individual variable layouts were creating by making boxes for each variable. The color of the quadrature chosen would then be displayed in the box as well as the sensitivity and current value. Below is an example of one of our layouts.

The graphics of the layouts are pretty self explanatory. Each of the color coded variables link to a quadrature.
Last year’s senior design team was using the Digi developmental board itself. This year we decided to design our own PCB since all the components on the Digi board were not being used and hence a smaller board would be easier. Also, mounting the quadratures, push buttons, and the 8 position switch to the board itself were considered in the process. The stages of PCB implementation are explained below.

For a PCB design, there many design programs available. Over the two semesters, we tried three different software’s starting from protel to multisim and then finally to Altium. So, the process described below is based off of Altium PCB design software.

Creating Electrical Parts Library:

Creating an electrical part is necessary when you do not have available predefined components on the library database. Altium has a wide variety of library components so we were fortunate enough to make just few components. As seen below in the figure, there are pins assigned. The pins are often decided as input or output pins and were made based off of the Digi development board specification sheet.

Designing Electrical schematic:

After creating all the library components not listed on the library, we design the electrical schematic of the system. The electrical schematic is a representation of the actually functionality of the board with the desired components. The components can be connected different ways and it all depends on the person designing the board. We chose to represent connections from one pin to another with net labels instead of wiring.
every pin. In net label, each pin is assigned a specific name. So, for example if pin 1 is supposed to be connected to pin 3, we can assign a net label of the same kind to represent its connection. Table 2 is a listing of all of the pin numbers with their respective net labels for the Digi Arm9 processor.

<table>
<thead>
<tr>
<th>X1 Connectors</th>
<th>X2 Connectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin Numbers A9M9360 Name</td>
<td>Pin Numbers A9M9360 Name</td>
</tr>
<tr>
<td>3</td>
<td>PWRGOOD</td>
</tr>
<tr>
<td>5</td>
<td>TCK</td>
</tr>
<tr>
<td>7</td>
<td>TDI</td>
</tr>
<tr>
<td>9</td>
<td>TRST#</td>
</tr>
<tr>
<td>11</td>
<td>CONF1/</td>
</tr>
<tr>
<td>13</td>
<td>CONF3/</td>
</tr>
<tr>
<td>15</td>
<td>CONF5</td>
</tr>
<tr>
<td>17</td>
<td>CONF7</td>
</tr>
<tr>
<td>25</td>
<td>SPIB_DI</td>
</tr>
<tr>
<td>31</td>
<td>SPIB_CLK</td>
</tr>
<tr>
<td>37</td>
<td>LCDDO</td>
</tr>
<tr>
<td>39</td>
<td>GND</td>
</tr>
<tr>
<td>41</td>
<td>LCDD3</td>
</tr>
<tr>
<td>42</td>
<td>LCDD4</td>
</tr>
<tr>
<td>43</td>
<td>LCDD5</td>
</tr>
<tr>
<td>44</td>
<td>LCDD6</td>
</tr>
<tr>
<td>45</td>
<td>LCDD7</td>
</tr>
<tr>
<td>46</td>
<td>LCDD8</td>
</tr>
<tr>
<td>47</td>
<td>LCDD9</td>
</tr>
<tr>
<td>48</td>
<td>LCDD10</td>
</tr>
<tr>
<td>49</td>
<td>LCDD11</td>
</tr>
<tr>
<td>50</td>
<td>LCDD12</td>
</tr>
<tr>
<td>51</td>
<td>LCDD13</td>
</tr>
<tr>
<td>52</td>
<td>LCDD14</td>
</tr>
<tr>
<td>53</td>
<td>LCDD15</td>
</tr>
<tr>
<td>54</td>
<td>LCDD16</td>
</tr>
<tr>
<td>55</td>
<td>LCDD17</td>
</tr>
<tr>
<td>62</td>
<td>LCD_PWREN#</td>
</tr>
<tr>
<td>63</td>
<td>LCD_AC_BDE</td>
</tr>
<tr>
<td>64</td>
<td>LCD_FRAME</td>
</tr>
<tr>
<td>65</td>
<td>LCD_HYSNCY</td>
</tr>
<tr>
<td>66</td>
<td>LCD_CLK</td>
</tr>
<tr>
<td>67</td>
<td>LCD_LENZ</td>
</tr>
<tr>
<td>79</td>
<td>GND</td>
</tr>
<tr>
<td>112</td>
<td>12C_SDA</td>
</tr>
<tr>
<td>113</td>
<td>USB_PWR</td>
</tr>
</tbody>
</table>

| Table 2 |
Creating footprints for Component:

Generally, after electrical components are defined, it is necessary to create footprints. Footprints are defined as the amount of area the components take on the board. This is necessary for the PCB. Hence, when doing that, we looked at the specification datasheet where we found the dimensions of the components. We used those numbers in Altium and defined the component footprint. If the component is a surface mount, it generally requires the information of a pad. Through hole components require through hole information. In the figure below, numbers 1 and 2 are the pads.

![Figure 16](image)

Compiling schematic for Errors:

After all the components on the schematic are assigned with footprints, it's time to update the schematic to a PCB. Before that, it's always a good practice to compile the schematic and make sure there are no errors on the schematic. The errors that we encountered during the process were no signal drivers, net label undefined, etc. After the compilation is successful, it is ready to be updated to a PCB schematic.

Updating schematic on PCB:

This is considered to be the final process of the design, even though in this stage there are a lot of things to be considered. In this stage, the electrical schematic gets updated to the PCB. Figure 17 is a sample view of what the board should look like. We did not get this far with the PCB design because we had complications getting the LCD to work. Hence, the following view can be taken as a sample and not the actually board design.
Unfortunately, we were unable to come out with the final product. The LCD pins could not be configured as there were fewer pins for the processor than the LCD. This is because we could not configure interruptible pins to perform multiple functions for the LCD. It would have been much easier if we had bought an LCD that was compatible with the processor itself. Hopefully, that will be taken into consideration next time if this project is further continued.
CHAPTER IV: PORTING TO LABVIEW

The biggest change to our project this semester was changing our operating system from NetOS to RTXOS in order to incorporate LabView. The existing operating system had limited functionality and made designing graphics very difficult. Figure 14 in Chapter 4 illustrates this. The GUI is functioning, but is not very sophisticated. GUI’s designed in LabView are for more advanced because LabView has predesigned blocks already created. Figure 18, below, shows a LabView GUI.

After making the decision to switch to LabView, we realized that this task was going to be very challenging. The 32-bit RISC ARM processor that the Digi board has is widely used in industry. Because of this fact, many boards have been created so that you can easily implement the NI LabVIEW Embedded Module. In order to do this, you need the board to operate with the RTX embedded operating system. The problem is that our Digi board is not one of these supported boards. Below is a description of the process that we used to attempt to port LabView to the Digi Connectcore 9p 9360.

From our research, we were able to indentify four major tasks in order to get LabView ported onto the board. Here is a list of the general steps. We did not attempt to develop peripheral and I/O drivers due to time constraints.

- Port the RTX Real-Time Kernel
- Integrate the Real-Time Agent module for debugging
- Create the target in LabVIEW and incorporate the Keil toolchain
- Develop peripheral and I/O drivers

LabView provided a porting guide that showed the basic steps for porting LabView onto an unsupported board. In order to get a better grasp of the material, the first thing done
was to follow the tutorial and perform the steps needed to get the board ported. This example demonstrated how to do so on a Philips NXP LPC3180 microcontroller on the phyCORE-ARM/LPC3180 evaluation board. When we did this, we created a project in Keil and then followed the steps provided. The source code was already contained within Keil and was zipped up. The contents of this source code were only about ten different classes. After that we were able to port the RTX Real-Time Kernel as well as integrate the Real-Time Agent for debugging. We did not bother with developing peripheral and I/O drivers because we wanted to make sure we could get that far with the Digi board. After finishing this step we felt that we had a good grasp for what needed to be done. We were not able to demonstrate our “mock-port” of the Phillips board because we did not actually own the board, but we were able to get the setup code running and create an executable. From this tutorial we were able to diagram the steps that we needed to take for our project.

**Incorporating the Keil Toolset**

The first thing that we needed to get done was to incorporate the keil toolset. When creating a project, certain variables must be entered. These are unique to each board and must be obtained by looking through documentation. Figure 19 shows one of the configuration pages that must be completed when creating a project. On this frame you must enter the clock speed as well as the addresses to important places in memory.

![Figure 19](image_url)
## Memory Address Locations

<table>
<thead>
<tr>
<th>PROCESSOR MODES</th>
<th>defn</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER</td>
<td>0x10</td>
</tr>
<tr>
<td>FIQ</td>
<td>0x11</td>
</tr>
<tr>
<td>IRQ</td>
<td>0x12</td>
</tr>
<tr>
<td>SVC</td>
<td>0x13</td>
</tr>
<tr>
<td>ABT</td>
<td>0x17</td>
</tr>
<tr>
<td>UND</td>
<td>0x1b</td>
</tr>
<tr>
<td>SYS</td>
<td>0x1f</td>
</tr>
</tbody>
</table>

**INTERRUPT BITS**

<table>
<thead>
<tr>
<th>BIT</th>
<th>defn</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0x80</td>
</tr>
<tr>
<td>F</td>
<td>0x40</td>
</tr>
<tr>
<td>ISR ADDR</td>
<td>0xA0900164</td>
</tr>
<tr>
<td>SWI EXIT</td>
<td>0x11</td>
</tr>
</tbody>
</table>

**LED ADDRESSES**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>defn</th>
</tr>
</thead>
<tbody>
<tr>
<td>LED ADDRESS</td>
<td>0x90600018</td>
</tr>
<tr>
<td>LED BASE</td>
<td>0x90600030</td>
</tr>
<tr>
<td>LED ADDR 1</td>
<td>0x90600024</td>
</tr>
<tr>
<td>LED BASE 1</td>
<td>0x90600034</td>
</tr>
</tbody>
</table>

**BASE ADDRESSES**

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>defn</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY CONTROLLER</td>
<td>0xA0700000</td>
</tr>
<tr>
<td>SCM</td>
<td>0xA0900000</td>
</tr>
<tr>
<td>SRAM ENABLE</td>
<td>0x2</td>
</tr>
<tr>
<td>SCM DYNAMIC CS4</td>
<td>0xA09001D0</td>
</tr>
<tr>
<td>BBUS RESET</td>
<td>0x90600000</td>
</tr>
<tr>
<td>CLOCK CONFIG</td>
<td>0xA090017C</td>
</tr>
<tr>
<td>BBUS ENDIAN</td>
<td>0x90600080</td>
</tr>
<tr>
<td>BBUS ENDIAN VALUE</td>
<td>0x10CD</td>
</tr>
<tr>
<td>BBUS CONFIG</td>
<td>0x90600070</td>
</tr>
<tr>
<td>PALL</td>
<td>0x103</td>
</tr>
<tr>
<td>SRAM ENABLE VALUE</td>
<td>0x00080000</td>
</tr>
<tr>
<td>NUM OF WORD IN RAM TABLE</td>
<td>4</td>
</tr>
<tr>
<td>CLOCK REGISTER</td>
<td>0xA090017C</td>
</tr>
<tr>
<td>RESET REGISTER</td>
<td>0xA0900180</td>
</tr>
<tr>
<td>DRAM CLOCK ENABLE REG</td>
<td>0x90600104</td>
</tr>
<tr>
<td>DRAM CLOCK ENABLE MASK 0</td>
<td>0x3FF1FFFF</td>
</tr>
<tr>
<td>DRAM CLOCK ENABLE MASK 1</td>
<td>0x3F1FFFF</td>
</tr>
<tr>
<td>DRAM CLOCK ENABLE MASK 2</td>
<td>0x31F1FFFF</td>
</tr>
<tr>
<td>DRAM CLOCK ENABLE MASK 3</td>
<td>0x1FF1FFFF</td>
</tr>
<tr>
<td>CLOCKS ON</td>
<td>0x7F</td>
</tr>
<tr>
<td>ENABLE MODS</td>
<td>0x6F</td>
</tr>
<tr>
<td>RESET BIT</td>
<td>0x80000000</td>
</tr>
<tr>
<td>WATCHDOG CONFIG</td>
<td>0xA0900174</td>
</tr>
<tr>
<td>WATCHDOG TIMER</td>
<td>0xA0900178</td>
</tr>
<tr>
<td>WATCHDOG FORCE RESET</td>
<td>0x90</td>
</tr>
</tbody>
</table>

**TABLE 3**
Table 3, above, is a listing of all of the important memory locations that we felt would be used to get the board running LabView. These would either be entered into the initial Keil setup or made as defines within our code.

Normally, when LabVIEW compiles, it makes calls to µVision, directing it to open up a preconfigured project that has been customized to the specific target. The only problem was that there was no preconfigured project for our board. This meant that we had to create our own by going through the start-up code for Digi ConnectCore 9p 9360. This seemed to be a task that sounded clear-cut and easy, but proved to be far more challenging.

By inspecting the documentation of the Digi ConnectCore 9p 9360, we found that the source code was initialized in a file called INIT.arm. INIT.arm provided the assembly code references and the low level subroutines needed to bring up the ConnectCore CPU. The routines included memory addressing, memory chip selects, memory testing, GPIO interfacing, interrupt service routines, and an interrupt vector table. Because we did not know the effects of removing one of these functions, we chose to implement them all. In order to do this we had to start at the initial file and then include every file that it referenced, as well as its location in the startup code folder. We then had to look through subsequent files and do the same until all of the needed files were included in our build path. We ran into a problem when compiling these files because the Keil compiler did not initially know how to preprocess assembly and C code together. In order to get by this fact, we first started to comment out lines of code that caused the compiler to fail. By doing this we were able to reach an endpoint for the startup code, however, it was not able to branch successfully into the RTX. Realizing this, we went through and modified certain parameters in an attempt to include the commented out functionality. Eventually, we realized that doing this would not produce the results we wanted.

Using a compiler flag we were able to compile the assembly code with C preprocessor definitions. This meant that we did not have to comment them out. It also allowed us to use Keil’s assembly compiler instead of the C compiler to compile INIT.arm. Once INIT.arm was compiled successfully, it began expecting C library files that supported its subroutines. We began the arduous process of including precompiled library files as well as including C source code that was to be compiled by the C compiler. By compiling things separately, we did not have to comment out as many lines of code. However, we ran into an issue with certain C defines not being provided at compile time. These defines were used to set up memory addresses, so we had to look up which ones were valid and explicitly define these values. Towards the end, the library files started expecting precompiled ThreadX libraries. It was at this point that we realized that this approach was also unacceptable because this would mean that ThreadX would be operating on top of the RTX operating system. ThreadX was the real time operating system that NETOS ran. If we were to continue down this approach we would have two real time operating systems running concurrently on our board. This means we would have had two interrupt vector tables, two operating systems calling
routines, and two sets of timer events. This would have tremendous performance impact, but it would have made pin mappings impossible. If there were no pin mappings, there would be no way to have input output for our device.

Our final approach was sifting through the C and assembly source code looking for #define and const declarations. We were specifically looking for references to memory addresses, hardware addresses and GPIO. We were also able to discover and condense assembly routines to blink provided LEDs. We were going to use these routines to ensure that our startup code was successfully executing on the Digi processor. Using the excel tables that were created we attempted to create a bare bones startup routine, without support for networking, VGA, or USB. We ran into issues in porting the GPIO setup routine. GPIO was provided under NetOS through a pre-compiled library and header file. The GPIO was setup by defining addresses and devices through a multiplexer. It was not possible to just include the library and reference the header file because the library expected to see other library files which referenced ThreadX. Finally because we were not able to view the source code for this setup routine we were once again left with a device that would have no input or output to speak of.

Porting the RTX Kernel

Step two for porting LabView is to port the RTX kernel onto the board. The RTX Real-Time Kernel is an executive for microcontrollers that provides a real-time operating system. The Real Time Operating System handles any common tasks such as scheduling, maintenance, and timing issues. To run the RTX Real-Time Kernel, the following hardware resources are required and reserved by the RTOS on the board

- **Peripheral Timer** for generating periodic ticks. The RTX Real-Time Kernel needs a count-up timer. If the timer used is a count-down timer, you need to convert the timer value.
- **Timer Interrupts** to interrupt the execution of a task and to start the system task scheduler os_clock_demon().
- **Forced Interrupts** to force a timer interrupt when isr_ functions are used.

We then copied RTX_Config.c to our device from the Keil startup directory. We had to modify certain defines within this code to get the RTX working. After that we needed to modify the Startup.s file. This file contains functions that are executed directly after a CPU reset. It provides various functions such as stack, memory, and clock initialization, as well as functions for mapping exception vectors.
Real Time Agent

When debugging embedded applications, many times it is not practical to halt a program to view or modify memory contents. By adding the Real-Time Agent to the application, you can view and modify the target memory “on the fly” without stopping the program. The Real-Time Agent is a small code module you can add to any ARM applications that allows your program to communicate back to the μVision Debugger.

We determined that the Real-Time Agent was not developed for our target because when we searched through its files we did not see our particular board listed in its directory. The folder “\Keil\ARM\RT Agent\RTX” told us this information. Because of this fact, we opened up the target that was provided for another Arm9 board similar to ours. From here we opened the RTX_Config.c file for and played around with that to see if we could get this function able to run with our board. We could open up the Agent along with the μVision Debugger, but were unable to test because we could not get our startup executable.

CHAPTER X: MARKETABILITY AND MANUFACTURABILITY

The goal for this project has remained the same throughout the past couple of years. It is believed that a product like this one would be of great value to individuals who work with, design, and study engines. It will allow for a safer environment as well as more control over the engine being used.

Drivven has decided that this is a device that they would like to pursue and gave us the great opportunity of doing so under their expert supervision. When a complete product is finished, we believe that it could be manufactured for production.

If LabView is eventually ported to the Digi board or if a new board is obtained, it will greatly increase the manufacturability of the product. Because LabView is able to run on an RTX operating system, it allows for the program to be put on many different types of boards. If the device continues to run NetOS, it will not be as practical because there are far less boards that incorporate this specific operating system.

CHAPTER XI: PROJECT MANAGEMENT

Our team was made up of three members. Saurav Joshee was the team leader. The responsibilities of the team leader were to communicate with our advisor Matt Viele, coordinate meeting and conference calls, and make sure progress was being made on assignments. The group work was split into several different chunks. Saurav was in charge of the hardware design and Layout throughout the entire year. In the fall, Dean
Kooiman was in charge of hardware I/O and Threads. Josh Gabler was in charge of GUI design in the fall. During the spring semester, Josh and Dean both worked on porting LabView, each trying their own methods. During this time they met every other week to discuss progress and give one another hints.

CHAPTER XII: ETHICAL CONCERNS/ISSUES

There are many ethical issues that come to mind when you are creating a project like this. First of all there are the ethical issues that you have with regards to the customers. The device should not cause harm to any of the customers. For our device, this means that it should not allow an illegal value to be entered for a variable and if it reaches an unsafe state it should shut off. Another issue that you have is with the company that you are working for. You should not disclose sensitive information about the project to other groups. The company is paying you for your time, so you should comply with their policies. To my knowledge, we have complied with every ethical and professional responsibility that we could think of.

CONCLUSION AND FUTURE WORK

Our team gained valuable experience this year. We discovered the fundamentals of porting an unsupported development board to a new operating system. We were able to experience many different programming environments and languages. Working with NetOS, ThreadX, wxWidgets, LabView, and RTX showed that we were able to adapt to new environments and learn a lot about their structure. We also gained a lot of knowledge about PCB design and their associated computer programs. Altium, Protel, MultiSim were all used to develop our PCB design. The knowledge of these programs will be very helpful for future projects in which we are involved.

If the project is continued, the team will have to make some decisions about the hardware they choose to implement. The current input/output devices are getting worn out and need to be replaced. We recommend placing a bread board inside the device enclosure so hardware is easily interchangeable by simply moving wires instead of soldering and de-soldering cables. This will greatly enhance the speed at which the embedded design team is able to troubleshoot misconnections and CPU to hardware I/O ports. We recommend switching to a board that is supported by both Keil and Labview. A design constraint that must be considered is the number of interruptible CPU I/O traces available. Many of the general purpose I/O traces are used by other devices on the Digi board. If interruptible GPIO is still a desired design constraint, the number of free interruptible traces must be considered. If a suitable board is not found that is supported by both Labview and Keil, a board must be chosen that is already
supported or easily portable to Keil’s compiler. Serious thought must occur before choosing to keep the existing LCD. The existing LCD’s interface is a 32-pin ribbon cable that will be hard to prototype with unless a new board is chosen that supports it. The existing LCD also requires a 9V power trace that was not provided on the Digi ConnectCore 9p 9360. Lastly if a new LCD is desired it is recommended that the interface is supported by the development board. If this is not possible, the LCD should be easily portable to Keil/Labview.

REFERENCES

LabView Embedded for ARM Porting Guide
http://zone.ni.com/devzone/cda/tut/p/id/6994

Digi: www.digi.com
Hatronix: www.hantronix.com
Hirose: www.hiroseusa.com
Mouser Electronic: www.mouser.com
National Semi-Conductors: www.national.com

APPENDIX A: ABBREVIATIONS

CAN Controller Area Network
CSU Colorado State University
ECU Engine control unit
FHEA Finite Heat Element Analysis
GPIO General Purpose Input/Output
GUI Graphic User Interface
I/O Input/Output
ISR Interrupt Service Routine
LCD Liquid Crystal Display
LED Light Emitting Diode
OS Operating System
PCB Printed Circuit Board
VGA Video Graphics Array
We were very fortunate to have a lot of the costs for this design project provided to us by last year’s group. They were able to buy the Digi board, LCD, and many of the hardware components. The table below is a list of the major expenses provided by last year’s team.

### Donated Items

<table>
<thead>
<tr>
<th>Materials</th>
<th>Serial Number</th>
<th>Base Cost</th>
<th>Total (w/ overhead)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Altera Cyclone II</td>
<td></td>
<td>$2000.00</td>
<td>$3000.00</td>
</tr>
<tr>
<td>ConnectCore 9F Net + OS</td>
<td></td>
<td>$499.99</td>
<td>$749.99</td>
</tr>
<tr>
<td>Hantronix 5.7” LCD</td>
<td></td>
<td>$180.00</td>
<td>$270.00</td>
</tr>
</tbody>
</table>

TABLE 4

Our expenses for this entire year did not exceed our budget. We had $300 dollars for the entire year and did not spend any of the money. If the project is continued, there will be some expenses for the team. These will include a new quadrature ($20), rotary switch ($10), and new device connectors. Depending on the specs of the project, a new board might also have to be bought to accommodate LabView.

We also were given virtual salaries from our advisor for the work that we completed. Saurav was the team leader and was given a salary of $70,000.

<table>
<thead>
<tr>
<th>Person</th>
<th>Salaries</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saurav Joshee</td>
<td>$70,000</td>
<td>80</td>
</tr>
<tr>
<td>Dean Kooiman</td>
<td>$65,000</td>
<td>85</td>
</tr>
<tr>
<td>Josh Gabler</td>
<td>$63,000</td>
<td>80</td>
</tr>
</tbody>
</table>

TABLE 5