5-2011

Software Verification for a Custom Instrument using VectorCAST and CodeSonar

Christina Dawn Ward
wardc2@usec.com

To the Graduate Council:

I am submitting herewith a thesis written by Christina Dawn Ward entitled "Software Verification for a Custom Instrument using VectorCAST and CodeSonar." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Qing Cao, Xiaorui Wang

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Software Verification for a Custom Instrument using VectorCAST and CodeSonar

A Thesis
Presented for The
Master of Science
Degree
The University of Tennessee, Knoxville

Christina Dawn Ward
May 2011

Dedication

For my father – Richard, my mother – Brenda, my sister – Lindsay, and my good friend – Carl. I could not have done this without you all. Thank you!

Acknowledgements

I am so lucky to be surrounded with the most wonderful, patient and inspiring people. I would like to thank my advisor, Dr. Peterson, for all of his advice and guidance during my time at UT. I appreciate Dr. Wang and Dr. Cao for serving on my committee and for all their valuable input. I would also like to thank my boss, Dwight Clayton, for encouraging me to obtain my Master's degree. Many thanks to all of my friends and coworkers for constantly checking in on my progress. A heartfelt appreciation to Carl – I could not have accomplished this degree without you. Thank you for being so supportive. Finally, my thanks and love to a family who means the world to me. I would not be where I am today without your unconditional love and support. I love you all!

Abstract

The goal of this thesis is to apply a structured verification process to a software package using a set of commercially available verification tools. The software package to be verified is adapted from a project that was developed to monitor an industrial machine at the Oak Ridge National Laboratory and includes two major subsystems. One subsystem, referred to as the Industrial Machine Monitoring Instrument (IMMI), connects to a machine and monitors operating parameters using common industrial sensors. A second subsystem, referred to as the Distributed Control System (DCS), interfaces between the IMMI and a personal computer, which provides a human machine interface using a hyperterminal. Both the IMMI and DCS are built around Freescale's MC9S12XDP microcontroller using CodeWarrior as the Integrated Development Environment (IDE). The software package subjected to the structured verification process includes the main C code with its header file and the code for its interrupt events for the IMMI as well as the main C code for the DCS and its interrupt events. The software package is exposed to the scrutiny of two verification tools, VectorCAST and CodeSonar. VectorCAST is used to execute test cases and provide results for code coverage based on statement and branch coverage. CodeSonar is used to identify issues with the code at compile time such as allocation/deallocation issues, unsafe functions, and language use problems. The results from both verification tools are evaluated and necessary changes made to the software package. The modified software is then tested again with VectorCAST and CodeSonar. The final verification step is downloading the modified code into the IMMI and DCS microcontrollers and testing the overall system to ensure the expected results are achieved with hardware that is developed to simulate realistic signals.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Motivation

Monitoring industrial machines is not a new concept. In fact, there are many instruments that have been developed over the years to monitor and control parameters such as flow rates, valve positions, temperatures, pressures, and vibration. For about three years, work has been done on a custom instrument that monitors and controls a specialized industrial machine. Ultimately, the instrument will be duplicated over a thousand times, placed on a local network and used to communicate with a central computer in a control room.

As part of the development of the instrument, an embedded microcontroller was programmed using CodeWarrior [11] as the integrated development environment (IDE). Naturally, problems were found during the initial development phase and repaired as necessary until a fully functional version of software was complete. The next step in the project was to build a prototype instrument that included supporting hardware circuits, FPGA firmware, and power supplies. Due to the complexity of the instrument, complete instrument testing required the development of a computer-based tester. Using the computer-based tester, hundreds of tests were performed on the new instrument and additional problems were discovered in the hardware and the microcontroller software that were not revealed during the initial development phase. Additional changes were made to the hardware and software until the instrument passed all the tests generated by the computer-based tester. At that point, the design was frozen and a few dozen prototype units were manufactured.

These prototypes or “beta” units were then deployed in the actual facility. During continuous facility testing, additional minor anomalies were detected that were not seen during the development phase or by the computer-based testing. Additional modifications were made as necessary to the instrument until facility testing was successful. Ultimately, the instrument operated as required and production of numerous units proceeded.

Looking back at the design process used during this project, it is evident that a formal method of software verification may have been a more efficient method of debugging the microcontroller code rather than the ad hoc method used. Also, some of the anomalies discovered during facility testing were extremely rare and it is possible that other rare problems may still exist, only to be discovered at a later date.

In an effort to assess the potential value of commercially available software verification tools, an evaluation version of VectorCAST [9] and CodeSonar [10] was used on an instrument, similar to that described above, specifically designed for this thesis. The instrument, referred to as an Industrial Machine Monitoring Instrument (IMMI), performs all the major functions of the actual instrument. A machine signal simulator was also developed and provides realistic signals for the IMMI to measure while software is being verified. Lastly, a Distributed Control System (DCS) was designed, fabricated, and programmed to provide an interface capability between the IMMI and a computer-based Human Machine Interface (HMI). Software developed for the DCS was

also subjected to the scrutiny of the VectorCAST and CodeSonar verification tools. A block diagram depicting the functional blocks of the project is shown in Figure 1.

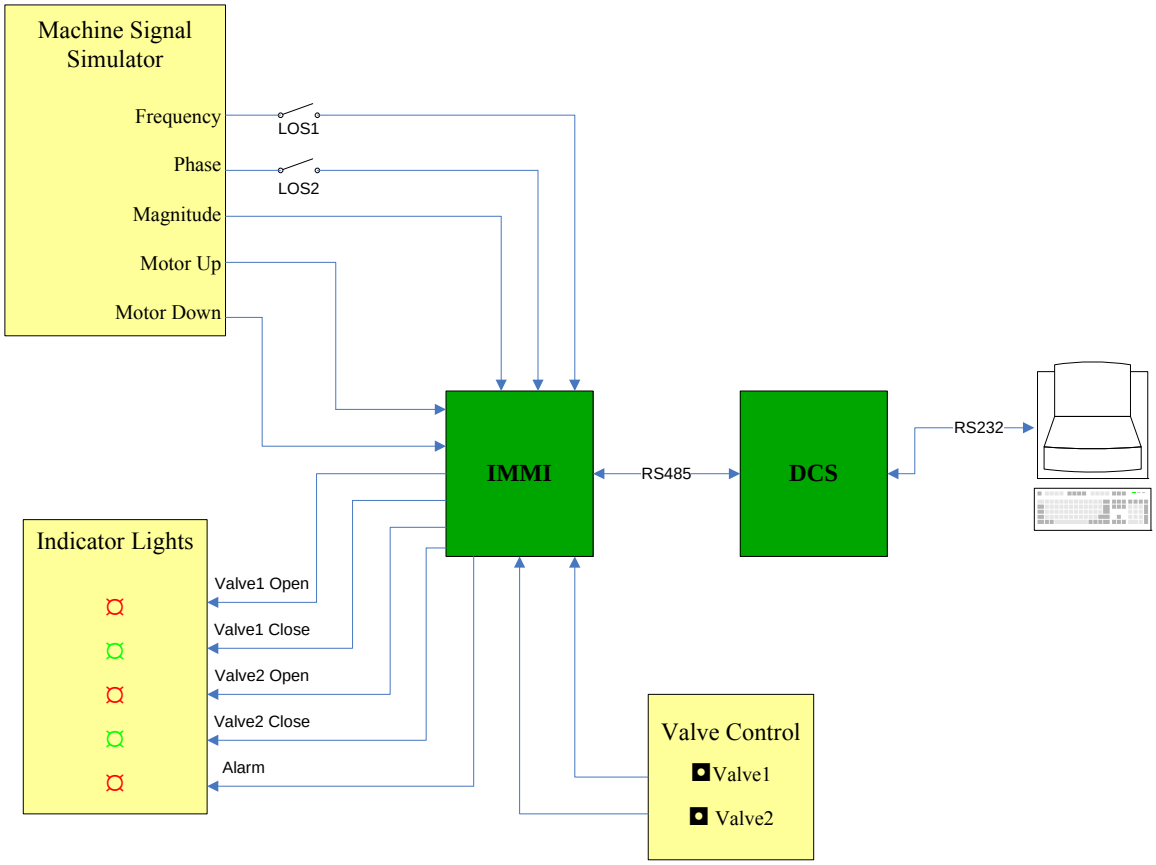


Figure 1: Overall Functional Block Diagram

Scope

The goal of this thesis is to expose a software package, consisting of C source code for the IMMI and DCS, to a set of verification tools in order to verify the software performs as expected. Hardware is created to produce signals that the IMMI would encounter based on the design requirements document, see Appendix A. These signals are used to help develop and verify the software for the IMMI as well as software for the DCS. The DCS is included so a user could observe the values reported by the IMMI and command the IMMI to perform certain tasks. The hardware designs and the software functions are discussed in Chapter 2. In Chapter 3, the methodology for verifying the software is discussed, which includes exposing the code to two different verification tools (VectorCAST and CodeSonar). The results from VectorCAST and CodeSonar are discussed in Chapter 4 as well as the changes made to both sets of software. The final version of the software is then tested using the hardware signals previously produced. The last chapter, Chapter 5, provides a summary of the work completed and areas where future work exists.

Chapter 2: Background

This chapter will discuss industrial applications the Industrial Machine Monitoring Instrument (IMMI) is targeting. It will also discuss hardware that was developed to simulate typical signals of these applications and the software functions to process that data.

Industrial Applications

There are a vast number of machines used in industry that incorporate some sort of rotating apparatus. Some machines may be based on old, well established designs and require little or no monitoring. Other machines may involve high speeds, close tolerances, and complex peripheral hardware. Examples of such machines include, but are not limited to, jet engines, turbomolecular pumps, gyroscopes, and high speed grinders. Table 1 shows the features of the industrial applications that can be monitored by the IMMI. For example, if the IMMI were monitoring a high speed grinder, the rotational speed could be measured for the main arbor. The rate of change in rotational speed and power consumption while coasting could indicate excessive load or depth of cut. A power consumption while coasting could also be a direct indication of bearing condition. A measurement of magnitude and phase of an accelerometer attached to the high speed grinder could be used to monitor the balance of the grind wheel and the measurements could be used to determine the level of wear. The valve control feature of the IMMI could be used to control the coolant flow. Finally, the position of the work piece could be tracked by monitoring the operation of the positioning motors.

Table 1: Industrial Application Features Potentially Monitored by the IMMI

Application	Rotational Speed	Rate of Change in Rotational Speed	Coasting Power Consumption	Magnitude & Phase	Valves	Auxiliary Motor (Direction and Time)
Jet Engines	Compressor shaft	Compressor drag; stall detection	Bearing drag; air flow	Turbine balance; compressor balance	Fuel flow	Turbo fan blade pitch control
Turbomolecular Pumps	Internal blade speed	Drag or blade failure	Excessive pressure	Bearing condition	Vacuum isolation	NA
Gyroscopes	Main spinning mass	Loss of drive power	Bearing condition	System balance; bearing condition	Vacuum isolation	Control valve for gas operated gyros
High Speed Grinders	Main arbor	Excessive load; depth of cut	Excessive load; depth of cut; bearing condition	Grind wheel condition or wear	Coolant flow	x-y-z table position

Jet engines are conceptually simple. They include rotating compressor blades at the front of the engine and rotating turbine blades at the rear of the engine [2]. Both sets of blades are mounted on a common shaft, and the region between the compressor and turbine forms a combustion area. As the compressor spins, it forces air into the combustion area where fuel is injected and the mixture is burned. Hot combustion gases exit the combustion area through the turbine blades, forcing the main engine shaft to turn, thus driving the compressor at the front of the engine. The combustion gases leaving the engine are traveling much faster than the air entering the engine. The acceleration of the air/exhaust times its mass results in force or thrust as per equation (1)

$$F = M \times A \tag{1}$$

where F=force, M=mass of air, and A=acceleration.

While the engine is simple conceptually, the overall hardware is extremely sophisticated. Close mechanical tolerances must be maintained between the spinning blades and stationary housing while balance and temperature profiles are maintained. Engine subsystems like the fuel pumps may be driven from the main engine shaft at a lower speed and may require phase angle monitoring relative to the main shaft. Therefore, some of the capabilities of the IMMI presented here may be directly useful in monitoring a jet engine, at least during its development phase.

Turbomolecular pumps are used in high-vacuum systems and are typically placed between a vacuum vessel and a standard (backing/roughing) vacuum pump. They operate much like a high speed fan and move gas molecules towards the outlet of the pump as the blades collide with the residual gas molecules in the pump. Turbomolecular pump operation is based on the concept

that gas molecules can be moved in a preferred direction by striking them with a solid surface (in this case, spinning blades within the pump) [4]. As gas molecules are hit by the spinning blades, they gain momentum that carries them to the outlet of the pump, where they are captured with a backing pump. The blades in the turbo pump are driven by an electric motor at speeds up to 90,000 revolutions per minute. If a keyphasor signal and an accelerometer were attached to the turbomolecular pump, the IMMI could be used to measure speed, vibration, and shifts in the angle of mechanical run out with respect to the keyphasor. In the event of an abrupt pump failure, the IMMI could quickly detect the fault and close valves between the turbomolecular pump and vacuum vessel. This would protect the vacuum vessel from contamination migrating from the turbomolecular pump.

Classic gyroscopes are used in a wide variety of applications, including aircraft autopilots, compasses, inertial guidance systems, and motion stabilizing platforms used for cameras. The most common type of gyroscope contains a spinning wheel or mass, which possesses a significant degree of angular momentum. Typically, mass is kept in motion by some manner of motor and the speed is often high so a large amount of angular momentum can be realized using a relatively small amount of mass.

Sophisticated gyroscopes used in critical applications could use some or all of the features designed into the IMMI. Speed, vibration, and angle of run-out would all be direct indicators of gyroscope health. If a gyroscope were driven by a gas turbine, gas flow might be adjusted with

motors, which the IMMI is designed to monitor. If the gyroscope were to fail, or simply operate abnormally, the IMMI could issue the appropriate alarm and shut the unit down.

High speed industrial grinders are used to finish the surface of cast or machined metal parts and are often used in automated applications. The main grinder motor typically turns the abrasive grind wheel at a high rate of speed, resulting in a very smooth finished surface. This type of industrial equipment could benefit from being monitored by the IMMI. The balance of the grinding wheel is a critical parameter that could be measured by the IMMI. The relative phase of the grind wheel imbalance could indicate the condition of the wheel. A motorized shuttle table to maneuver the work piece under the grind wheel could be monitored by the IMMI as it translates into various positions, right and left or up and down. In the event of grinder malfunction or excessive grind wheel wear, the IMMI could report the condition to a central production computer and place the grinder in a safe condition, which includes turning off the coolant flow valves. Excessive grinder load resulting from a misplaced work-piece could also be detected by the IMMI's rate of change detector, and a protective response could be initiated.

Overview of Requirements

Systems like jet engines, turbomolecular pumps, gyroscopes, and high speed grinders all require monitoring systems to assess the health of the equipment and to protect personnel involved. Requirements for such monitoring systems would most likely include frequency, magnitude, and phase, direction and operation time of a motor, control of valves using switches, and alarms produced for abnormal conditions. The Industrial Machine Monitoring Instrument (IMMI) was

designed to meet similar requirements. The requirements the IMMI must meet are described in the Design Requirements Document given in A.

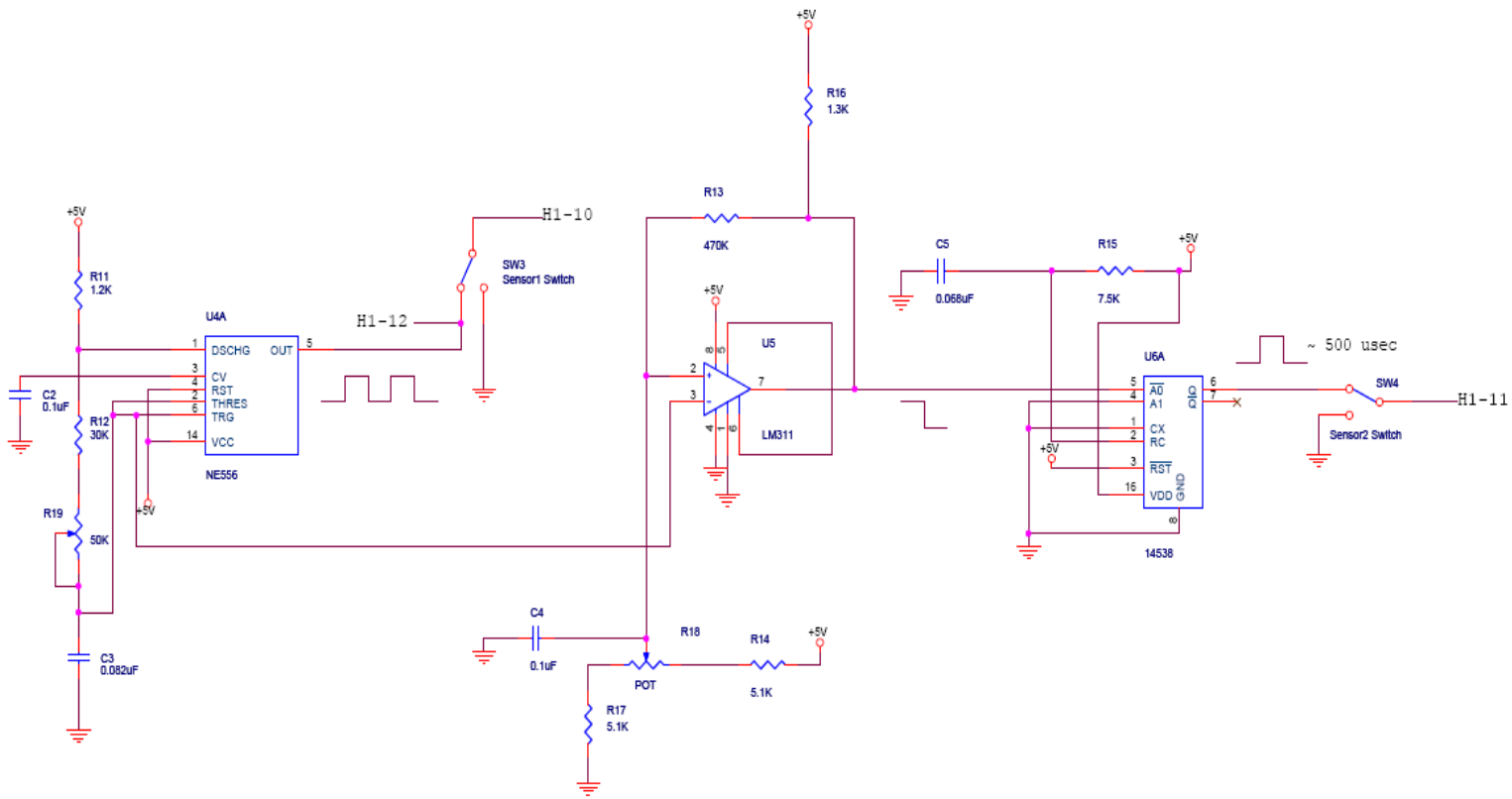
Hardware Produced

Hardware was created to simulate the kind of signals one would expect to analyze from jet engine, turbomolecular pump, gyroscope, or high speed grinder sensors. The schematics are provided in each section with a description of how they function. There are six separate hardware pieces that are needed to verify the functionality of the IMMI and DCS.

Sensor1 Frequency and Sensor1 to Sensor2 Phase Control

The IMMI is required to monitor the rotational frequency of a machine and the phase angle between a keyphasor and an additional sensor such as an accelerometer. To test the IMMI, signals are generated to simulate a selectable frequency and a phase shifted signal at a controllable angle relative to the keyphasor signal.

The circuit selected to accomplish the above requirements, see Figure 2, is based on a simple timer, U4A, (half of a NE556 dual timer) connected as an astable multivibrator. In that configuration, a timing capacitor, C3, is allowed to charge up to 2/3 Vcc through R11, R12, and R19. When the voltage on capacitor C3 reaches 2/3 Vcc, the NE556 timer switches to the discharge mode and capacitor C3 is discharged through R12 and R19. When the voltage across C3 gets down to 1/3 Vcc, the NE556 timer stops discharging and it is again allowed to charge up to 2/3 Vcc. As capacitor C3 is charging, the output of the NE556 timer, pin 5, is high and while capacitor C3 is discharging, the output of the NE556 timer is low.



Note: H1 stands for Header 1 on the Technological Arts Adapt9S12XD boards. The number listed after H1 is the pin number the signal should be connected to on the header.

Figure 2: Sensor1 Frequency and Sensor1/Sensor2 Phase Control Schematic

The high and low output of the NE556 timer simulates a keyphasor of known frequency for the IMMI. The output frequency of the NE556 timer can easily be adjusted by varying the resistance of R19 to produce a frequency per equation (2).

$$\text{Frequency} = \frac{1.44}{(R11 + 2(R12 + R19))C3} \tag{2}$$

For the NE556 timing circuit used, the timing capacitor voltage ramps up to 2/3 Vcc and down to 1/3 Vcc synchronously with the output signal, no matter what the operating frequency. A stable phase delay can, therefore, be generated by triggering a monostable multivibrator, U6A, based on the analog voltage applied to the timing capacitor, C3. To do this, an LM311 comparator, U5, is used to monitor the voltage of the NE556 timing capacitor, C3. When the timing capacitor voltage reaches a level selected by the potentiometer, R18, the output of the LM311 comparator drops low, which triggers a 14538 monostable multivibrator, U6A. Feedback around the LM311 comparator is provided by R13 and ensures oscillation free operation of the comparator. When the NE556 timing capacitor discharges through the comparator set point, the output of the comparator rises but does not retrigger U6A. The result is a positive pulse from U6A at a selectable phase angle relative to the keyphasor. The pulse width is independent of phase angle and determined by the components used with U6A as defined in equation (3).

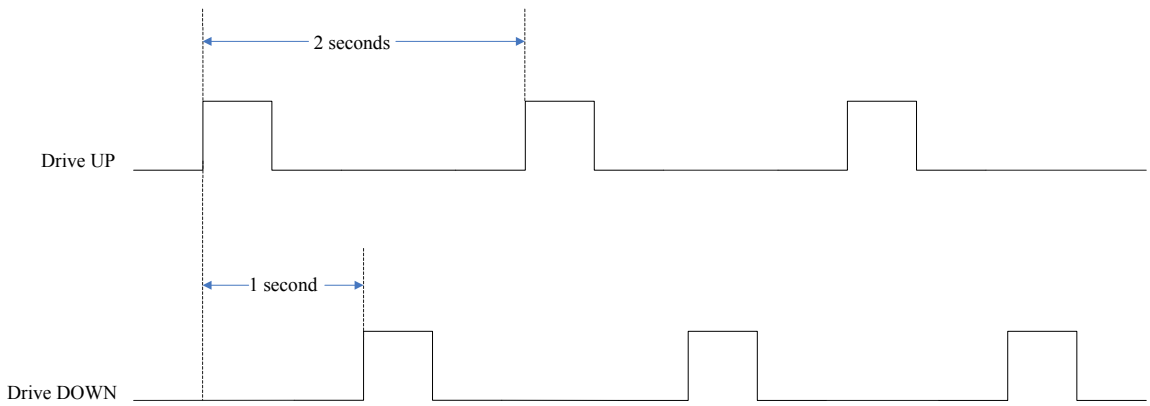
$$\text{Pulse Time} = R15 \times C5 \tag{3}$$

The IMMI includes fault detection circuits that must be exercised during testing. To aid in this process, switches SW3 and SW4, are incorporated in the output signals of the circuits described above. They allow a convenient method of removing the signals from the IMMI as necessary. It is noted that no switch debounce function is necessary on these signals because extra transitions here will not cause an erroneous response.

Motor Up and Down Simulation

Another requirement of the IMMI is to monitor adjustments made by a motor on a machine as it compensates for changing operating conditions. It is assumed the motor system includes an electronic controller that provides two digital outputs. One output is active high when the motor moves clockwise and a second output is active high when the motor moves counter-clockwise. Clockwise and counter-clockwise motor rotation translates into physical movement (Up/Down or Right/Left) depending upon the specific mechanism used. The IMMI is designed to measure the length of time and direction the motor operates while continuously reporting this information to the DCS.

A test circuit designed to simulate the motor system produces two alternating output signals spaced approximately one second apart. The pulse width from each output slowly increases in width over a 10 second period, and then slowly decreases in width for the next 10 seconds. The minimum pulse width is approximately 200 milliseconds while the maximum pulse width is approximately 500 milliseconds. A timing diagram of these two test signals is shown in Figure 3.



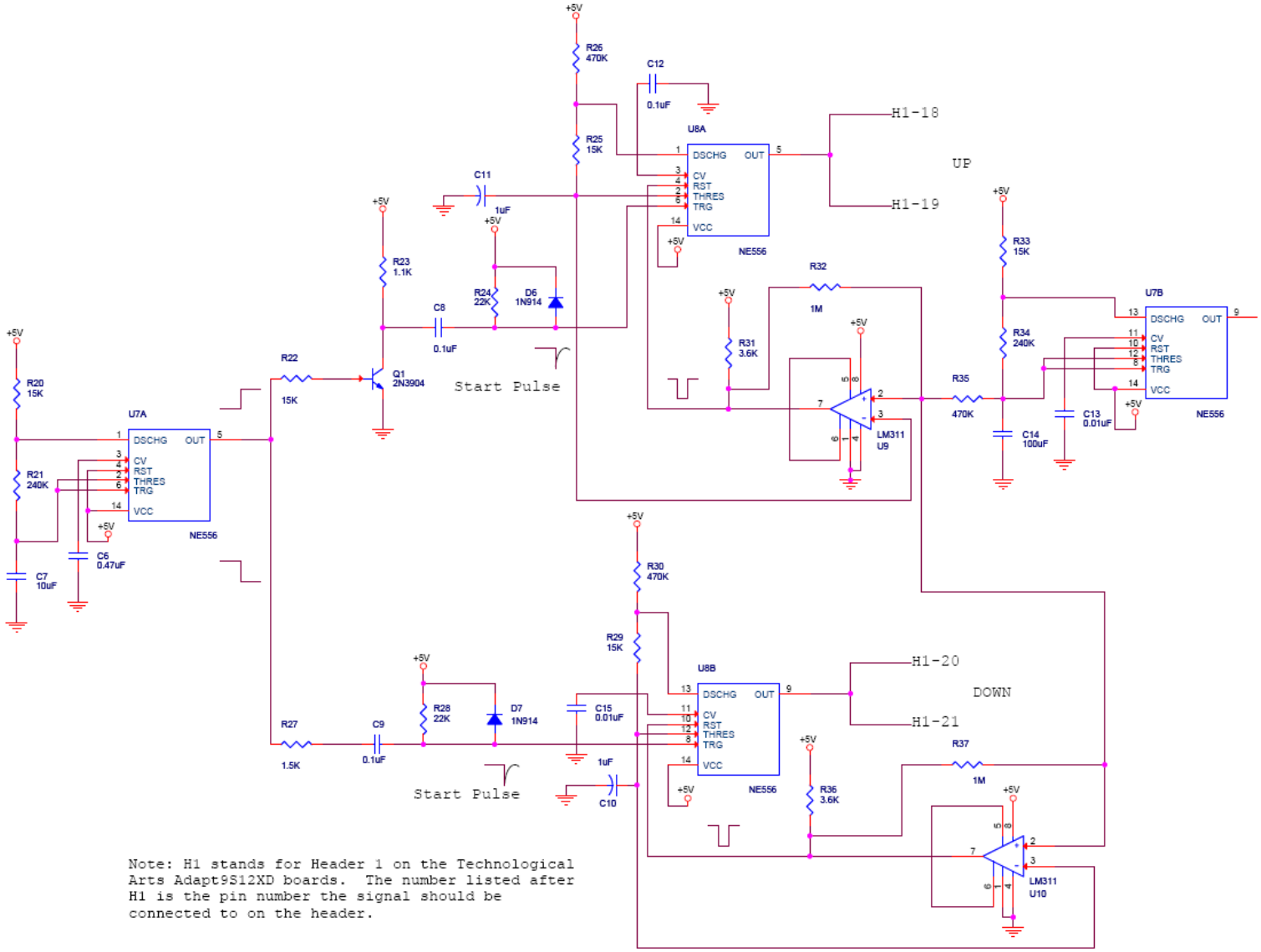
Note: Pulse width for drive UP and drive DOWN varies between 200 ms and 500 ms over a 20 second period.

Figure 3: Motor Up and Down Timing Diagram

Circuitry used to develop the test signals includes a time base, a ramp generator, UP and DOWN pulse generators, and two pulse generator reset circuits. A schematic of the test signal generator is shown in Figure 4.

The time base used to trigger the two system outputs is one half of a NE556 timer, U7A, connected as an astable multivibrator. Timing resistors R20 and R21 and timing capacitor C7 were selected to produce a waveform with a 2 second period and a 50% duty cycle. This results in fairly even spacing between the two output pulses and arises from the ratio between R20 and R21. Combined, R20 and R21, form a 255K ohm path for capacitor C7 to charge up to 2/3 Vcc, while R20 provides a 240K ohm path for C7 to discharge to 1/3 Vcc. The rising edge of the time base output is used to initiate the UP motor pulse output and the falling edge of the time base is used to initiate the DOWN motor pulse output.

Another NE556 dual timer, U8, is used to produce the two basic output pulses from this circuit. One is triggered on the rising pulse from the time base, U8A, and the other is triggered from the falling edge of the time base, U8B. To trigger U8A on the rising edge of the time base, Q1 is used to invert the positive output edge from U7A. As that output rises, Q1 is biased on and pulls one side of C8 low. This forces the trigger input of U8A to momentarily go low and a timing cycle for the UP motor simulation is started. When the output from the time base U7A falls, a trigger pulse is developed as C9 charges through R27, thus triggering U8B and starting a timing cycle for the DOWN motor simulation. Clamp circuits are used at both trigger inputs of U8 to prevent excessive voltage when the trigger pulses go back to a high level.



Note: H1 stands for Header 1 on the Technological Arts Adapt9S12XD boards. The number listed after H1 is the pin number the signal should be connected to on the header.

Figure 4: Motor UP and DOWN Simulator Schematic

The pulse widths from the UP and DOWN pulse generators must gradually change so that the IMMI can demonstrate active pulse width measurement. The basis for the varying pulse width is a ramp generator built around U7B. Connected as another astable multivibrator, U7B has a relatively long period (20 seconds) developed by R33, R34, and C14. The ratio of resistance between R33 and R34 yields a near 50% duty cycle at the output of U7B. The usable signal from the ramp generator is the voltage developed across C14 as it slowly charges up to 2/3 Vcc and discharges down to 1/3 Vcc. This varying voltage is used to determine the level of charge allowed on the timing capacitors, C10 and C11, in the UP and DOWN pulse generators before they are reset.

Using two voltage comparators, U9 and U10, the voltage on each pulse generator timing capacitor is compared to the ramp generator voltage produced by U7B. When either pulse generator timing capacitor voltage reaches a level equal to the ramp generator voltage, a reset pulse is developed by the associated comparator and the timing pulse is terminated. In the unlikely event of the ramp generator voltage being higher than 2/3 Vcc, the pulse will simply reset itself when its timing capacitor reaches 2/3 Vcc. The effect of the ramp generator and comparators is to prematurely terminate the output pulse of U8A or U8B based on the changing voltage developed by the ramp generator, U7B.

Resistors R32 and R37 are used to provide hysteresis for the voltage comparators U9 and U10, thus avoiding the possibility for oscillations at the comparator outputs.

Pushbutton Signal Conditioning

This instrument, like many others, requires some input from a user, which may take the form of a switch or pushbutton where electrical contacts can be monitored with a circuit. When a set of pushbutton contacts is monitored by a high-speed, high-input-impedance device such as a microcontroller, a pull-up resistor is typically used to bias the input high and the pushbutton contacts are used to pull the input low.

A mechanical contact bounce is usually experienced when pushbutton contacts open or close. This causes the signal from the pushbutton to transition from high to low multiple times as the switch is moved a single time. Often times the signal bounce from the pushbutton causes undesirable circuit behavior. One such example is when the pushbutton output is used to generate an interrupt signal to the microcontroller. If signal bounce is experienced there, multiple interrupts will be generated, resulting in abnormal system response.

There are a couple of techniques available to solve the pushbutton bounce problem in high speed electronic circuits. The simplest is to filter the pushbutton output with a resistor/capacitor (RC) network with a time constant long enough to mask the bounce phenomenon from the pushbutton. If this technique is used, the output of the RC network must have a relatively slow rise and fall time and may have to be enhanced with a Schmitt trigger [12]. Another approach available to debounce a pushbutton (the method used for this thesis) is to use two cross coupled NAND gates configured as a set/reset (RS) flip-flop as shown in Figure 5.

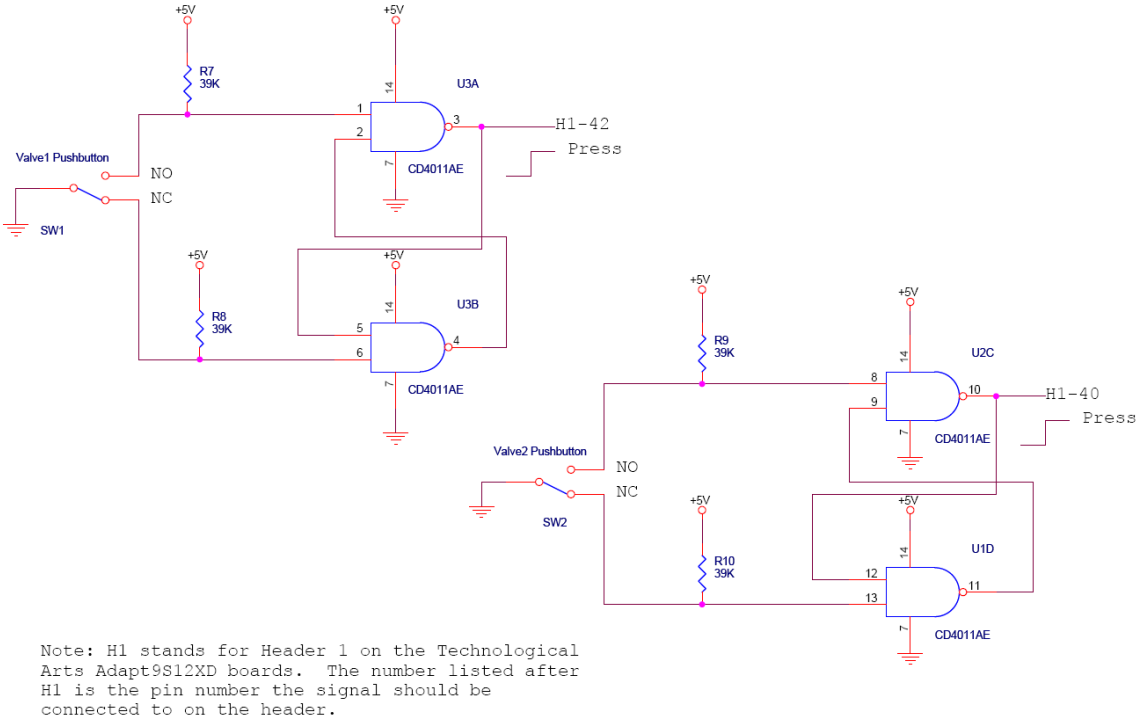


Figure 5: Pushbutton Signal Conditioning Schematic

In this circuit, two NAND gates are connected as a RS flip flop. The two active inputs at pin 1 of U3A and pin 6 of U3B are pulled up with 39K ohm resistors and connected to the normally open (NO) and normally closed (NC) contacts of the pushbutton. The common contact of the pushbutton is connected to ground. Before the pushbutton is pressed, the NC contact of the pushbutton forces a low state at pin 6 of U3B, the lower NAND gate, forcing its output at pin 4 of U3B to go high. This high state is applied to pin 2 of U3A, the upper NAND gate. The other input to the upper NAND gate, pin 1 of U3A, is pulled high by a 39K ohm resistor, forcing the output at pin 3 of U3A to the low state. The output from pin 3 of U3A is also cross-coupled to pin 5 of U3B, the lower NAND gate, which reinforces the high state seen on pin 4 of U3B.

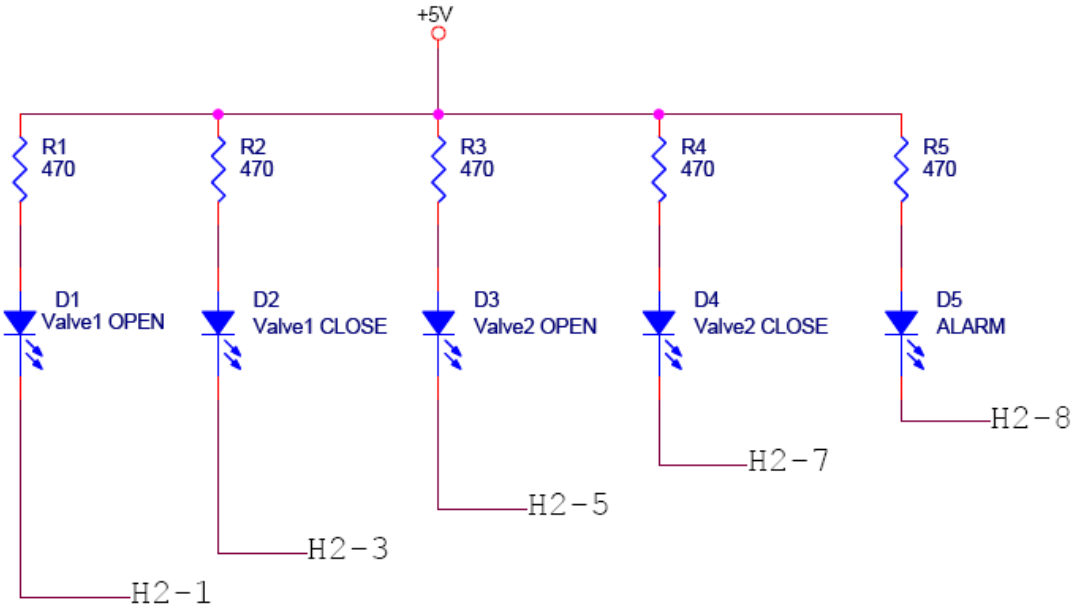
When the Valve1 pushbutton is first pressed, the wiper within the pushbutton (common contact) leaves the NC contact and the pull-up resistor on pin 6 of U3B pulls the line high. This allows pin 6 of U3B, the lower NAND gate, to go high. If the wiper in the pushbutton does not break its contact with the NC terminal cleanly, multiple high/low states will be seen on pin 6 of U3B, the lower NAND gate. This will not cause any change in state of the overall RS flip flop because the low level output line from the upper NAND gate is already forcing the output of the low level NAND gate high. Therefore, any transitions at pin 6 of U3B are ignored while the wiper in the pushbutton travels from the NC contact to the NO contact.

As the wiper in the pushbutton hits the NO contact, there is a very good chance of mechanical bounce, which results in a series of high and low states to the input of the upper NAND gate at pin 1 of U3A. However, the first time the wiper in the pushbutton touches the NO contact, a low

level is applied to pin 1 of U3A and the output at pin 3 of U3A is forced high. This output signal is cross coupled to the input of the lower NAND gate at pin 5 of U3B. By then, the input at pin 6 of U3B is high and stable because the pushbutton wiper is all the way over at the NO position. With both inputs of the lower NAND gate high, the output at pin 4 of U4B must be low. This output signal is cross coupled to the input at pin 2 of U3A and reinforces the low level seen at pin 1 of U3A during the first contact between the wiper and the NO terminal. As the pushbutton is released, the reverse process takes place. The circuit is duplicated for the Valve2 pushbutton.

LED Indicators

Light Emitting Diode (LED) indicators are used by the IMMI to show the state of various outputs. Normally, output ports from electronic devices such as a microcontroller have a greater ability to sink current than to source current. This is particularly the case if the output driver is configured as an open collector or open drain device. To take advantage of this type drive device, the LEDs (to be used for valve positions and an alarm indicator) were connected as shown in Figure 6. The 470 ohm resistors (R1, R2, R3, R4, and R5) provide current limiting for any given LED, which is illuminated as its cathode is pulled low by the output drive device.

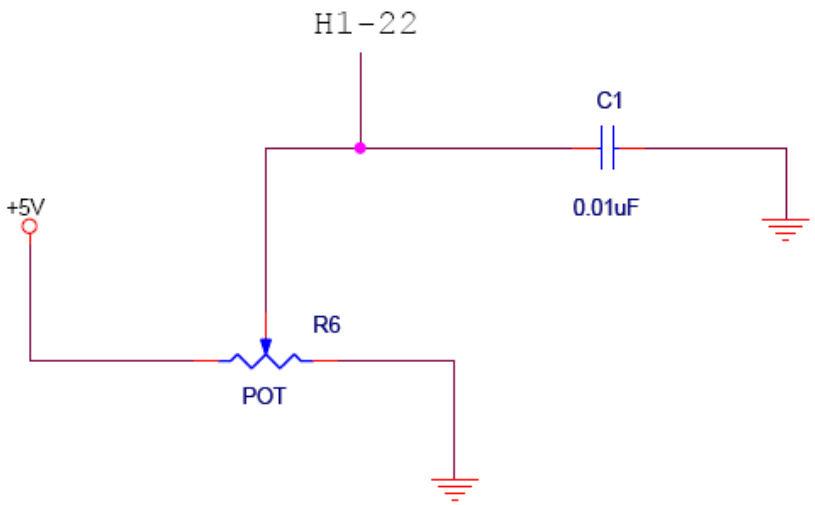


Note: H2 stands for Header 2 on the Technological Arts Adapt9S12XD boards. The number listed after H2 is the pin number the signal should be connected to on the header.

Figure 6: LED Indicators Schematic

Sensor2 Magnitude

An analog voltage must be provided to the IMMI to simulate a variable output level from Sensor2. This would emulate amplitude from a sensor such as an accelerometer. A simple potentiometer, R6, is used as a voltage divider between 5Vdc and ground as in Figure 7. Capacitor C1 provides some filtering of the analog voltage as the wiper in the potentiometer is moved.



Note: H1 stands for Header 1 on the Technological Arts Adapt9S12XD boards. The number listed after H1 is the pin number the signal should be connected to on the header.

Figure 7: Sensor2 Magnitude Schematic

Microcontroller Boards

The microcontroller that is selected for use in the IMMI is Freescale's MC9S12XDP512 microcontroller [7], due to previous experience with this particular component. Two prototype microcontroller boards (Adapt9S12XD) were purchased from Technological Arts to demonstrate a fully functional IMMI and DCS. The prototype boards consist of the microcontroller along with a RS232 port and screw terminals for a RS485 communications interface [6]. The schematic for Technological Arts' Adapt9s12XD can be found online [6]. Another feature of the prototype boards is the easy access to all the I/O from two 50 pin headers [6]. One prototype board will represent the IMMI and the other board will represent the DCS.

Software Developed for the IMMI

CodeWarrior IDE

The integrated development environment (IDE) chosen to develop the software for the IMMI and DCS was Freescale's CodeWarrior IDE [11]. An extension that was added to CodeWarrior was Processor Expert, which allows designers the ability to accelerate their design time by using modular, reusable, and fully tested functions [13]. Processor Expert provides configurable components they call *Embedded Beans* for a variety of functions the microcontroller is capable of handling [13]. For instance, there is a bean for serial communications interface (SCI) that can be setup to handle RS232 or RS485 communications. Other *Embedded Beans* that Processor Expert provides are timers, external interrupts, general I/O, analog to digital converter processing, and capture timer [13]. It is up to the user to configure the bean properly. Processor Expert then generates all the initialization code needed for the *Embedded Bean* and the user must provide code to handle the response to interrupts generated from the simulation signals.

MC9S12XDP512 Microcontroller

Freescale's MC9S12XDP512 microcontroller used for this project is derived from Motorola's M68HC11. The S12X core is a "high speed, 16-bit processing unit that has a programming model identical to that of the industry standard M68HC11 central processing unit (CPU)" [8]. The instruction set is also a "superset of M68HC11's instruction set" [8]. The MC9S12XDP512 microcontroller features that will be utilized consist of "standard on-chip peripherals including up to 512Kbytes of Flash EEPROM, 32Kbytes of RAM, six asynchronous serial communications interfaces (SCI), three serial peripheral interfaces (SPI), an 8-channel Input

Capture (IC)/Output Capture (OC) enhanced capture timer, a 16-channel, 10-bit analog-to-digital converter (ADC), and a periodic interrupt timer" [7].

The code written for the IMMI takes advantage of the periodic interrupt timer set at 1ms for all timers within the code. Three enhanced capture timer channels are used to capture the count value of the free running clock (2^{16}) to calculate frequency and to determine a phase between two sensors. One SCI is used to communicate with the DCS. Six external interrupts, eight general I/O, and one ADC channel are also used.

The code for the DCS takes advantage of the periodic timer also set at 1ms for use within the code. Two SCIs are used: one for communications with the IMMI and one for communications with the HMI. The only code left utilizes two general I/O pins.

Frequency Measurement

The frequency of sensor1 is calculated and used in the sensor1 rate of change measurement as well as in the decel product alarm calculation. The sensor1 frequency measurement is made using captured values from the microcontroller's free running counter along with the number of times the free running counter overflows. The free running counter is a 2^{16} bit counter (maximum value of 65535).

When a measurement is ready to be made all variables are reset except for the free running counter. An interrupt then occurs for each rising edge of the sensor1 signal. The interrupt routine captures the free running counter value at the first sensor1 signal (icapture1) and verifies

it is not close to an overflow (must be less than 65000). Being too close to an overflow can produce a bad frequency measurement, especially since the MC9S12XDP512 microcontroller does not have nested interrupts. If the value is accepted, the overflow variable is cleared and is incremented each time the counter starts over (overflow). The interrupt handling routine looks for six overflows of the counter. The reason for using six overflows is to ensure a measurement can be finished within one second. Mathematically speaking, an overflow of the counter occurs once every 128 milliseconds, which is based on the 512 KHz clock. Therefore, six overflows of the counter takes 768 milliseconds. A variable for the number of periods (sensor pulses) is also kept. After six overflows or at least two sensor1 periods, the final free running counter value is captured (icapture2) as well as the total overflows (exactoverflows) and total number of periods (exactperiods). A flag is then set to allow the frequency measurement to be calculated.

In the main program, the frequency measurement is calculated by first determining the count value as shown in equation (4).

$$\text{Count} = (exactoverflows \times 2^{16}) + (icapture2 - icapture1) \tag{4}$$

The count value is then used in the final equation that produces the actual frequency value (see equation (5).

$$\text{Frequency} = \frac{512KHz \times (exactperiods - 1)}{Count} \tag{5}$$

The resulting frequency value is then reported to the DCS as parameter 3. A flag is also set for use in the sensor1 rate of change and decel product alarm calculations, which is merely an indication that a new frequency measurement has completed.

Phase Measurement

The phase angle between sensor1 and sensor2 needs to be measured and presented in polar form, between 0° and 360°. In order to provide a stable reading, a numeric average of sixteen readings is desirable. Normally this would be a simple task except for the case where individual measurements straddle the boundary between 360° and 0°. At this boundary, a simple numeric average may give an erroneous result. For example, if eight measurements of 359° were averaged together with eight measurements of 1°, the resulting average would be an erroneous 180°. To avoid this anomaly, basic angular measurements are taken over 720°, thus eliminating the discontinuity at 360°.

To perform a phase angle measurement, sensor1 and sensor2 are first fed into two enhanced capture timer I/O inputs. Then the value of the free running counter is captured when a measurement is ready to be made and an interrupt is detected from sensor1. The captured value (icapture3) must be less than 65000 so that the final count of the free running counter does not come close to overflowing at 65535. If icapture3 is greater than 65000 the measurement must start over. If icapture3 is valid then the free running counter is captured a second time when an interrupt occurs from sensor2 (icapture4). Finally, as the third sensor1 interrupt is detected, the final free running counter value is captured (icapture5) representing 720°.

In the main program, the icapture values are used to calculate a Sensor1Count (icapture5-icapture3) and a Sensor2Count (icapture4-icapture3). Sensor1Count represents a full 720° period of sensor1 and Sensor2Count represents the fraction of a sensor1 period that passed before sensor2 was detected. If sensor2 did not produce an interrupt between the two sensor1 interrupts then the Sensor2Count will be set equal to the Sensor1Count. A ratio (fraction) between the counts is then obtained as shown in equation (6) and represents the angular placement between sensor2 with respect to sensor1 relative to 720°.

$$\text{Sensor2Sensor1\_Fraction} = \frac{Sensor2Count}{Sensor1Count} \tag{6}$$

Each time a phase angle is measured between sensor1 and sensor2, the resulting fraction is calculated and stored in one of five bins, determined by the magnitude of the fraction. Bin1 ranges from 0.0 to 0.125, Bin2 from 0.125 to 0.250, Bin3 from 0.250 to 0.375, Bin4 from 0.375 to 0.500 and Bin5 from 0.500 to 0.625. These bins represent the first five quadrants of the eight possible quadrants contained in the 720° measurement period. If the signals from sensor1 and sensor2 are reasonably stable, all sixteen fractional data points will be placed in a single bin or adjacent bins. Any fractional data placed in Bin1 is copied to Bin5 after adding 0.500. This allows the phase angle averaging algorithms to properly analyze data spanning Bin4 and Bin5.

The summation of samples in all sets of adjacent bins is then analyzed and the largest number of samples in any two adjacent bins is then used to determine phase. Not all the samples may be in two adjacent bins so only the actual number of samples in those bins are considered. This means the total number of sample values may no longer be 16 but it must be at least 4.

The phase equation is shown in equation (7).

$$\text{Sensor1\_Sensor2\_Degree} = TotalFraction * \left(\frac{720.0}{SampleCounter} \right) \tag{7}$$

A traditional polar plot spans a range of 0° to 360° but does not include 0.00°. So if the whole number of the measured angle is 0° and if the decimal value is less than 0.05° then the indicated reading will be 360°. If the decimal value of the measured angle is greater than 0.05° then the indicated reading will be 0.1°. If the sample value is less than 4, phase measurement data is scattered between too many quadrants and a reliable phase measurement is not possible. In that case, Sensor1_Sensor2_Degree is reported as 0°. Since 0° is not a normal value on a polar plot, it is a perfect way to indicate a problem with the calculation. Otherwise, the reported value to the DCS is equal to Sensor1_Sensor2_Degree. After the value has been assigned to the variable sent to the DCS for the phase, the Bin Sum, Bin Counter, and Sensor2Sensor1_Fraction variables are cleared. The sample_counter_fixed value is reassigned to sixteen.

The remaining section of code in the PhaseMeasure() function includes a timeout section. If a measurement is not made within 8 seconds then all the pertinent variables to restart a calculation are cleared.

Sensor1 Rate of Change

To implement the sensor1 rate of change and decel product alarm functions, the IMMI utilizes one 60 element array of sensor1 frequencies taken at intervals slightly longer than 1 second. The sensor1 rate of change is determined by subtracting the oldest machine speed stored in array element 59 from the most recent machine speed stored in array element 1, see Figure 8.

The resulting difference in sensor1 frequencies has to be multiplied by 1.0060 to account for the actual time the frequency values fill the array. The units for rate of change are RPS/minute. A new sensor1 frequency is entered in array element 0 and all previous sensor1 frequencies are shifted to the next higher array element.

60 Element Array

0
1
2
3
4
5
6
7

.
.
.

53
54
55
56
57
58
59

$$\text{Decel Product} = \frac{ArrayElement[11] - ArrayElement[1]}{10.24\,\text{sec}} \times ArrayElement[1]$$

$$\text{RateofChange} = (ArrayElement[0] - ArrayElement[59]) \times 1.0060$$

Figure 8: Decel Product and Rate of Change Calculations

Motor Movement Interval Time and Direction

The motor that will be monitored is expected to provide two signals. One signal will represent the motor moving in an upward direction and the other signal will represent the motor moving in a downward direction. The motor could also move in a direction from left to right, which case the method being described still holds true. Both signals will indicate movement when they transition from zero voltage to a positive voltage and positive voltage to zero voltage when movement has ceased.

The method used to measure the amount of time the motor is operational will require both signals to be duplicated. Therefore, two external interrupt pins will be used for the up motion and two external interrupt pins will be used for the down motion. One external interrupt is programmed to look for a rising edge signal and the other external interrupt is programmed to look for a falling edge signal. When the motor moves in either direction, the rising edge triggered interrupt occurs and a timer (uptimer or downtimer) is started. The timer continues to increment until the falling edge interrupt occurs and stops the timer. A flag (up or down) is then set so the main motor movement function can finish the calculation (divide the timer value by 1000 since the timer value is in milliseconds and to make the value negative if the motor moved down) and update the motor movement variable (parameter 9 – upSec or downSec) that is sent to the DCS. The direction of the motor movement is then reported after the interval time is reported. The up and down flags are cleared so old values are not reported again.

Alarms

The alarm LED will be illuminated anytime an alarm is present. If failure detection scenarios (loss of Sensor1 and loss of Sensor2 OR loss of Sensor2 and Decel Product alarm) are present, the alarm LED will flash at a rate of 200ms. The alarm LED will not be illuminated under normal operation conditions.

Loss of Sensor1

Sensor1 is used to make a frequency measurement explained in section 2.4 under the heading "Frequency Measurement". Within the interrupt handling code of the frequency measurement, a timer is set for 100 milliseconds (LOS1_Timer). Each time an interrupt occurs the timer will be reset. The timer is decremented in the timer interrupt handling code and the loss of sensor1 is evaluated. Sensor1 is determined to be missing if the timer equals zero and is not already missing (this statement is to ensure the valves don't continue to close after a manual de-isolation of a singular alarm). The valves are then isolated based on the loss of sensor1 and the LOS1 alarm is sent to the DCS (parameter 6). Sensor1 is reported normal when the signal returns and the LOS1_Timer is greater than zero.

Loss of Sensor2

Sensor2 is used in the phase measurement explained in an earlier section. Within the interrupt handling code of the sensor2 phase measurement, a timer is set for 100 milliseconds (LOS2_Timer). Each time an interrupt occurs the timer will be reset. The timer is decremented in the timer interrupt handling code and the loss of sensor2 is evaluated. Sensor2 is determined missing if the timer equals zero and is not already missing (this statement is to ensure the valves don't continue to close after a manual de-isolation of a singular alarm). The valves are then

isolated based on the loss of sensor2 and the LOS2 alarm is sent to the DCS (parameter 8). Sensor2 is reported normal when the signal returns and the LOS2_Timer is greater than zero.

Decel Product

The decel product alarm is determined by using the rate of change frequency array elements 1 through 11. The frequency value in element1 is subtracted from the value in element 11 and the value is divided by 10.24 to provide a deceleration rate over 10 seconds. However, each frequency value is not placed in the rate of change array every second, thus 10.24 is used. This calculation produces a value that is then multiplied by the current frequency, which results in the decel product value. The decel product value is compared to a rate of 50 RPS^2/second. If the decel product value is $\geq$ 50 RPS^2/second, the decel product alarm closes the valves and is reported to the DCS as parameter 7 (Decelalarm). The alarm will continue to be asserted for 7.5 seconds. The remainder of the decel product alarm code deals with not allowing the valves to continue closing after the first closure of the valves so that a manual de-isolation may occur.

Communications to DCS

Communications with the DCS is required to be a half-duplex RS485 communications network with a custom protocol. The baud rate is set to 9600 baud with no parity, 8 bits, and 1 stop bit.

Receive

As a packet is transmitted to the IMMI from the DCS, characters are handled individually by the interrupt handling code. The first two characters are the start characters, which should be a 2. If the first character is a 2 then a flag is set (STX) and if the second character is a 2 then the character counter is cleared and another flag is set (ACPT). If the start characters are not received then the packet will not be analyzed. The other characters are interrogated to ensure

they are valid characters. Valid characters include numbers 0-9, a space, negative sign, and a decimal point. Once determined to be valid, the character is then placed in the receive buffer. If the character is not valid then the packet is no longer handled. The end of file (EOF) character is a 3. When a 3 is received, the receive buffer counter is checked to make sure data was actually received. If there is data present then the end of file character is replaced with zero and the communications stream can be interrogated further by setting a flag (RS485infoReady).

The function (RS485Comm) in the main program calculates the length of the received packet. The length must be greater than zero so that the received length value can be captured as well as the checksum value. The received packet data is then added up so the checksum can be verified. If the checksums are in agreement then the received length field is compared with the calculated length. Assuming the length fields match, the data is then processed and placed into the params[] array. The values are then assigned to the appropriate variables. If the calculated checksum or data length field does not match the received values then the packet is no longer processed and the receive buffer counter is cleared.

Transmit

Every second a packet composed of the 12 parameters, as shown in Table 2, is sent to the DCS.

Table 2: Parameters Transmitted to DCS

Name	Parameter Number	IMMI variable name
Valve1 Position	1	Valve1Position
Valve2 Position	2	Valve2Position
Sensor1 Frequency	3	sensor1Freq
Sensor1 Rate of Change	4	sensor1ROC
Sensor1 vs. Sensor2 Phase Change	5	sensor1_sensor2_phase
Loss of Sensor1 Fault Status	6	LOS1
Sensor1 Deceleration Alarm	7	sensor1Decel
Loss of Sensor2 Fault Status	8	LOS2
Motor Run Time	9	motorRT
Motor Up Movement	10	motorUp
Motor Down Movement	11	motorDown
Sensor2 Magnitude	12	sensor2Mag

The parameters are put into an array and the total numbers of characters are captured. The packet of data is then assembled according to Appendix A. The start characters (2) are placed in the RS485_TxBuffer array element 0 and 1. The next two elements contain the number of characters in the data. There must be less than 100 characters so the value can be placed in RS485_TxBuffer array element 2 and 3. A data count less than 10 means a 0 will be placed in element 2 and the data count will be placed in element 3. The data is then placed in RS485_TxBuffer array starting at element 4 and ending when the all the data has been placed in the array. The checksum is then calculated and placed in the two RS485_TxBuffer array elements after the data. The final RS485_TxBuffer array element contains the end of frame character (3). The RS485 transmitter is enabled and the entire RS485_TxBuffer is transmitted to the DCS.

Pushbuttons/Valves/LEDs

There are two pushbutton switches to control valve positions (Valve1 and Valve2) independently. For each valve, there is also a red LED indicating and open valve and a green LED indicating a closed valve. The valves are not to be operated manually or by DCS command when failure detection scenarios (loss of Sensor1 and loss of Sensor2 OR loss of Sensor2 and Decel Product alarm) are present. Under normal operation, both pushbutton switches operate their valve the same. After a momentary press of the pushbutton switch, the current position is determined.

If the valve is in the open position then the valve will close immediately as indicated by the green LED illuminated and the red LED not illuminated. The valve position is also reported to the DCS as parameter 1 or 2 (depending on what valve pushbutton was pressed).

If the valve is in the closed position then a five second timer will begin counting down. The red LED will begin flashing once per second as the green LED remains illuminated. A second press of the pushbutton must occur after 3 seconds of the original pushbutton press to successfully open the valve. The red LED will then be fully illuminated. The green LED will not be illuminated. The valve position is then reported to the DCS as parameter 1 or 2 (depending on what valve pushbutton was pressed). If a second pushbutton press occurs before 3 seconds has elapsed then the opening process of the valve is terminated and the green LED will remain illuminated while the red LED will stop flashing and not be illuminated. If a second pushbutton does not occur within five seconds of the first pushbutton press then the opening process of the valve is also terminated. The green LED will remain illuminated and the red LED will not be illuminated.

The DCS can also control the position of the valves under normal operation. When a command is sent to either close or open the valves, the request is completed immediately. The LEDs are appropriately illuminated. A request to close valves means the green LED will be illuminated and the red LED will not be illuminated. A request to open the valves means the red LED will be illuminated and the green LED will not be illuminated.

Software Developed for DCS

Communications with IMMI

Communications with the IMMI is required to be a half-duplex RS485 communications network with a custom protocol. The baud rate is set to 9600 baud with no parity, 8 bits, and 1 stop bit.

Receive

The DCS receives a packet from the IMMI once a second and is processed in the same manner as the IMMI code described in section 2.4 under the heading "Communications with DCS".

Transmit

The DCS communicates to the IMMI only when the user/operator requests a valve to be opened or closed. If the operator requests valve1 to be opened or closed then the code under sendpacket1 is executed. If the operator requests valve2 to be opened or closed then the code under sendpacket2 is executed. Both packets are assembled as described in the transmit section under Communications with DCS.

Communications with HMI

Communications with the Human Machine Interface (HMI) is required to be RS232 communications network. The baud rate is set to 9600 baud with no parity, 8 bits, and 1 stop bit. Communications from the DCS to the HMI will occur once per second.

Receive

The HMI for this application is hyperterminal [14] or any comparable program. Once a connection is established, the user/operator can enter commands on the line where the cursor is present, as shown in Figure 9.

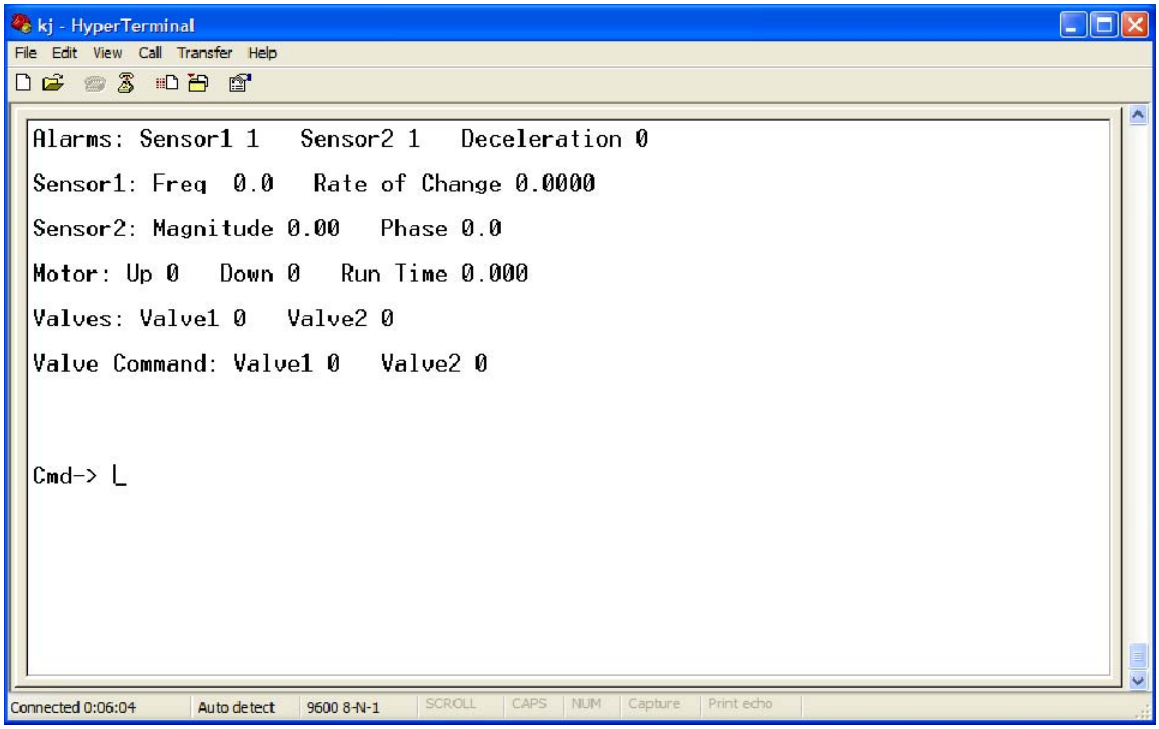


Figure 9: HyperTerminal Display

Each letter typed will be echoed back to the HMI. Only the following commands can be entered: version (this will display the version number of the code running), valve1o (this commands valve1 to open), valve1c (this commands valve1 to close), valve2o (this commands valve2 to open), and valve2c (this commands valve2 to close). Once the command is typed on the hyperterminal screen, the enter key must be pressed for the command to be executed. If a typo occurred, the backspace key can be used to maneuver the cursor to the position in error so the letter/word can be retyped.

Transmit

The DCS transmits all the data shown in Figure 9. The data is updated every second. When the user/operator enters a command, a notice will appear on the screen making the user/operator aware of what is taking place. For instance, if valve1o is typed on the screen and the enter key is pressed then "Valve1 is Opening" will appear, as shown in Figure 10. Anything other than commands listed in the receive section will result in an error message (invalid command).

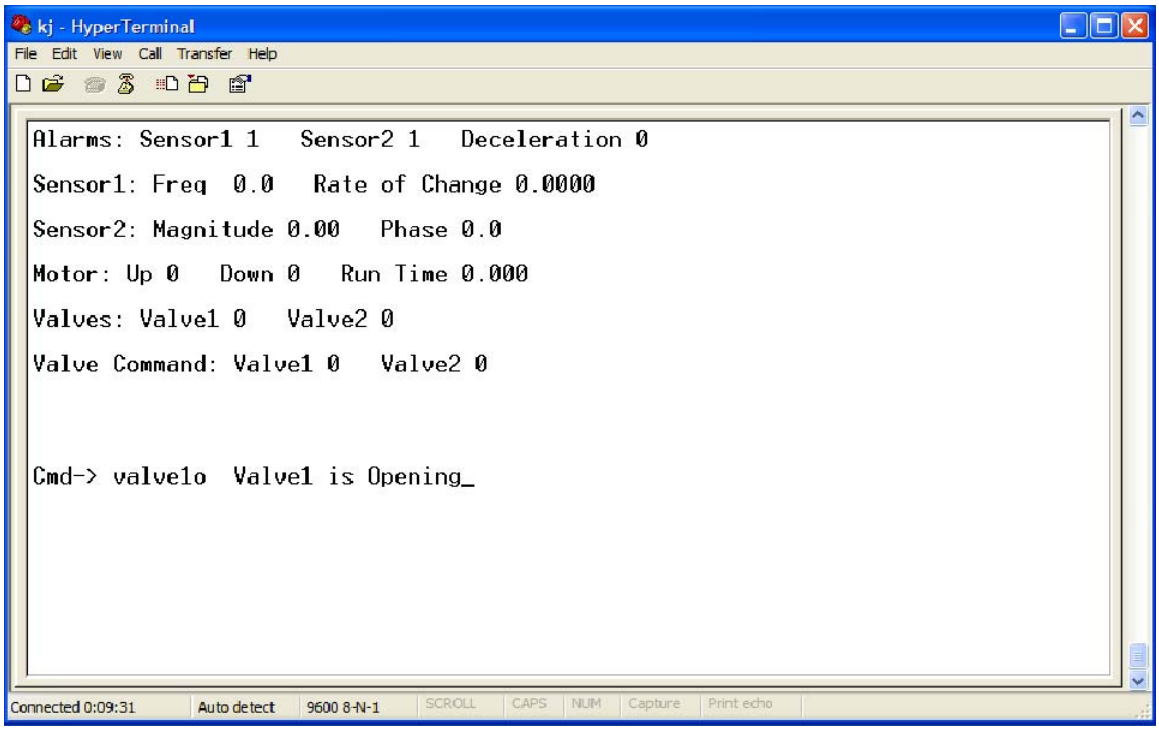


Figure 10: Command Entered by User

The overall system described in Chapter 2 is focused on the operation of the IMMI, which is designed to monitor common industrial machines. The system also includes a DCS to provide a link between a HMI and the IMMI. Finally, realistic machine signals are generated by custom peripheral hardware and applied to the IMMI for system level verification.

Chapter 3: Approach

This chapter describes the approach to the verification process used for this thesis.

Methodology

Although ad hoc verification does work, it is usually utilized after a problem is already identified. The goal to be accomplished for this thesis is to verify the software package before deploying the IMMI. Software verification is a huge task to undertake with many different kinds of verification such as verifying worst case timing, verifying stack size, and performing logic tests to ensure, for example, there are no memory leaks. Therefore, a more structured approach to verification has been chosen but only a few methods of verification will be explored. The verification approach is to find and apply off-the-shelf verification tools directly to the existing software as well as using the hardware created to perform system level testing.

The verification process used for this thesis has similar characteristics to the Functional Verification Cycle, see Figure 11, introduced by Wile, Goss, and Roesner [15]. The verification plan for this thesis is to identify verification tools to assist in analyzing code coverage by placing the two main modules (main code and interrupt code) under test and to identify bugs and weaknesses within the code. The code will also be verified by using the hardware created as a machine emulator.

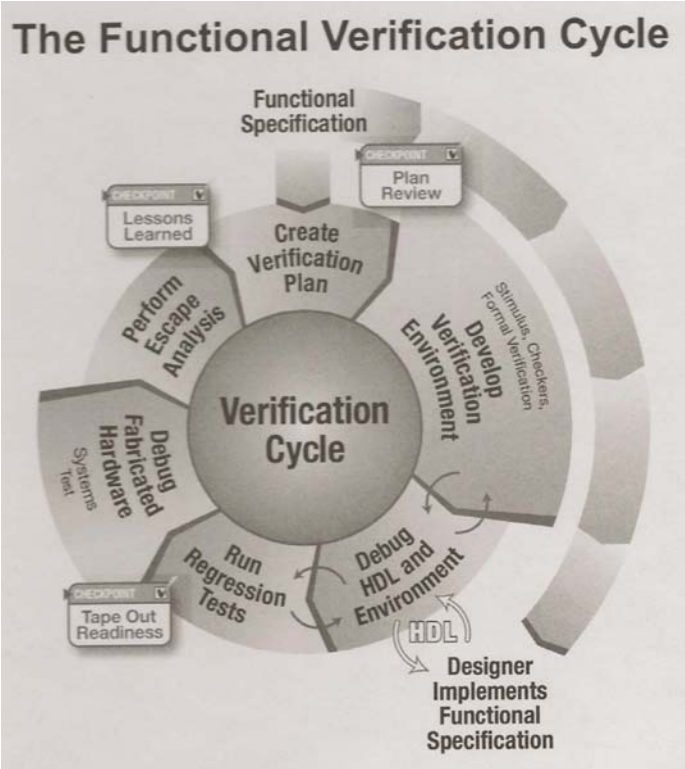


Figure 11: Functional Verification Cycle used for HDL code

The main features of interest for the verification tool(s) are providing the ability to perform unit testing for code coverage testing and identifying bugs and weaknesses within the software package. The criteria used when selecting a verification tool consisted of the tool being geared towards embedded systems, it had to be compatible with CodeWarrior (version 4.6) along with the Processor Expert plug-in, and finally, the tool should be able to be used with Freescale's MC9S12XDP512 microcontroller. After numerous searches, two verification tools were identified: VectorCAST by Vector Software and CodeSonar by GrammaTech. VectorCAST assists in creating test cases that are applied to the software package to verify all lines of code could be reached and functioned as intended [9]. CodeSonar identifies static issues with software such as buffer overflows and divide by zero [10]. An evaluation version of both verification tools is used to complete the verification process. Further details of VectorCAST and CodeSonar are to follow.

VectorCast

"VectorCAST is a suite of tools for automating the entire process associated with conducting unit and integration testing" [9]. VectorCAST builds an environment composed of source code for C, C++, or Ada and generates a test harness for one or more units under test and for any dependent units to be stubbed. Stubbing a unit allows for a function to be tested without having to test another function that is called. This is useful when testing code that is not completed. An advantage of using VectorCAST is the ability to modify source code and then rebuild the environment, which will regenerate the test harness automatically. This allows for the verification process to begin as soon as one function is complete instead of waiting until the entire software package is finalized.

Once an environment is built, test cases can be added and executed. Depending on the type of coverage (statement, branch, or Modified Condition/Decision Coverage) selected, VectorCAST color-codes the source code to indicate the source code lines that were covered by the test case. This information is also captured for the entire unit under test and provided as the Aggregate Coverage Report. Several other reports are also generated that list the execution results of expected versus actual results (listed under Execution Results), pass/fail status of all test cases applied to a unit under test (Management Report), ranking of the function's complexity which indicates the number of unique paths through the function (listed under the Metrics section in other reports), coverage percentage of each function within the source code (also listed under the Metrics section), and a report that includes all the above reports to provide an entire verification report (Full Report).

The test cases are set up to allow the user full control of the inputs and expected results to different variables. A test case can be set up to perform one pass through the code or multiple passes. VectorCAST uses the compiler indicated during setup to execute the tests and the results are then displayed in the VectorCAST environment.

CodeSonar

CodeSonar locates bugs in C/C++ projects while the source code is being compiled. A full build of the project is not necessary with CodeSonar since it can handle incremental builds [10]. The results from CodeSonar are sent to a Hub (database) that is accessed via a Web GUI. The Web GUI displays the analysis results and other information about the build. The analysis provides

all the warnings associated with the project. From there, each warning can be selected to obtain further details (including the exact path through the code that produced the warning). CodeSonar checks for warnings associated with many types of bugs and weaknesses including buffer overrun and underrun, empty statements, integer overflow of allocation size, overlapping memory regions, memory leaks, unreachable code, null pointer dereference division by zero, among others.

Implementation

The strategy for verifying the software package is shown in the flow chart of Figure 12 and Figure 13. The first verification process is to expose the DCS software and the IMMI software to the simulated signals to obtain a system level baseline of the code operation. The code will then be applied to VectorCAST, where an environment will be created to test code coverage based on whether a statement was executed or not. The goal is to create test cases that will achieve 100% coverage of the code. Each test case is to be created based on a requirement(s) from Appendix A. This process for VectorCAST would be similar to the Develop Verification Environment section of Figure 11. The next section of Figure 11, Debugging the HDL and Environment, would consist of executing each test case and reviewing the results to determine if the test case executed as intended and if any modification to the source code should occur. Once the DCS and IMMI software have achieved their maximum coverage, the code is applied to CodeSonar. The verification environment for CodeSonar is the software itself. The debugging process occurs after the analysis report is published by CodeSonar and reviewed for potential problems and fixes identified.

After a modified software package is ready, the code is once again applied to VectorCAST and CodeSonar. This is where the "Run Regression Tests" of Figure 11 occurs. Also, additional test cases within VectorCAST may be required to test any new code that is added. The verification process used, Figure 12 and Figure 13, then calls for the verification of the code using branch coverage. Branch coverage tests for whether each branching statement within the code has been executed as TRUE and/or FALSE.

Finally, the software package is subjected to the same simulated input signals as before for a final system level verification process. This step would compare to Figure 11's section titled "Debug Fabricated Hardware: System Test". If the integration testing of the software and hardware produce expected results then for the purpose of this thesis the verification process would be considered complete. Otherwise, abnormal operation needs to be documented and explained.

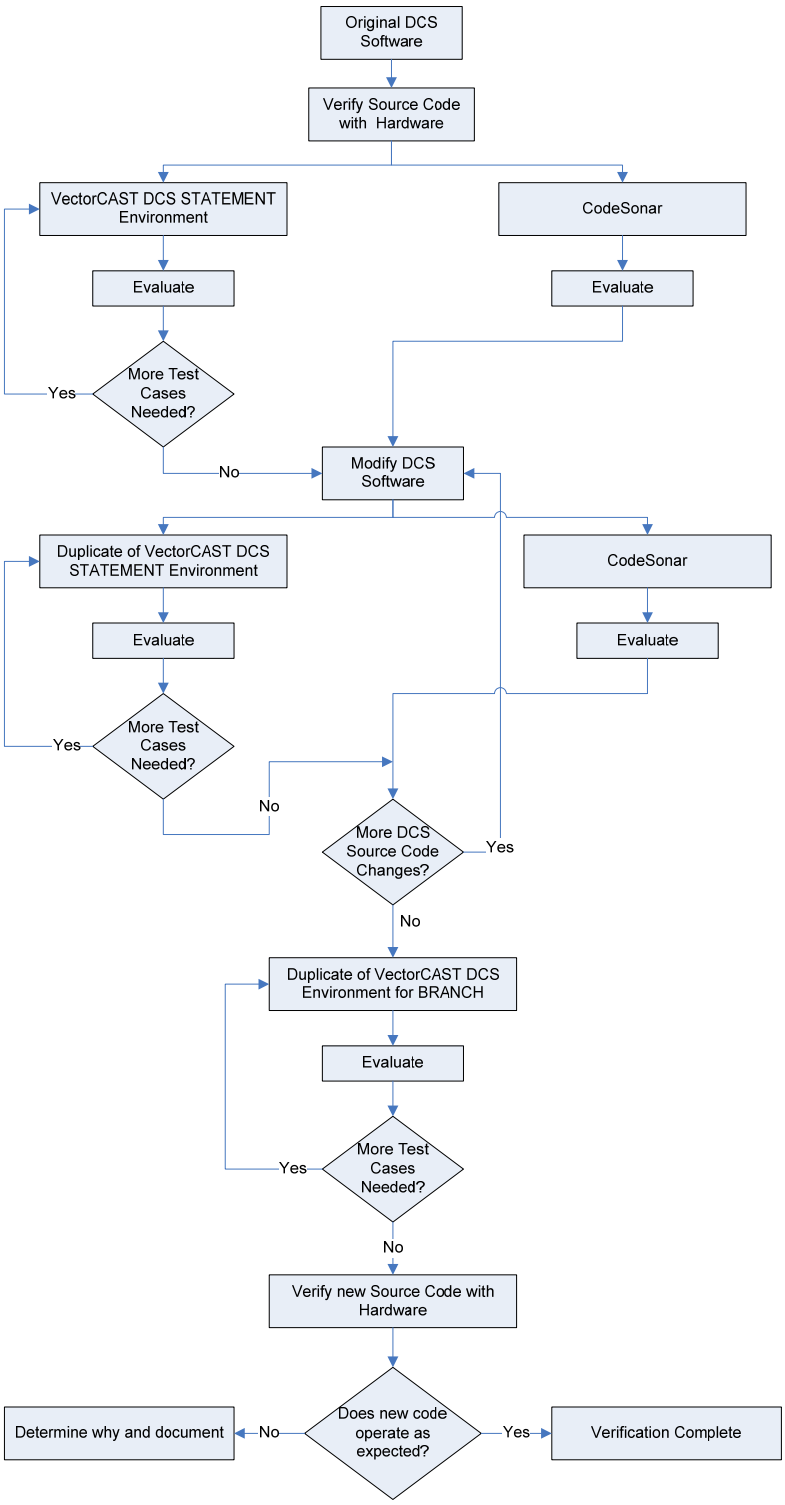


Figure 12: DCS Software Verification Implementation

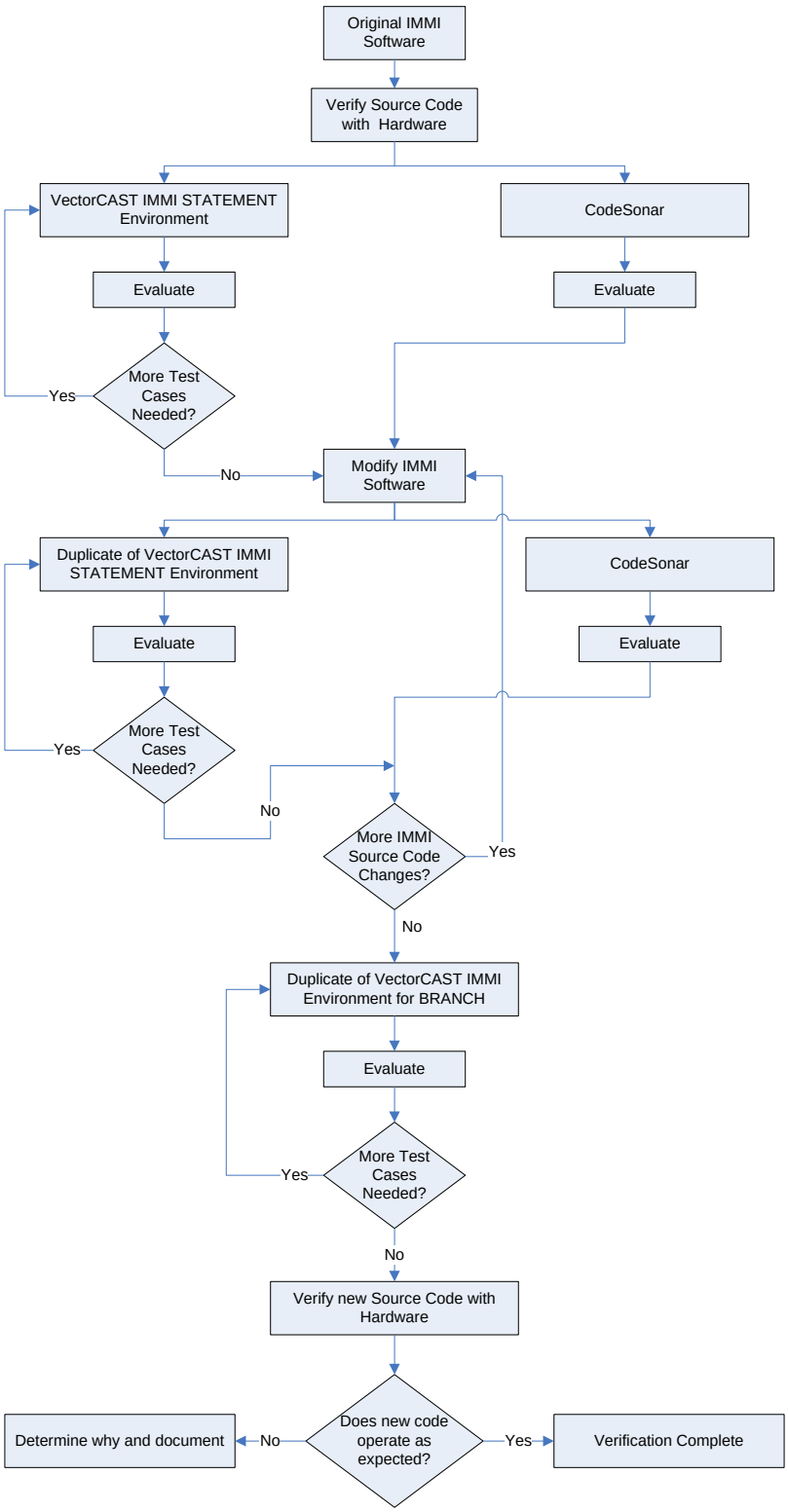


Figure 13: IMMI Software Verification Implementation

Chapter 4: Results

This chapter describes the results that were obtained using VectorCAST and CodeSonar as verification tools. It also describes the software changes made due to the findings and the observed operation of the IMMI and DCS when the set of simulated signals are applied.

DCS

All of the DCS test cases were created to not only test the code but to also ensure the code meets the Design Requirements Document (DRD). Requirement 1 from the DRD is the only requirement the DCS must meet. The test cases for Communications.c and Events.c, as a whole, satisfy the communication requirement.

VectorCAST – Statement Coverage

Original Communications.c

Communications.c includes four functions: main, RS485Comm, parseCommand, and display. A list of functions and the test cases with a description of what is being tested is in Appendix B.i. The metrics report for Communications.c is shown in Table 3. All of the metric reports include the name of the unit under test and the name of all the functions within the code. The table also includes a column for the complexity, which indicates the number of unique paths through the function, a column for the number of test cases created to verify the function, a number of statements or branches covered versus the total number of statements or branches within the function, and finally the percentage of statement or branch coverage reached. The test cases are designed to achieve 100% statement coverage.

Table 3: Communications.c Metrics Report

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Communications	Main	3	1	6/7	85%
	RS485Comm	17	12	48/48	100%
	parseCommand	6	6	33/33	100%
	display	26	12	92/109	84%
Total	**4**	**52**	**31**	**179/197**	**90%**

Two functions achieved 100% coverage (RS485Comm and parseCommand). There were 12 test cases created to fully test RS485Comm and 6 test cases used to fully test parseCommand. The test case created to test the main function had one statement that was not executed as shown in Figure 14.

Statement 7 will never be reached by any test case created. As noted in the comments provided by CodeWarrior/Processor Expert, the infinite for loop is not to be modified. Therefore, the main code will never reach 100% coverage.

There were 12 test cases executed to test the display function but 17 statements were never executed. Figure 15 shows only the statements not covered by the test cases. Statements 2-6 are never tested because "escape" is not recognized as a function in the Events.c code. These statements are marked for removal in the next version of Communications.c. The remaining statements are never reached because the transmission to the IMMI is fixed. The charCount will always be 3 and the checksum may change but will never be less than 100 or greater than 1000. This is the case when sendpacket1 or sendpacket2 are true. These statements can be removed but are not necessary. If they are removed there should still exist a statement that exits from the transmission packet routine so the code doesn't hang.

```
                void  main(void)
                {

                    /* Write your local variable definition here */
                    /*** Processor Expert internal initialization. DON'T REMOVE THIS
 1 1      *         PE_low_level_init();
                    /*** End of Processor Expert internal initialization.
 1 2      *         ( (void)(_PORTE . Byte &= ~(byte)(64)) );
 1 3      *         ( (void)(_PORTE . Byte &= ~(byte)(32)) );
 1 4      *         for(;;)
                    {
 1 5      *            RS485Comm();
 1 6      *            display();
                    }
                    /*** Processor Expert end of main routine. DON'T MODIFY THIS COD
 1 7                for(;;)
                    {
                    }
                      /*** Processor Expert end of main routine. DON'T WRITE CODE BE

                }  /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/
```

Figure 14: Main Function Statement Coverage

```
4 2                escape=0;
4 3                AS1_RxCount=0;
4 4                echo=0;
                  // manualMode=FALSE;
4 5                commandActive=0;
4 6                retValue=printf("\x1B[16;1f\x1B[K");

4 43                  RS485_TxBuffer[2]=charCount[0];
4 44                  RS485_TxBuffer[3]=charCount[1];

4 52                  RS485_TxBuffer[toMainboard+4]=checkSum[0];
4 53                  RS485_TxBuffer[toMainboard+5]=checkSum[1];


4 57                     RS485_TxBuffer[toMainboard+4]=checkSum[2];
4 58                     RS485_TxBuffer[toMainboard+5]=checkSum[3];


4 77                  RS485_TxBuffer[2]=charCount[0];
4 78                  RS485_TxBuffer[3]=charCount[1];


4 86                  RS485_TxBuffer[toMainboard+4]=checkSum[0];
4 87                  RS485_TxBuffer[toMainboard+5]=checkSum[1];


4 91                     RS485_TxBuffer[toMainboard+4]=checkSum[2];
4 92                     RS485_TxBuffer[toMainboard+5]=checkSum[3];
```

Figure 15: Display Function Statements Not Covered

Original Events.c

Events.c includes 12 functions: RS485_OnError, RS485_OnRxChar, Timer_OnInterrupt, RS485_OnFullRxBuf, RS485_OnTxComplete, AS1_OnError, AS1_OnRxChar, AS1_OnTxChar, AS1_OnFullRxBuf, AS1_OnFreeTxBuf, RS485_OnTxChar, and RS485_OnFreeTxBuf. The test cases for each function are listed in Appendix A.i. The metrics report for Events.c is shown in Table 4.

All of the functions that show 0% coverage are not being used. They all need to be removed in the next version of the software. The very last test case in RS485_OnRxChar called 2STARTCHAR&VALIDCHAR failed during execution. This failure was caused because the code continued to be executed even though it should have stopped after three passes. The execution of the test case was stopped manually so the test case was recorded as a failure even though the three passes that were of interest passed the test. Several emails were sent to Vector Software regarding this issue but it could not be resolved. All of the tests cases for AS1_OnRxChar failed because the test was stopped manually.

Table 4: Events.c Metrics Report

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Events	RS485_OnError	1	1	4/4	100%
	RS485_OnRxChar	10	9	28/28	100%
	Timer_OnInterrupt	5	2	12/12	100%
	RS485_OnFullRxBuf	1	0	0/1	0%
	RS485_OnTxComplete	1	1	2/2	100%
	AS1_OnError	1	0	0/1	0%
	AS1_OnRxChar	5	3	11/11	100%
	AS1_OnTxChar	1	0	0/1	0%
	AS1_OnFullRxBuf	1	0	0/1	0%
	AS1_OnFreeTxBuf	1	0	0/1	0%
	RS485_OnTxChar	1	0	0/1	0%
	RS485_OnFreeTxBuf	1	0	0/1	0%
Totals	**12**	**29**	**16**	**57/64**	**89%**

CodeSonar

The entire DCS project, which included Communications.c and Events.c, was rebuilt as CodeSonar eavesdropped. CodeSonar's analysis identified four problems, see Table 5. Three of the problems were located in RS485Comm function and the other problem was located in the display function. Two of the problems were listed as Buffer Overrun and the other two were listed as Redundant Conditions.

The Buffer Overrun in the display function was captured because of the following statement: *ntoMainboard=sprintf(checksum, "%d", nCount);*

It was noted that *sprintf()* is 12 bytes while checksum is 8 bytes. Therefore, the DCS source code will be modified to increase checksum to 12 bytes.

The Buffer Overrun in the RS485Comm function was captured because of the following statement:

for(j=0; j<pdulen; j++) check=check+RS485_RxBuffer[j+2];

The variable *RS485_RxBuffer* is 64 bytes and if *j+2* is greater than 63 there poses a problem. Therefore, a statement will be added to the modified DCS source code to verify *pdulen* is not greater than 62 to ensure there can not be a buffer overrun.

Table 5: CodeSonar's Analysis Report for DCS

File	Function	Line Number	Warning
Communications.c	Display	474	Buffer Overrun
Communications.c	RS485Comm	184	Buffer Overrun
Communications.c	RS485Comm	178	Redundant Condition
Communications.c	RS485Comm	176	Redundant Condition

The final two warnings are Redundant Conditions from the RS485Comm function. The first statement is *if(NULL !=strncpy(hold, RS485_RxBuffer, 2)) pdulen=strtol(hold, NULL, 10);* and the second statement is *if(NULL !=strncpy(hold, RS485_RxBuffer+datalen+2, 2)) chksum=strtol(hold, NULL, 10);*

Both conditions should always evaluate to TRUE but in the unlikely event they evaluate to FALSE the assignment will not be executed. No change will be made due to these warnings since the statements are protective in nature.

VectorCast – Statement Coverage

Communications.c with Changes

The source code that was modified was described in the Communications.c section under Original Code. The final code can be found in Appendix D.i. The only changes that occurred were in the display and RS485Comm functions. The metrics report for the new Communications.c code is shown in Table 6.

No additional test cases were added to test the changes in the RS485Comm function since the existing test cases verified the changes. However, one test case (NTOMAINBOARD) was added to test the changes in the display function. The test case, NTOMAINBOARD, tests to verify the checksum, when *sendpacket1=TRUE,* can not be greater than 3 digits by setting the two variables that are being passed, valve1command and valve2, to their maximum value. The expected result is *ntomainboard* should equal 3 and the test results show that is the case.

Table 6: Metrics Report for Communications.c with Changes

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Communications.c	main	3	1	6/7	85%
	RS485Comm	18	12	51/51	100%
	parseCommand	6	6	33/33	100%
	display	23	13	89/91	97%
Total	**4**	**50**	**32**	**179/182**	**98%**

Events.c with Changes

The only changes that were to occur in Events.c were the removal of unused code. To do this, the Processor Expert *Embedded Beans* had to be modified to have the code disabled. Once this is completed and the code is complied, the unused interrupt events are removed from Events.c. The metrics report, Table 7, shows Events.c was fully tested with 100% statement coverage. The final code for Events.c can be found in Appendix D.iii.

Table 7: Metrics Report for Code Changes within Events.c

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Events.c	RS485_OnError	1	1	4/4	100%
	RS485_OnRxChar	10	9	28/28	100%
	Timer_OnInterrupt	5	2	12/12	100%
	RS485_OnTxComplete	1	1	2/2	100%
	AS1_OnRxChar	5	3	11/11	100%
Total	**5**	**22**	**16**	**57/57**	**100%**

VectorCAST - Branch Coverage

The environment created under statement coverage for both units under test was duplicated but rebuilt for branch coverage. Branch coverage tests for branch functions, such as if statements, to be evaluated as TRUE, FALSE, or equal to a particular value, depending on the condition that is being considered.

To satisfy the branch coverage for Communications.c, additional tests were created to achieve an acceptable coverage percentage. These test cases are described in Appendix B.iii. The metrics report is shown in Table 8.

The test cases that were added to provide additional coverage for the branch statements in Events.c are in Appendix B.iv. The metrics report for Events.c is shown Table 9.

Table 8: Metrics Report using Branch Coverage for Communications.c

Unit	Function	Complexity	Test Cases	Branch Coverage	Coverage Percentage
Communications.c	Main	3	1	2/3	66%
	RS485Comm	18	14	32/35	91%
	parseCommand	6	6	11/11	100%
	display	23	14	41/45	91%
Total	**4**	**50**	**35**	**86/94**	**91%**

Table 9: Metrics Report using Branch Coverage for Events.c

Unit	Function	Complexity	Test Cases	Branch Coverage	Coverage Percentage
Events.c	RS485_OnError	1	1	1/1	100%
	RS485_OnRxChar	10	11	19/19	100%
	Timer_OnInterrupt	5	3	9/9	100%
	RS485_OnTxComplete	1	1	1/1	100%
	AS1_OnRxChar	5	4	9/9	100%
Total	**5**	**22**	**20**	**39/39**	**100%**

CodeSonar

When the modified DCS software was recompiled, CodeSonar was activated. The analysis report identified 4 warnings. The warnings are listed in Table 10.

Three of the four conditions were previously identified by CodeSonar. There was no change to the code to resolve the Redundant Conditions. Therefore, it is expected that they would remain as a warning. The Buffer Overrun, however, was addressed but still remains as a warning. The statement of concern is:

```
for(j=0; j<pdulen; j++) check=check+RS485_RxBuffer[j+2];
```

The variable *RS485_RxBuffer* is 64 bytes and if *j+2* is greater than 63 there poses a problem. The code that was added to correct this problem is:

```
if(pdulen> strlen(RS485_RxBuffer) && pdulen<62){
        RS485_RxCount=0
        Return;
        }
```

This code ensures *pdulen* is less than 62 so when the statement that issued a warning is evaluated for *j + 2*, where *j* is less than *pdulen*, it will not be larger than 63. The additional code should have cleared the warning but since CodeSonar is strictly concerned with *j+2*, instead of a value less than *pdulen +2*, the warning remains.

Table 10: CodeSonar Warning Report for DCS Code with Changes

File	Function	Line Number	Warning
Communications.c	RS485Comm	185	Buffer Overrun
Communications.c	RS485Comm	179	Redundant Condition
Communications.c	RS485Comm	173	Redundant Condition
Communications.c	Main	153	Unreachable Conditional

The Unreachable Conditional warning in the main section of the code was generated because an infinite for loop exists after another infinite for loop. CodeWarrior indicates this section of the code is to remain untouched for Processor Expert.

IMMI

VectorCast – Statement Coverage

Original MIP_LC3081709.c

The main C source code for IMMI is called MIP_LC3081709.c. This file is composed of eleven functions. The functions are called main, RS485Comm, Switches, FrequencyMeasurement, FrequencyActivation, RateofChange, PhaseMeasure, Alarms, isolate, MotorTimer, and ADCMeasurements. A variety of test cases were applied to these functions in order to meet design requirements as well as complete statement coverage. Appendix Chapter 5:C.i provides a list of the test cases along with a description of what is being tested. The metrics report is shown in Table 11.

Table 11: Metrics Report using Statement Coverage for IMMI's MIP_LC3081709.c

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
MIP_LC3081709.c	main	3	1	40/41	97%
	RS485Comm	22	19	64/70	91%
	Switches	28	15	101/113	89%
	FrequencyMeasurement	1	2	3/3	100%
	FrequencyActivation	6	5	24/24	100%
	RateofChange	12	9	40/40	100%
	PhaseMeasure	23	18	89/89	100%
	Alarms	15	11	40/40	100%
	Isolate	1	1	8/8	100%
	MotorTimer	3	2	8/8	100%
	ADCMeasurements	6	5	11/11	100%
Total	**11**	**120**	**88**	**428/447**	**95%**

Original Events.c

There are fifteen interrupts used in IMMI to handle various signals. Appendix Chapter 5:C.ii provides a description of the test cases. Table 12 shows the metrics report.

The functions that did not have 100% coverage were due to the way VectorCAST converts the original source code into the test harness. Any code that directly changes or observes the value of a port, such as changing the value of the signal that determines whether an LED is illuminated or not, is modified in a way that made it impossible to alter the state of the signal for testing purposes. Therefore, the four functions were not able to achieve 100% coverage but had it been possible to alter the input value for the four interrupt functions the complete statement coverage would have been achieved.

Table 12: Metrics Report using Statement Coverage for IMMI's Events.c

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Events.c	Timer_OnInterrupt	15	5	32/32	100%
	AD1_OnEnd	1	1	2/2	100%
	Valve2_Switch_OnInterrupt	2	2	2/4	50%
	Valve1_Switch_OnInterrupt	2	1	2/4	50%
	down_falling_OnInterrupt	2	1	3/3	100%
	down_rising_OnInterrupt	2	1	1/2	50%
	up_falling_OnInterrupt	2	1	3/3	100%
	up_rising_OnInterrupt	2	1	1/2	50%
	Sensor1_OnCapture	4	4	15/15	100%
	Sensor2_OnCapture	2	2	7/7	100%
	Sensor1_Frequency_OnCapture	5	7	16/16	100%
	Sensor1_Frequency_OnOverflow	1	1	3/3	100%
	RS485_OnError	1	1	4/4	100%
	RS485_OnRxChar	10	7	27/27	100%
	RS485_OnTxComplete	1	1	2/2	100%
Total	**15**	**52**	**36**	**120/126**	**95%**

CodeSonar

The warnings uncovered with CodeSonar, while observing IMMI compile, consisted of seventeen warnings, see Table 13, in which there were 14 Buffer Overruns, 1 Useless Assignment, and 2 Redundant Conditions.

The two Buffer Overrun warnings in the RateofChange function are due to the following statement:

```
for(i=60;i>-1;i--){
        array_shift[i+1]=Rate_of_Change_array[i];
        Rate_of_Change_array[i+1]= array_shift[i+1];
}
```

Where *array_shift* and *Rate_of_Change_array* are 60 element arrays.

The first warning indicates *array_shift* is too small and will overrun. The second warning indicates *Rate_of_Change_array* is too small as well causing an overrun. Upon a second look, CodeSonar is exactly correct. When *i* is equal to 60 (maximum value in the for loop), the first assignment within the for loop places the value at *Rate_of_Change_array* element 60 in *array_shift* at element 61. This will cause a buffer overrun since *array_shift* is a 60 element array. When *i* is still equal to 60, the second assignment places the value in *array_shift* element 61 at *Rate_of_Change_array* element 61. Therefore, both arrays really need to be increased in size to 63 element arrays.

Table 13: CodeSonar Warning Report for Original IMMI Code

File	Function	Line Number	Warning
MIP_LC3081709.c	RateofChange	630	Buffer Overrun
MIP_LC3081709.c	RateofChange	631	Buffer Overrun
MIP_LC3081709.c	RS485Comm	299	Buffer Overrun
MIP_LC3081709.c	RS485Comm	344	Buffer Overrun
MIP_LC3081709.c	RS485Comm	356	Buffer Overrun
MIP_LC3081709.c	RS485Comm	358	Buffer Overrun
MIP_LC3081709.c	RS485Comm	359	Buffer Overrun
MIP_LC3081709.c	RS485Comm	361	Buffer Overrun
MIP_LC3081709.c	RS485Comm	362	Buffer Overrun
MIP_LC3081709.c	RS485Comm	364	Buffer Overrun
MIP_LC3081709.c	RS485Comm	365	Buffer Overrun
MIP_LC3081709.c	RS485Comm	367	Buffer Overrun
MIP_LC3081709.c	RS485Comm	368	Buffer Overrun
MIP_LC3081709.c	RS485Comm	371	Buffer Overrun
MIP_LC3081709.c	Main	240	Useless Assignment
MIP_LC3081709.c	RS485Comm	291	Redundant Condition
MIP_LC3081709.c	RS485Comm	293	Redundant Condition

The remaining Buffer Overrun warnings were in the RS485Comm function. The first warning comes from the following statement:

for(j=0;j<pdulen;j++) check=check+RS485_RxBuffer[j+2];

The warning stems from the possibility that pdulen could be greater than 62. If this is the case then RS485_RxBuffer, which is a 64 element array, will overrun. To address this problem, either pdulen needs to be limited in size or RS485_RxBuffer needs to be increased in size.

The second warning is due to the following statement:

toCommboard = sprintf(workBuffer, "%d %d %4.1f %1.4f %4.1f %d %d %d %2.3f %d %d %3.2f", Valve1Position, Valve2Position, sensor1Freq, sensor1ROC, sensor1_sensor2_phase, LOS1, sensor1Decel, LOS2, motorRT, motorUp, motorDown, sensor2Mag);

The warning comes from the possibility that the array, *workBuffer*, is not large enough. It is currently a 64 element array. The values that are to be placed in *workBuffer* vary from Boolean values to floating values. The floating values are given a format for the number of significant digits and the number of decimal places to be used. This format is a minimum for significant digit representation. Therefore, the *workBuffer* array needs to be increased to handle the maximum values. This also means there needs to be additional code that limits the size of the parameters.

The third Buffer Overrun is due to the following statement:

for(j=4; j<toCommboard+4;j++) RS485_TxBuffer[j]=(RS485_TComData)workBuffer[j-4];

The warning suggests *RS485_TxBuffer* will overrun if, as a maximum value, *toCommboard* equaled 93. This warning will be resolved once the size of *workBuffer* and *RS485_TxBuffer* are modified.

The fourth warning is due to the following statement:

for(j=0;j<toCommboard;j++) nCount=nCount+workBuffer[j];

Since workBuffer is a 64 element array, it can not be accessed when *toComboard* is greater than 64. Therefore, this causes a Buffer Overrun warning. This warning will be resolved when workBuffer is increased in size.

The fifth Buffer Overrun is due to the following statement:

ntoCommboard = sprintf(checkSum, "%d", nCount);

Where *checkSum* is an 8 element array.

The warning indicates the number of bytes written to checkSum could potentially be more than the number of allocated bytes. This would occur if nCount is larger than 8 digits, which occurs when workBuffer is extremely large due to the floating values. With a size limit of the floating value variables, an 8 element array for checkSum should be sufficient. However, to ensure checkSum will not overrun the array should be increased to 12 elements.

The sixth through eleventh Buffer Overruns are due to the following statements:

```
If(nCount<100){
        RS485_TxBuffer[toCommboard+4]=checkSum[0];
        RS485_TxBuffer[toCommboard+5]=checkSum[1];
} elseif(nCount>=100 && nCount<1000){
        RS485_TxBuffer[toCommboard+4]=checkSum[1];
        RS485_TxBuffer[toCommboard+5]=checkSum[2];
} else{
        RS485_TxBuffer[toCommboard+4]=checkSum[2];
        RS485_TxBuffer[toCommboard+5]=checkSum[3];
}
```

Where *RS485_TxBuffer* is a 64 element array.

Following the same pattern as before, if *toCommboard* is equal to 93, then *RS485_TxBuffer* will overrun. When *toCommboard* equals 93 and 4 is added, then the statements above would place the values of *checkSum[0], checkSum[1]* and *checkSum[2]* at *RS485_TxBuffer[97]*. The values of *checkSum[1], checkSum[2]* and *checkSum[3]* would be placed at *RS485_TxBuffer[98]* when *toCommboard* is added to 5. The array elements of 97 and 98 are much greater than 64, thus the reason for the Buffer Overrun warnings. The size of *RS485_TxBuffer* may need to be increased but the value will depend on the new maximum size of *workBuffer*, which will be changed as a result of warnings previously discussed.

The final Buffer Overrun is due to the following statement:

RS485_TxBuffer[toCommboard+6]=3;

Using toCommboard equal to 93, RS485_TxBuffer will overrun. Once again, this warning will be resolved when maximum values are added for the transmitted data.

The Useless Assignment warning in the main function is due to two identical statements within ten lines of each other. The assignment is TSCR2_TOI=1, which is used to enable the overflow interrupt. One assignment just needs to be removed.

The final two warnings resulted from Redundant Conditions. The first statement is:

if(NULL !=strncpy(hold, RS485_RxBuffer, 2)) pdulen=strtol(hold, NULL, 10);

and the second statement is:

if(NULL !=strncpy(hold, RS485_RxBuffer+datalen+2, 2)) chksum=strtol(hold, NULL, 10);

These conditions were also present in the DCS software and will be handled the same way, which was no change necessary.

VectorCast – Statement Coverage

MIP_LC3081709.c with Changes

Variable changes to MIP_LC3081709.c code are found in Table 14 and the code that was removed, modified or added is in Table 15.

Table 14: Variables Modified Based on Findings

Variables	Original	New
checkSum[]	8	12
RS485_RxBuffer[]	64	20
RS485_TxBuffer[]	64	80
workBuffer[]	64	74
Rate_of_Change_array[]	60	63
array_shift[]	60	63

Table 15: Source Code Removed, Modified or Added to MIP_LC3081709.c

Function	Action	Code
main	Removed	*TSCR2_TOI=1*
RS485Comm	Removed	*If(nCount<2){* *RS485_TxBuffer[2]=(char)'0';* *RS485_TxBuffer[3]=charCount[0];* *}else{* *RS485_TxBuffer[2]=charCount[0];* *RS485_TxBuffer[3]=charCount[1];* *}*
	Replaced with	*If(nCount>=2){* *RS485_TxBuffer[2]=charCount[0];* *RS485_TxBuffer[3]=charCount[1];* *}else{* *return;* *}*
RS485Comm	Removed	*If(nCount<100){* *RS485_TxBuffer[toCommboard+4]=checkSum[0];* *RS485_TxBuffer[toCommboard+5]=checkSum[1];* *}elseif(nCount>=100 && nCount<1000){* *RS485_TxBuffer[toCommboard+4]=checkSum[1];* *RS485_TxBuffer[toCommboard+5]=checkSum[2];* *}else{* *RS485_TxBuffer[toCommboard+4]=checkSum[2];* *RS485_TxBuffer[toCommboard+4]=checkSum[3];* *}*
	Replaced with	*If(nCount>=1000){* *RS485_TxBuffer[toCommboard+4]=checkSum[2];* *RS485_TxBuffer[toCommboard+5]=checkSum[3];* *}else{* *return;* *}*
RS485Comm	Added	*If(pdulen>(datalen+4) && pdulen<18){* *RS485_RxCount=0;* *return;* *}* *Before….* *If((NULL!=strncpy(hold,RS485_RxBuffer+datalen+2, 2)){* *Chksum=strtol(hold, NULL, 10);* *}*
RS485Comm	Added	*If(Valve1Position<0 || Valve1Position>1){* *Valve1Position=FALSE;*

Function	Action	Code
		Close_Valve1_ClrVal(); *Open_Valve1_SetVal();* *Valve1Sw=FALSE;* *Valve1Count=0;* *}* *If(Valve2Position<0 || Valve2Position>1){* *Valve2Position=FALSE;* *Close_Valve2_ClrVal();* *Open_Valve2_SetVal();* *Valve2Sw=FALSE;* *Valve2Count=0;* *}*
Switches	Removed	*{elseif(Valve1Position==FALSE && Valve1Count>0 && Valve1TimeOut==0){* *Valve1SW=FALSE;* *Valve1Position=FALSE;* *Open_Valve1_SetVal();* *Close_Valve1_ClrVal();* *Valve1Count=0;* *Valve1Valid=0;* *}*
Switches	Removed	*{elseif(Valve2Position==FALSE && Valve2Count>0 && Valve2TimeOut==0){* *Valve2SW=FALSE;* *Valve2Position=FALSE;* *Open_Valve2_SetVal();* *Close_Valve2_ClrVal();* *Valve2Count=0;* *Valve2Valid=0;* *}*
FrequencyActivation	Added	*If(sensor1Freq>9999) sensor1Freq=9999;* *If(sensor1Freq<0) sensor1Freq=0;* In the *if(pDone==TRUE)* statement
RateofChange	Removed	*For(i=60;i>-1;i--){* *array_shift[i+1]=Rate_of_Change_array[i];* *Rate_of_Change_array[i+1]=array_shift[i+1];* *}*
	Replaced with	*For(i=61;i>-1;i--){* *array_shift[i+1]=Rate_of_Change_array[i];* *Rate_of_Change_array[i+1]=array_shift[i+1];* *}*

Function	Action	Code
RateofChange	Removed	*Decel_orig=(array_shift[11]-array_shift[1])/10.24;*
	Replaced with	*Decel_orig=(array_shift[11]-array_shift[1])/10;*
RateofChange	Removed	*If(rate_of_change_counter==59 || min_past==1){* *array_shift[0]=Rate_of_Change_array[0];* *min_past=1;* *Rate_of_Change_orig=((array_shift[0]-array_shift[59])*1.0060);* *sensor1ROC=Rate_of_Change_orig;* *}*
	Replaced with	*If(rate_of_change_counter==61 || min_past==1){* *min_past=1;* *Rate_of_Change_orig=(array_shift[1]-array_shift[61]);* *sensor1ROC=Rate_of_Change_orig;* *if(sensor1ROC>=10) sensor1ROC=9.9999;* *if(sensor1ROC<=-10) sensor1ROC=-9.9999;* *rate_of_change_counter=65;* *}*
Alarms	Added	*If(LOS1<0 || LOS1>1) LOS1=TRUE;* *If(LOS2<0 || LOS2>1) LOS2=TRUE;* *If(sensor1Decel<0 || sensor1Decel>1) sensor1Decel=FALSE;*
MotorTimer	Added	Inside *if(up==TRUE)* statement: *If(motorRT>=100) motorRT=99.999;* *If(motorRT<0) motorRT=0;* Inside *if(down==TRUE)* statement: *If(motorRT<=-100) motorRT=-99.999;* *If(motorRT>0) motorRT=0;*
ADCMeasurements	Added	Inside *case0:* *If(sensor2Mag>=1000) sensor2Mag=999.99;* *If(sensor2Mag<0) sensor2Mag=0;*

The final code for MIP_LC3081709.c can be found in Appendix D.iv. The modified code was then rebuilt with VectorCAST and the test cases were executed to determine the new coverage. There were 11 test cases added to verify the modified source code. The new test cases and descriptions can be found in Appendix Chapter 5:C.iii. The metrics report is shown in Table 16.

Table 16: Metrics Report using Statement Coverage for Modified IMMI Code

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
MIP_LC3081709.c	Main	3	1	39/40	97%
	RS485Comm	24	22	78/80	97%
	Switches	26	15	99/99	100%
	FrequencyMeasurement	1	2	3/3	100%
	FrequencyActivation	8	7	28/28	100%
	RateofChange	14	11	43/43	100%
	PhaseMeasure	23	18	89/89	100%
	Alarms	18	14	46/46	100%
	Isolate	1	1	8/8	100%
	MotorTimer	7	2	12/16	75%
	ADCMeasurements	8	6	14/15	93%
Total	**11**	**133**	**99**	**459/467**	**98%**

Events.c with Changes

There were no changes to the Events.c code for IMMI since there was 100% coverage achieved with the original code and no warnings from CodeSonar. However, the test cases were still applied to Events.c just to verify nothing changed due to the modifications of the main source code, see Table 17 for the metrics report. The final code for Events.c can be found in Appendix D.vi. The only difference between Table 12 and Table 17 is the removal of a test case for Valve2_Switch_OnInterrupt.

Table 17: Metrics Report for Events.c

Unit	Function	Complexity	Test Cases	Statement Coverage	Coverage Percentage
Events.c	Timer_OnInterrupt	15	5	32/32	100%
	AD1_OnEnd	1	1	2/2	100%
	Valve2_Switch_OnInterrupt	2	1	2/4	50%
	Valve1_Switch_OnInterrupt	2	1	2/4	50%
	down_falling_OnInterrupt	2	1	3/3	100%
	down_rising_OnInterrupt	2	1	1/2	50%
	up_falling_OnInterrupt	2	1	3/3	100%
	up_rising_OnInterrupt	2	1	1/2	50%
	Sensor1_OnCapture	4	4	15/15	100%
	Sensor2_OnCapture	2	2	7/7	100%
	Sensor1_Frequency_OnCapture	5	7	16/16	100%
	Sensor1_Frequency_OnOverflow	1	1	3/3	100%
	RS485_OnError	1	1	4/4	100%
	RS485_OnRxChar	10	7	27/27	100%
	RS485_OnTxComplete	1	1	2/2	100%
Total	**15**	**52**	**35**	**120/126**	**95%**

VectorCAST- Branch Coverage

The IMMI VectorCAST environment was duplicated and rebuilt for Branch coverage. Appendix Chapter 5:C.iv and Appendix Chapter 5:C.v show the additional test cases need for Branch coverage. All of the test cases were executed for both units under test and the metrics report for MIP_LC3081709.c is in Table 18.

The one statement in the main function that was not covered was the same statement previously mentioned (infinite for loop for Processor Expert). In the RS485Comm function, there were five lines that were not fully tested. Three of those lines must evaluate to 0 within the if statement and testing for a FALSE condition was not possible. Since the packet to the DCS is created within the IMMI, the statement that places the data length in the packet (*if nCountpacket >=2*) can not be evaluated for less than 2 so it doesn't satisfy the FALSE condition. The same is true for the checksum (*if nCount>=1000*).

In the PhaseMeasure function, only one statement wasn't evaluated fully. After a phase sample is taken, a fraction value greater than 0.625 is discarded. However, the statement that evaluates that can not be tested for a value less than 0.625 since the statements above it already satisfied that condition.

Table 18: Metrics Report using Branch Coverage for MIP_LC3081709.c

Unit	Function	Complexity	Test Cases	Branch Coverage	Coverage Percentage
MIP_LC3081709.c	Main	3	1	2/3	66%
	RS485Comm	24	23	42/47	89%
	Switches	26	15	51/51	100%
	FrequencyMeasurement	1	2	1/1	100%
	FrequencyActivation	8	7	15/15	100%
	RateofChange	14	14	27/27	100%
	PhaseMeasure	23	18	44/45	97%
	Alarms	18	15	35/35	100%
	isolate	1	1	1/1	100%
	MotorTimer	7	2	9/13	69%
	ADCMeasurements	8	6	12/14	85%
Total	**11**	**133**	**104**	**239/253**	**94%**

The MotorTimer function did not evaluate four lines completely. The four lines were the additional code that could not be tested for values greater than 100 and less than zero for an up movement and less than -100 and greater than 0 for a down movement. The reason is due to the assignment type of the timer. The timer is an unsigned integer and can only be as large as 65535 but once it is divided by 1000 the maximum value can only reach 65.535. Therefore it will never reach 100 or -100. This code just becomes a safety net.

The final function that wasn't tested fully is the ADCMeasurements function. The statement testing for Sensor2 magnitude to be less than 0 was not testable. Also, the statement looking for a value greater than 0 was unable to be tested for a value less than 0. Forcing variables defined as positive numbers to negative numbers was not possible with VectorCAST.

The metrics report for Events.c is shown in Table 19. Four tests were added to provide additional coverage for Events.c. The six functions that only achieved 66% coverage were due to the fact that either the TRUE statement (in the case of Valve2_Switch_OnInterrupt, Valve1_Switch_OnInterrupt, down_rising_OnInterrupt, and up_rising_OnInterrupt) or FALSE statement (in the case of down_falling_OnInterrupt and up_falling_OnInterrupt) couldn't be executed because the variable being evaluated was unable to be modified through VectorCAST.

Table 19: Metrics Report using Branch Coverage for Events.c with Changes

Unit	Function	Complexity	Test Cases	Branch Coverage	Coverage Percentage
Events.c	Timer_OnInterrupt	15	6	29/29	100%
	AD1_OnEnd	1	1	1/1	100%
	Valve2_Switch_OnInterrupt	2	1	2/3	66%
	Valve1_Switch_OnInterrupt	2	1	2/3	66%
	down_falling_OnInterrupt	2	1	2/3	66%
	down_rising_OnInterrupt	2	1	2/3	66%
	up_falling_OnInterrupt	2	1	2/3	66%
	up_rising_OnInterrupt	2	1	2/3	66%
	Sensor1_OnCapture	4	4	7/7	100%
	Sensor2_OnCapture	2	2	3/3	100%
	Sensor1_Frequency_OnCapture	5	8	9/9	100%
	Sensor1_Frequency_OnOverflow	1	1	1/1	100%
	RS485_OnError	1	1	1/1	100%
	RS485_OnRxChar	10	9	19/19	100%
	RS485_OnTxComplete	1	1	1/1	100%
Total	**15**	**52**	**39**	**83/89**	**93%**

CodeSonar

After making all the changes to the IMMI source code, the new code was compiled as CodeSonar eavesdropped. This time CodeSonar only found three warnings. Two of those warnings were the Redundant Conditions, which were expected to appear. The last warning was a Buffer Overrun warning in the RS485Comm function. The Buffer Overrun warning was due to the following statement:

for(j=0;j<pdulen;j++) check=check+RS485_RxBuffer[j+2];

Several statements before, new code was added to limit the size of pdulen to less than 18. This should have eliminated the Buffer Overrun warning. Therefore, no changes will be made to remove this warning since limiting the size of pdulen should be sufficient.

Hardware

Before the software package was verified using VectorCAST and CodeSonar, the IMMI software was downloaded to the IMMI prototype board and likewise with the DCS. A serial cable was connected between the DCS prototype board and a PC. On the PC, a hyperterminal program was started and set to 9600 baud, no parity, 8 data bits, and 1 stop bit. The screen immediately displayed all the variables and the data that the IMMI was transmitting. The first line of data on the screen is labeled Alarms as shown in Figure 9. When the value next to Sensor1, Sensor2 and Deceleration is 0 then there is no alarm and when it is 1 then there is an alarm. An alarm status means the valves should have automatically closed. The LEDs will indicate their status (green for closed and red for open). Sensor1 and Sensor2 can easily be removed to demonstrate an alarm status by switching the appropriate hardware switches (SW3 and SW4 respectively). An alarm LED will also be illuminated when a sensor is lost. When

demonstrated, Sensor1 and Sensor2 behave as designed and will recover immediately when returned to a normal state. The alarm LED will immediately extinguish when Sensor1 or Sensor2 are returned to a normal state. To demonstrate a Deceleration alarm, the frequency has to change very rapidly by quickly changing the pot value (R19). The alarm also performs as expected. Once a deceleration alarm is triggered, the recovery time is about seven seconds, even if the alarm should have returned to normal. The alarm LED will flash very rapidly when a failure detection scenario has occurred (loss of Sensor1 and loss of Sensor2 or loss of Sensor2 and deceleration alarm). Both scenarios were tested to confirm the alarm LED behaves as expected.

The second line of data shown in Figure 9 is labeled Sensor1. The Sensor1 data includes the frequency and the rate of change. As R19 is changed in one direction the frequency increases and as it is changed in the opposite direction the frequency decreases. If R19 is left at some random position the frequency will be measured and will remain stable. While the frequency value was changing due to the changing value of R19, the rate of change value was also being updated. As the frequency values were increasing the rate of change value was positive and as the frequency values started decreasing the rate of change values became negative.

The third line is labeled Sensor2 in Figure 9 and includes Magnitude and Phase. To change the magnitude value, R6 is changed to either increase or decrease the value. The magnitude operates as expected and reports a stable value when R6 is stationary. The phase value increases and decreases as R18 is changed. However, when R18 is stationary the phase value is not exactly

stable. Since this is an observed flaw, the intent is to hopefully have VectorCAST and CodeSonar assist in identifying the issue so it can be resolved.

The fourth line in Figure 9 is labeled Motor and contains the Up and Down status lines as well as the run time value. As the motor moves, a value of 0 next to Up and Down represent no movement and a value of 1 represents motor movement. When 1 is displayed for either Up or Down, the run time will be displayed for that particular movement. The run time will also display a negative number when the movement is Down and a positive number for an upward movement. There is no potentiometer to change the rate at which the motor moves up or down. The hardware was designed to continually ramp the movement up and down for increasing lengths. The values displayed for Up, Down and Run Time appear to be correct.

The fifth line in Figure 9 is labeled Valves and includes the status for Valve1 and Valve2. A 0 represents a closed valve and a 1 represents an open valve. This status will change automatically when an alarm occurs. The status will also change when the valve is manually opened or closed using the pushbuttons or when a command is sent via the DCS. As a valve is changed manually from a closed position to an open position, the red LED of the valve attempted to open will begin to flash and the green LED will remain fully illuminated. After three seconds, the pushbutton can be pressed again to complete the opening process. The red LED will remain fully illuminated, the green LED will not be illuminated, and the status of the valve will change to 1. If the opening process of a valve is not adhered to, the red LED will stop flashing, the green LED will remain fully illuminated, and the valve status will be 0. To manually change a valve from

an open position to a closed position only one press of the pushbutton is required to complete the closure of the valve. The red LED will not be illuminated, the green LED will be fully illuminated, and the status of the valve will be 0. The opening and closing of both valves manually work as expected.

The sixth line in Figure 9 is labeled Valve Command and includes the command status for both valves. When a valve is closed and commanded open by a user, the command status is changed from 0 (close valve) to a 1 (open valve). The red LED will immediately be fully illuminated, the green LED will not be illuminated, and the valve status will be 1. The last commanded value will remain until the user commands the valve to open or close. The command is only transmitted by the DCS to the IMMI once. The IMMI performs as expected when commands are sent to open or close the valves.

After the software package has been modified based on the findings from VectorCAST and CodeSonar, the code for the IMMI and DCS are downloaded to their prototype boards for one more verification process using the simulated signals. Following the same sequence of tests using the simulated signals as before, the code produced identical results.

Chapter 5: Conclusions and Future Work

Summary

The goal of this thesis was to verify a software package consisting of C source code for an industrial machine monitoring instrument (IMMI) and a distributed control system (DCS). The software to be verified was first introduced to signals that were developed to simulate a typical environment that the IMMI and DCS would encounter. The hardware prototype provided the ability to perform live debugging and testing. Real performance between the IMMI and DCS could be observed and it was during this verification process that a measurement issue was uncovered with the phase between Sensor1 and Sensor2. No modification was made to the code to correct the problem to allow the verification tools the opportunity to identify the error.

The verification tools selected were VectorCAST [9] and CodeSonar [10]. VectorCAST made it possible to apply numerous kinds of test cases to the software and would then compare the results against expected values. The test cases were created to achieve 100% coverage for statement and branch coverage. By verifying the software with VectorCAST several lines of code were found to be redundant or unreachable. If the code was redundant then it was removed. The code that was determined unreachable actually provided a path for the code to continue if, for instance, a corrupted variable occurred. However, the unreachable code could have been written to just exit and restart the measurement. VectorCAST has many capabilities that were not utilized for this thesis. One such capability is integration testing with the real hardware. This type of testing definitely needs to be pursued as future work.

Once CodeSonar was activated, it eavesdropped on the compilation of the source code looking for static issues that may cause serious problems if left unresolved. In fact, CodeSonar uncovered about 15 or so potential places were a buffer could overrun, 2 redundant conditions, and 1 useless assignment. The items were addressed by increasing array sizes and limiting the size of transmitted variables.

The final software package was then downloaded to prototype boards and tested with the hardware prototype (simulated signals). The new software successfully performed as expected. Neither verification tool addressed the unstable phase reading. Although VectorCAST and CodeSonar uncovered some items that needed to be addressed for creating a strong software package, they do not complete the verification process. These tools are only a small part of the verification process.

Each of the verification tools (hardware prototype, VectorCAST, and CodeSonar) addressed different aspects of the verification process. Hardware prototyping can identify timing issues between functions that can manifest its self into inaccurate measurements. Unit testing can be used to fully verify a function but its interactions with other sections of code are lost unless integration testing is fully tested as well. Finally, static testing can identify problems that may or may not cause an issue that can be easily seen. It is up to the designer to determine if a static issue needs to be addressed.

Future Work

Verifying software is by no means easy nor is there one tool that can do the job. It takes a tremendous amount of time to verify existing software even with complete documentation. The verification of this software package, as previously mentioned, is by no means complete. This software package still needs more integration testing. There needs to be testing to ensure the interrupt code and the main code function as intended when operating as a unit. For example, what happens when an interrupt comes in and is being processed, while at the exact moment the main code is stepping through a section of code that uses those interrupt variables? This type of scenario may be exactly what is causing the unstable phase reading.

References

[1] Brain, Marshall. "How Gyroscopes Work." *Howstuffworks.* Web. 12 Dec. 2010. http://www.howstuffworks.com/gryoscope.htm/
[2] "How a Jet Engine Works." *Explain that Stuff.* Web. 12 Dec. 2010. http://www.explainthatstuff.com/jetengine.html/
[3] "Gyroscope." *Wikipedia.* Web. 12 Dec. 2010. http://en.wikipedia.org/wiki/Gryoscope/
[4] Brain, Marshall. "How Gas Turbine Engines Work." *Howstuffworks.* Web. 12 Dec. 2010. http://science.howstuffworks.com/transport/flight/modern/turbine.htm/
[5] "Turbomolecular Pump." *Wikipedia.* Web. 12 Dec. 2010. http://en.wikipedia.org/wiki/Turbomolecular_pump/
[6] Wildermoth, Brett. *Microprocessor Techniques.* Griffith School of Engineering. Web. 12 Dec. 2010. http://maxwell.me.gu.edu.au/bw/2303eng/Lab-PDFs/OU-07-004.pdf/
[7] "MC9S12XDP512RMV2 Data Sheet." *Freescale Semiconductor.* Rev 2.21. Oct. 2009. Web. 21 Jan 2010. http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S12XDP512RMV2.pdf/
[8] "S12XCPUV1 Reference Manual." Freescale Semiconductor. Rev 0.0101. Mar. 2005. Web. 19 Jan. 2010. http://www.freescale.com/files/microcontrollers/doc/ref_manual/S12XCPUV1.pdf/
[9] "VectorCAST Getting Started." Vector Software. Version 5.1. 19 Oct. 2010.
[10] "CodeSonar® User Guide and Technical Reference." GrammaTech. Release 3.6. n.p.
[11] "CodeWarrior™ Development Studio 8/16-Bit IDE User's Guide." *Freescale Semiconductor.* 27 Sep. 2005.
[12] "Schmitt Trigger." *Wikipedia.* Web. 31 March 2011. http://en.wikipedia.org/wiki/schmitt_trigger
[13] "Processor Expert: CodeWarrior Plug-in for Freescale HCS12/HCS12X User Manual." *Freescale Semiconductor.* Rev. 2.89. n.p.
[14] Shah, Amol. "Configuring HYPERTERMINAL for Serial Communications." *DNA Technology.* Nov. 2007. Web. 31 March 2011. http://www.dnatechnindia.com/index.php/Tutorials/8051-Tutorial/HYPERTERMINAL.html
[15] Wile, Bruce, John C. Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle.* San Francisco: Morgan Kaufmann, 2005. Print.

Appendix

A. Design Requirements Document

The Industrial Machine Monitoring Instrument (IMMI) will monitor and measure the following operating parameters of a machine: frequency, magnitude and phase, and motor operational time. Valves are maintained in a throughput position during normal machine operation. Abnormal conditions such as a loss of sensor1, loss of sensor2, or a rapid deceleration will automatically initiate prompt machine isolation by the IMMI, which is indicated by the red (open) and green (closed) LEDs.

1. Distributed Control System
    a. The IMMI shall communicate with the Distributed Control System (DCS) using a half-duplex RS485 communications set at a 9600 baud rate with a custom protocol scheme.
        i. Custom Protocol is similar to TCP protocol in that it consists of five fields:
            1. Start frame
            2. Data length
            3. Data
            4. Checksum
            5. End of frame
        ii. Each word (one character) in the RS485 packet shall be a 16-bit integer (2 bytes).

Packet Fields	
SF (start frame)	2 STX (binary 0x02) characters
PDU (protocol description unit) Length	Number of characters in the PDU
PDU	Contains the ASCII string of data information
PDU Sum	Sum of the individual PDU characters (decimal values) and truncating so that only the rightmost two digits remain. The PDU sum that is given is the ASCII value of the truncated checksum
EF (end of frame)	1 ETX (binary 0x03) character

b. The IMMI shall transmit parameters 1-12 once per second to the DCS.

Name	Parameter Number
Valve1 Position	1
Valve2 Position	2
Sensor1 Frequency	3
Sensor1 Rate of Change	4
Sensor1 vs. Sensor2 Phase Change	5
Loss of Sensor1 Fault Status	6
Sensor1 Deceleration Alarm	7
Loss of Sensor2 Fault Status	8
Motor Run Time	9
Motor Up Movement	10
Motor Down Movement	11
Sensor2 Magnitude	12

c. DCS shall transmit Valve1 and Valve2 commands to IMMI only upon a command.

2. Sensor1
   a. The IMMI shall measure the frequency of sensor1 signal and report the value as parameter 3 to the DCS.
   b. The change in the sensor1 frequency over the last 60 seconds shall be measured every second and reported to the DCS as parameter 4.
3. Sensor2
   a. The peak signal shall be measured and reported to the DCS as parameter 12.
   b. The relative phase angle between the falling edge of sensor1 and the positive edge of sensor2 shall be measured and reported to the DCS as parameter 5.
4. Motor
   a. The motor run time shall be measured by the IMMI and reported to the DCS as parameter 9.
   b. The IMMI shall also report the “up” direction of motor movements as parameter 10 to the DCS and the “down” direction of motor movements as parameter 11.
5. Alarms
   a. The IMMI shall automatically initiate prompt machine isolation by closing both valves during abnormal conditions (loss of sensor1, loss of sensor2, and decel product)
      i. Loss of Sensor1 (LOS1)
         1. The IMMI shall continuously detect the presence of the sensor1 signal and indicate a LOS1 status if the signal is absent for more than 100ms.
         2. The status of the sensor1 signal shall be reported to the DCS as parameter 6.
      ii. Loss of Sensor2 (LOS2)

1. The IMMI shall continuously detect the presence of sensor2 at all frequencies and indicate a LOS2 status if the signal is absent for more than 100ms.
2. The status of the sensor2 signal shall be reported to the DCS as parameter 8.

iii. Decel Product

1. The IMMI shall measure the sensor1 frequency rate of change value over 10 seconds and multiply it by the sensor1 frequency to calculate the decel product.
2. The IMMI shall issue a decel product alarm and hold the alarm for at least 7.5 seconds when the decel product exceeds 50.0 RPS^2/second and the status shall be reported to the DCS as parameter 7. The alarm status shall be held high at least 7 seconds.

b. The IMMI shall allow de-isolation of the machine from valves 1 and 2 as long as failure detection scenarios are not present.

c. The IMMI shall NOT allow DCS commands or IMMI manual pushbutton commands to open the valves in the event of either failure detection scenario:

i. Loss of Sensor1 and Loss of Sensor2, OR

ii. Loss of Sensor2 and Decel Product alarm

6. Valves and LEDs

a. The IMMI shall place the initial position of the valves in a closed state as indicated by fully illuminating the green LEDs.

b. The IMMI shall control valve1 and valve2 independently.

c. The IMMI shall accept commands for valve1 and valve2 to open/close via the DCS and IMMI manual pushbuttons.

d. The IMMI shall allow DCS commands and IMMI manual pushbutton control to change valve1 and valve2 positions as long as the failure detection scenarios of section 5.c do not exist.

e. The IMMI shall report the status of each valve position to the DCS. Parameter 1 shall represent the position of valve1 as parameter 2 shall represent the position of valve2.

f. *<u>IMMI Manual Pushbutton Control Requirements</u>*

i. The IMMI shall contain one pushbutton for valve1 and another pushbutton for valve2.

ii. Valve position change from closed to open:

1. The IMMI shall respond to a momentary press of the pushbutton, for either valve, to activate the five second timer to begin the opening process of a valve.
2. The IMMI shall respond to a second momentary press of the pushbutton after three seconds but before five seconds has lapsed by opening the valve.
3. The IMMI shall respond to a second momentary press of the pushbutton occurring before three seconds has lapsed by

terminating the opening process of a valve, leaving the valve in the closed position.

4. The IMMI shall respond to not receiving a second momentary press of the pushbutton within five seconds by terminating the opening process of a valve, leaving the valve in the closed position.

iii. *Valve position change from open to closed:*

1. The IMMI shall respond immediately to a momentary press of the pushbutton by closing the valve.

iv. *Red LED labeled "Alarm"*

1. The IMMI shall indicate all abnormal conditions by illuminating the "Alarm" LED.
2. The IMMI shall indicate normal conditions by not illuminating the "Alarm" LED.
3. The IMMI shall indicate a failure detection scenario (section 5.c) is present by flashing the "Alarm" LED at a rate of 200ms.

v. *Red LED labeled "OPEN"*

1. The IMMI shall have two red LEDs to indicate an open valve for both valve1 and valve2.
2. *Valve position change from closed to open using IMMI manual pushbuttons:*
    a. The IMMI shall flash the red LED at a rate of once per second during the opening process of a valve via a manual pushbutton.
    b. The IMMI shall change the state of the red LED from flashing to fully illuminated if the opening process of a valve, initiated by a manual pushbutton, was successful.
3. *Valve position change from open to closed using IMMI manual pushbuttons:*
    a. The IMMI shall change the red LED to the OFF state (not illuminated).
4. *Valve position change from closed to open using DCS commands:*
    a. The IMMI shall change the red LED to the ON state (illuminated).
5. *Valve position change from open to closed using DCS commands:*
    a. The IMMI shall change the red LED to the OFF state (not illuminated).

vi. *Green LED labeled "CLOSE"*

1. The IMMI shall have two green LEDs to indicate a closed valve for valve 1 and vavle2.
2. *Valve position change from closed to open using IMMI manual pushbuttons:*

a. The IMMI shall keep the green LED illuminated during the opening process of a valve, initiated by a manual pushbutton.
b. The IMMI shall change the green LED from the ON state (illuminate) to the OFF state (not illuminated) if the opening process of a valve, initiated by a manual pushbutton, was successful.

3. *Valve position change from open to closed using IMMI manual pushbuttons:*
    a. The IMMI shall change the green LED to the ON state (illuminated).
4. *Valve position change from closed to open using DCS command:*
    a. The IMMI shall change the green LED to the OFF state (not illuminated) upon successfully opening the valve.
5. *Valve position change from open to closed using DCS command:*
    a. The IMMI shall change the green LED to the ON state (illuminated).

B. DCS Test Cases

i. Statement Coverage for Original Main Code - Communications.c

Function	Test Case Name	Description
Main	VCAST_MAIN.001	Verifies communication streams are properly sent from DCS to HMI and from DCS to IMMI.
RS485Comm	VALIDPACKET	A valid packet was sent to the DCS and this test case verifies the packet is handled correctly; the data length and checksum are calculated correctly and are the same as the received data length and checksum.
	INCORRECTDATALENGTH	A packet is sent with the incorrect data length. The DCS should accept the packet and once the received data length is compared to the calculated data length, the packet should be discarded.
	INCORRECTCHECKSUM	A packet is sent with the incorrect checksum. The DCS should accept the packet and once the received checksum is compared to the calculated checksum, the packet should be discarded.
	DATALENGTHZERO	A packet is sent with a data length of zero. The DCS should accept the packet but not proceed further once the received data length is compared with the calculated data length. The packet should then be discarded.
	MOREDATA	A packet is sent with too many data fields. The DCS should accept the packet and once it is determined there are more parameters than expected, the packet should be discarded.
	DATANOTREADY	A packet was sent but the ready flag was not set to TRUE. In this case the packet is not processed and the function is exited.
	PARAM3HIGH	A valid packet is sent and the data for sensor1Freq is set to 4000.0. The displayed data for sensor1Freq should be 3276.7 since the transmitted value is greater than 3276.7. This new value is the maximum value for the displayed parameter.

Function	Test Case Name	Description
RS485Comm	PARAM4HIGH	A valid packet is sent and the data for sensor1ROC is set to 4.2767. The displayed data for sensor1ROC should be 3.2767 since the transmitted value is greater than 3.2767. This new value is the maximum value for the displayed parameter.
	PARAM4LOW	A valid packet is sent and the data for sensor1ROC is set to -4.2767. The displayed data for sensor1ROC should be -3.2767 since the transmitted value is more negative than -3.2767. This new value is the maximum value for the displayed parameter.
	PARAM5HIGH	A valid packet is sent and the data for sensor2Phase is set to 380.0. The displayed data for sensor2Phase should be 0.0 since the transmitted value is greater than 360.0. This new value is the maximum value for the displayed parameter.
	PARAM9HIGH	A valid packet is sent and the data for motorRT is set to 35.767. The displayed data for motorRT should be 32.767 since the transmitted value is greater than 32.767. This new value is the maximum value for the displayed parameter.
	PARAM9LOW	A valid packet is sent and the data for motorRT is set to -35.767. The displayed data for motorRT should be -32.767 since the transmitted value is more negative than -32.767. This new value is the maximum value for the displayed parameter.
parseCommand	VALVE1OPENING	The command “valve1o” is entered, which is a valid command. The appropriate flags should be set so a packet with the information can be transmitted.
	VERSION	The command “version” is entered, which is a valid command. The version number should be transmitted to the hyperterminal for display on the screen.
	VALVE1CLOSING	The command “valve1c” is entered, which is a valid command. The appropriate flags should be set so a packet with the information can be transmitted.

Function	Test Case Name	Description
parseCommand	VALVE2OPENING	The command “valve2o” is entered, which is a valid command. The appropriate flags should be set so a packet with the information can be transmitted.
	VALVE2CLOSING	The command “valve2c” is entered, which is a valid command. The appropriate flags should be set so a packet with the information can be transmitted.
	INCCORECTCOMMAND	The command “valve20” is entered, which is an invalid command. The string “invalid command” is sent to the hyperterminal for display on the screen.
display	VALVE1CLOSE	A command is entered to close valve1. The packet transmitted to the IMMI is verified that the valve1 parameter is set to 0.
	VERSION	A command is entered for the version number. It is verified that nothing is transmitted to the IMMI, only to the hyperterminal.
	HYPERTERMINAL	When nothing is commanded from the hyperterminal, the variables used for holding the messages on the screen are verified to be 0.
	INVALIDCOMMAND	An invalid command is sent. It is verified that nothing is transmitted to the IMMI, only a message that there is an invalid command is transmitted to the hyperterminal.
	BACKSPACE	While entering a command using the hyperterminal, a mistake was made and the backspace key was used. The backspace character should be recognized and the character counter should be decremented so the old value is written over.
	VALVE2CLOSE	A command is entered to close valve2. The packet transmitted to the IMMI is verified that the valve2 parameter is set to 0.
	VALVE2OPEN	A command is entered to open valve2. The packet transmitted to the IMMI is verified that the valve2 parameter is set to 1.
	VALVE1OPEN	A command is entered to open valve1. The packet transmitted to the IMMI is verified that the valve1 parameter is set to 1.

Function	Test Case Name	Description
display	SENDPACKET1	Once a command to open or close valve1 is validated, the packet for transmission to the IMMI is verified for its accuracy.
	STATUSTIMER	The status timer is set to 1000 and verified it was reset to 1000.
	TRANSMITTERFULL.001	The RS485_sendblock is set to have an output value that is representative of a full transmit buffer as the packet trying to be sent is for opening valve1.
	TRANSMITTERFULL.002	The RS485_sendblock is set to have an output value that is representative of a full transmit buffer as the packet trying to be sent is for opening valve2.

ii. Statement Coverage for Original Interrupt Code – Events.c

Function	Test Case Name	Description
RS485_OnError	RS485_ONERROR.001	Once an RS485_OnError interrupt occurs, it is verified a few parameters are reset to zero.
RS485_OnRxChar	2STARTCHARACTERS	Two start characters (2) are sent to verify the interrupt processes but doesn't place them in the receive buffer.
	1STARTCHARACTER	Only one start character is sent to be sure the interrupt code is executed properly.
	1ENDCHARACTER.001	An end character is sent to verify the interrupt code can still be accessed after 2 start characters. A null character is entered into the receive buffer.
	1ENDCHARACTER.002	An end character is sent after other data has been processed and sitting in the receive buffer to verify the data is not overwritten.
	NOSTARTCHARACTER	As a new packet is started, a value other than the expected start character (s) of 2 is sent to verify the character is not accepted.
	2INCORRECTSTARTCHARACTERS	A new packet is started by sending a start character of 2 but instead of receiving another 2, a different character is sent, which should restart the process of accepting a packet.
	CHARACTERGREATERTHAN57	A character greater than 57 in decimal (9 in ASCII) should stop the process of accepting new characters and instead restart the process.

Function	Test Case Name	Description
RS485_OnRxChar	CHARACTERLESSTHAN48	Characters less than 48 in decimal but are not a minus sign (45) or space (32) should stop the process of accepting new characters and instead restart the process.
	2STARTCHAR&VALIDCHAR	A packet that includes two start characters and one valid character should be accepted.
Timer_OnInterrupt	TIMER_ONINTERRUPT.001	The counters are set to various values and each time they should increment.
	TIMER_ONINTERRUPT.002	The counters are set to a different set of values and one is shown to reset once its value is above 1000.
RS485_OnFullRxBuf	NA	
RS485_OnTxComplete	RS485_ONTXCOMPLETE.001	When this interrupt occurs, the disable and enable lines for the RS485 transmitter and receiver are set to 0.
AS1_OnError	NA	
AS1_OnRxChar	BACKSPACE	As a backspace character enters the interrupt code, the backspace flag is set TRUE.
	VALIDCHARACTER	A single valid character enters the interrupt code and should be accepted.
	CARRIAGERETURN	A carriage return enters the interrupt code; it is processed and completes the packet process.
AS1_OnTxChar	NA	
AS1_OnFullRxBuf	NA	
AS1_OnFreeTxBuf	NA	
RS485_OnTxChar	NA	
RS485_OnFreeTxBuf	NA	

iii. Additional Test Cases for Branch Coverage – Communications.c

Function	Test Case Name	Description
RS485Comm	DATALENGTHQUALZERO	A packet is sent with a data length of zero. The DCS should accept the packet but not proceed further once the calculated data length is evaluated to be equal to zero. The packet should then be discarded.
	DATALENNOTEQUALPDULEN	A packet is sent with a data length slightly different from the calculated data length. The packet should not be evaluated any further since they don't match.
display	BACKSPACEECHO	When a backspace is entered, the value should be echoed back to the hyperterminal.

iv. Additional Test Cases for Branch Coverage – Events.c

Function	Test Case Name	Description
RS485_OnRxChar	NOCHARACTER	No character was sent to ensure the interrupt code would exit without evaluating anything.
	SMALLRXCOUNT	A counter was set to be greater than the received buffer so it should not increment since a value somehow got missed.
Timer_OnInterrupt	TIMER_ONINTERRUPT.003	The timers were set to zero to make sure they could not go below 0.
AS1_OnRxChar	NOCHARACTER	No character was sent to ensure the interrupt code would exit without evaluating anything.

C. IMMI Test Cases

i. Statement Coverage for Original Main Code – MIP_LC3081709.c

Function	Test Case Name	Requirement	Description
Main	VCAST_MAIN.001	No Requirement	Tests to ensure all the functions are called and the correct assignments are made.
RS485Comm	VALIDPACKET.001	6.b-d; f.vi.1; f.vi.5.a	A valid packet from DCS commanding both valves closed is received and should be processed correctly.
	VALIDPACKET.002	6.b-d; f.vi.1; f.v.4.a; f.vi.4.a	A valid packet from DCS commanding both valves open is received and should be processed correctly.
	INCORRECTDATALENGHTH	1.a; 1.c; 6.b-d;	A packet from DCS is sent with an incorrect data length. The packet should stop being evaluated when the received data length is compared to the calculated data length.
	DATALENGTHZERO	1.a; 1.c; 6.b-d;	A packet from DCS is sent with no data just a data length value of 0. The packet should not be processed once the calculated data length is determined to be 0.
	MOREDATA	1.a; 1.c; 6.b-d;	A packet from DCS is sent with an extra parameter. Once the extra parameter is recognized, the packet should be discarded.
	DATANOTREADY	1.a; 1.c; 6.b-d;	A packet from DCS is sent but the flag indicating the packet is ready is not set so the packet should not be evaluated.
	PARAM1.001	1.a; 1.c; 6.b-d; f.v.5.a; f.vi.5.a	A packet from DCS that commands valve1 to close and valve2 to close while valve1 is open.

Function	Test Case Name	Requirement	Description
			Verify the correct variables are updated.
RS485Comm	PARAM1.002	1.a; 1.c; 6.b-d;	A packet from DCS that commands valve1 to open and valve2 to close while valve1 is open. Verify the correct variables are updated.
	PARAM1.003	1.a; 1.c; 6.b-d; f.v.4.a; f.vi.4.a	A packet from DCS that commands valve1 to open and valve2 to close while valve1 is closed. Verify the correct variables are updated.
	PARAM2.001	1.a; 1.c; 6.b-d; f.v.5.a; f.vi.5.a	A packet from DCS that commands valve1 to close and valve2 to close while valve2 is open. Verify the correct variables are updated.
	PARAM2.002	1.a; 1.c; 6.b-d;	A packet from DCS that commands valve1 to close and valve2 to open while valve2 is open. Verify the correct variables are updated.
	PARAM2.003	1.a; 1.c; 6.b-d; f.v.4.a; f.vi.4.a	A packet from DCS that commands valve1 to close and valve2 to open while valve2 is closed. Verify the correct variables are updated.
	MOTORRTEQUALZERO	1.a-c; 4.a-b; 5.e; 6.b-d;	A packet from DCS is sent to close both valves and all the variables to be transmitted to DCS are set with reasonable values except MotorRT is set to 0. Ensure the packet is assembled correctly and motor up and down are set to 0.
	MOTORRTBIG	1.a-c; 4.a-b; 5.e; 6.b-d;	A packet from DCS is sent to close both valves and all the variables to be transmitted to DCS are set with reasonable values and MotorRT is set to 0.525. Ensure the packet is assembled correctly and motor up is

Function	Test Case Name	Requirement	Description
			1 while motor down is 0.
RS485Comm	MOTORRTSMALL	1.a-c; 4.a-b; 5.e; 6.b-d;	A packet from DCS is sent to close both valves and all the variables to be transmitted to DCS are set with reasonable values and MotorRT is set to -0.3124. Ensure the packet is assembled correctly and motor up is 0 while motor down is 1.
	TRANSMITTERFULL.001	No Requirement	The RS485_sendblock is set to have an output value that is representative of a full transmit buffer as a packet trying to be sent is for opening valve1 while a packet is reviewed for opening valve1.
	TRANSMITTERFULL.002	No Requirement	The RS485_sendblock is set to have an output value that is representative of a full transmit buffer as a packet trying to be sent is for opening valve2 while a packet is reviewed for opening valve2.
	MINDATA	1.a-b	All the data to be transmitted to DCS are set to zeros. There should be no problem creating the packet. It also provides info on how small the packet would potentially be.
	MAXDATA	1.a-b	All the data to be transmitted to DCS are set to very large numbers. The packet should limit the size of the data.
Switches	SWITCHESDONTWORK	5.c	Set the flag to disable valve operation both manually and via command while attempting to open a valve. Valve should not

Function	Test Case Name	Requirement	Description
			open.
	MANUALVALVE1OPENTOCLOSE	5.b; 6.b-d; 6.f.v.1; 6.f.v.3.a; 6.f.vi.3.a; 6.f.iii.1	Close valve1 from an open position manually.
Switches	MANUALVALVE2OPENTOCLOSE	5.b; 6.b-d; 6.f.v.1; 6.f.v.3.a; 6.f.vi.3.a; 6.f.iii.1	Close valve2 from an open position manually.
	MANUALVALVE1CLOSETOOPEN.001	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.1	Open valve1 from a closed position manually. Verify opening process begins and red LED is told to flash.
	MANUALVALVE1CLOSETOOPEN.002	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.3	Opening process of valve1 has begun and a second press of the pushbutton occurs 1 second after the first press. Verify opening process is terminated.
	MANUALVALVE1CLOSETOOPEN.003	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.b; 6.f.vi.2.b; 6.f.ii.2	Opening process of valve1 has begun and a second press of the pushbutton occurs after 3 seconds of the first press. Verify opening process is completed.
	MANUALVALVE1CLOSETOOPEN.004	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.4	Opening process of valve1 has begun and a second press of the pushbutton never occurs. Verify opening process is terminated.
	MANUALVALVE2CLOSETOOPEN.001	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.1	Open valve2 from a closed position manually. Verify opening process begins and red LED is told to flash.
	MANUALVALVE2CLOSETOOPEN.002	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.3	Opening process of valve2 has begun and a second press of the pushbutton occurs 1 second after the first press. Verify opening process is terminated.

Function	Test Case Name	Requirement	Description
	MANUALVALVE2CLOSETOOPEN.003	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.b; 6.f.vi.2.b; 6.f.ii.2	Opening process of valve2 has begun and a second press of the pushbutton occurs after 3 seconds of the first press. Verify opening process is completed.
Switches	MANUALVALVE2CLOSETOOPEN.004	5.b; 6.b-d; 6.f.v.1; 6.f.v.2.a; 6.f.vi.2.a; 6.f.ii.4	Opening process of valve2 has begun and a second press of the pushbutton never occurs. Verify opening process is terminated.
	DCSCOMMANDVALVE1CLOSE	1.a; 1.c; 5.b; 6.b-d; 6.f.v.5.a; f.vi.5.a	DCS commands valve1 to close while valve1 is open.
	DCSCOMMANDVALVE1OPEN	1.a; 1.c; 5.b; 6.b-d; f.v.4.a; f.vi.4.a	DCS commands valve1 to open while valve1 is closed.
	DCSCOMMANDVALVE2CLOSE	1.a; 1.c; 5.b; 6.b-d; f.v.5.a; f.vi.5.a	DCS commands valve2 to close while valve2 is open.
	DCSCOMMANDVALVE2OPEN	1.a; 1.c; 5.b; 6.b-d; f.v.4.a; f.vi.4.a	DCS commands valve2 to open while valve2 is closed.
FrequencyMeasurement	GOODMEASUREMENT	2.a	Values are provided such that a value should be calculated for the frequency.
	DIVIDEBYZERO	No Requirement	Values are provided that would result in a division by zero. The resulting frequency value should be set to 0.
FrequencyActivation	STARTACTIVATION.001	2.a	A frequency measurement is to begin and the variables should be cleared depending on the values for the period (1, 2 and 3) and the overflow set to 0.
	STARTACTIVATION.002	2.a	A frequency measurement is to begin and the variables should be cleared depending on the values for the period (1, 2 and 3) and the overflow is set to 8.
	MAKEMEASUREMENT	2.a	All the variables are

Function	Test Case Name	Requirement	Description
			obtained and the ready flag is set for a frequency measurement to be calculated.
FrequencyActivation	LOSTSENSOR1	5.a.i.1-2	A frequency measurement was set to be made when sensor1 was removed. The frequency value reported should be 0.
	TIMEREQUALZERO	No Requirement	A frequency measurement could have been made but the ready flag was not set and the 8 second timer expired. The variables should be cleared and a new measurement should begin.
RateofChange	NOTREADY	No Requirement	The flag indicating a frequency measurement has completed is set to FALSE. The rate of change code should not be executed.
	SENSOR1MISSING.001	5.a.i.1	The frequency measurement has been completed but sensor1 is missing and the rate of change timer is greater than 0. The rate of change function should be exited.
	SENSORMISSINGLESSTHAN10SEC	5.a.i.1	Sensor1 is missing. It has been less than 10 seconds so the frequency should be placed in the array and will be 0.0. No calculations will be performed since it's been less than 10 seconds.
	SENSORMISSINGMORETHAN10SEC	5.a.iii.1-2	Sensor1 is missing. It has been more than 10 seconds but less than 60 seconds. The frequency should be placed in the array and will be 0.0. Since the entire 10 second array is filled with 0.0 then the decel product should evaluate

Function	Test Case Name	Requirement	Description
			to 0 and no alarm will be initiated.
RateofChange	SENSORMISSINGMORETHAN60SEC	2.b; 5.a.i.1; 5.a.iii1-2	Sensor1 is missing. It has been more than 60 seconds. The frequency should be placed in the array and will be 0.0. Since the entire 60 second array is filled with 0.0 then the decel product should evaluate to 0 as well as the Rate of Change value. No deceleration alarm will be initiated.
	DECELDATA.001	5.a.iii.1-2	The rate of change array is filled with the same value. A new smaller value is entered and the decel is less than the trip value.
	ROCDATA.002	2.b	The rate of change array is filled with the same value. A new smaller value is entered and the rate of change will be a negative number.
	DECELDATA.002	5.a.iii.1-2	The rate of change array is filled with the same value. A new smaller value is entered and the decel is greater than the trip value so the decel alarm is activated and isolation occurs.
	DECELDATA.003	5.a.iii.1-2	Same test as above but no isolation since the alarm is already active.
PhaseMeasure	STARTMEASUREMENT	3.b	A phase measurement is ready to begin so all the variables are to be cleared and the timeout timer is to be reset.
	SAMPLE.001	3.b	Values are provided to place a sample in Bin1 and Bin5.
	SAMPLE.002	3.b	Values are provided to place a sample in Bin2.
	SAMPLE.003	3.b	Values are provided to place a sample in Bin3.
	SAMPLE.004	3.b	Values are provided to

Function	Test Case Name	Requirement	Description
			place a sample in Bin4.
	SAMPLE.005	3.b	Values are provided to place a sample in Bin5.
PhaseMeasure	SAMPLE.006	3.b	Values are provided to place a sample out-of-range.
	SAMPLE.007	3.b	Values are provided such that the sensor2_sensor1_fraction is equal to 0, which is not used as a sample.
	16SAMPLES.001	3.b	16 sample values are provided landing in bin1, bin2 and bin5 to produce 90 degrees.
	16SAMPLES.002	3.b	Same as above.
	16SAMPLES.003	3.b	16 sample values are provided landing in bin3 to produce 250 degrees.
	16SAMPLES.004	3.b	16 sample values are provided landing in bin5 to produce 55 degrees.
	16SAMPLES.005	3.b	16 sample values are provided landing in bin5 to produce 0 degrees.
	16SAMPLES.006	3.b	16 sample values are provided landing in bin1 and bin5 to produce 0.40 which is then evaluated to be 0 degrees.
	16SAMPLES.007	3.b	16 sample values are provided landing in bin1 and bin5 to produce 360 degrees.
	16SAMPLES.008	3.b	16 sample values are provided landing in bin5 to produce 360 degrees.
	16SAMPLES.009	3.b	16 sample values are provided landing in bin1 and bin5 to produce 0.074 which is between 0.05 and 0.1 and is evaluated to be 0.1 degrees.
	TIMEOUT	No Requirement	The eight second timer expires because a new measurement has not been made so variables need to be reset.
Alarms	FAILUREDETECTION.001	5.c.i; 6.d; 6.f.iv.3	Sensor1 and Sensor2 are missing. Valves should

Function	Test Case Name	Requirement	Description
			not be operated manually or via a command. Alarm LED should flash quickly and is checked to be off for 100ms.
Alarms	FAILUREDETECTION.002	5.c.ii; 6.d; 6.f.iv.3	Sensor2 is missing and Deceleration alarm is active. Valves should not be operated manually or via a command. Alarm LED should flash quickly and is checked to be off for 100ms.
	LOS1ALARM	5.a; 6.f.iv.1	Sensor1 has been removed. Isolation should occur and alarm LED should be on.
	LOS1NORMAL	6.f.iv.2	Sensor1 has been restored. The alarm LED should be off.
	LOS2ALARM	5.a; 5.a.ii.1-2; 6.f.iv.1	Sensor2 has been removed. Isolation should occur and alarm LED should be on.
	LOS2NORMAL	5.a.ii.1-2; 6.f.iv.2	Sensor2 has been restored. The alarm LED should be off.
	FAILUREDETECTION.003	5.c.i; 6.d; 6.f.iv.3	Sensor1 and Sensor2 are missing. Valves should not be operated manually or via a command. Alarm LED should flash quickly and is checked to be on for 100ms.
	FAILUREDETECTION.004	5.c.ii; 6.d; 6.f.iv.3	Sensor2 is missing and Deceleration alarm is active. Valves should not be operated manually or via a command. Alarm LED should flash quickly and is checked to be on for 100ms.
	DECELALARM.001	5.a.iii.2	Deceleration alarm has occurred and the hold up timer has not expired. Decel alarm is calculated to still be active but no isolation occurs.
	DECELALARM.002	5.a.iii.2	Deceleration alarm is active, the timer has expired and another

Function	Test Case Name	Requirement	Description
			deceleration alarm comes in. The timer is not restarted and no isolation occurs.
Alarms	DECELALARMTONORMAL	6.f.iv.2	Deceleration has been in alarm and is returned to normal. Alarm LED should be off.
Isolate	ISOLATIONACTIVATED	5.a	Isolation is to occur. Verify valve positions are closed.
MotorTimer	UP	4.a-b	An up movement has been made and measured.
	DOWN	4.a-b	A down movement has been made and measured.
ADCMeasurements	SENSOR2MAG.001	3.a	A magnitude value was obtained and reported.
	SENSOR2MAG.002	3.a	A magnitude value was not captured so the code doesn't execute.
	SENSOR2MAG.003	3.a	A value was captured on another channel. Verify the value doesn't get reported as the magnitude.
	SENSOR2MAG.004	3.a	The channel number is greater than 0. Ensure the channel number is reset to 0.
	SENSOR2MAG.005	3.a	A magnitude value is captured but if the phase value can not be measured then don't measure the magnitude.

ii. Statement Coverage for Original Interrupt Code – Events.c

Function	Test Case Name	Requirement	Description
Timer_OnInterrupt	TIMERS.001	No Requirement	TimerCounter, which is a blink timer to indicate the microcontroller is alive, is set to 0 and will increment to 1. Other timers set at different values yet valid values. Some will increment and others will decrement.
	TIMERS.002	5.a.i.1	All timers are set as Timers.001 with the exception of LOS1 timer, which is set to 0 to cause a loss of Sensor1 alarm.
	TIMERS.003	5.a.ii.1	All timers are set as Timers.001 with the exception of LOS2 timer, which is set to 0 to cause a loss of Sensor2 alarm.
	TIMERS.004	No Requirement	TimerCounter is set to 499 and should increment to 500 while the others are set at various values.
	TIMERS.005	No Requirement	TimerCounter is set to 999 and should increment to 1000 and then reset to 0 while the others are set at various values.
AD1_OnEnd	AD1CHANNEL	3.a	Channel1 is set to 0 so the value that would be captured is for Sensor2 magnitude.
Valve2_Switch_OnInterrupt	VALVE2PUSHBUTTON.001	6.b-c; 6.f.i	Valve2 is placed in an open position. When an interrupt occurs the valve switch flag is set to close the valve.
	VALVE2PUSHBUTTON.002	6.b-c; 6.f.i	Valve2 is placed in a closed position. When an interrupt occurs the valve switch flag is set to open the valve.

Function	Test Case Name	Requirement	Description
Valve1_Switch_OnInterrupt	VALVE1PUSHBUTTON.001	6.b-c; 6.f.i	Valve1 is placed in an open position. When an interrupt occurs the valve switch flag is set to close the valve.
down_falling_OnInterrupt	FALLINGEDGEDOWN	4.a-b	A falling edge is received, which should stop the down timer and let the main code know the run time calculation can be completed.
down_rising_OnInterrupt	RISINGEDGEDOWN	4.a-b	A rising edge is received, which should start the down timer.
Up_falling_OnInterrupt	FALLINGENDGEUP	4.a-b	A falling edge is received, which should stop the up timer and let the main code know the run time calculation can be completed.
Up_rising_OnInterrupt	RISINGEDGEUP	4.a-b	A rising edge is received, which should start the up timer.
Sensor1_OnCapture	SENSOR1PHASEVALVETOBIG	3.b	The value of icapture3 is greater than 65000, which should set the restart flag for use by the main code.
	SENSOR1PHASEVALID	3.b	A valid icapture3 value is received during the first period so the overflow flags are reset and sensor2 has the ability to capture a value.
	SENSOR1PULSE1	3.b	A value for icapture3 was captured and on the second period only a counter is incremented.
	SENSOR2PULSE2	3.b	A value is provided for icapture3 and the period is set to 2, which means icapture5 is captured and a measurement is ready to be made.
Sensor2_OnCapture	SENSOR2ACTIVATED	3.b	The flag that arms sensor2 is active so a

Function	Test Case Name	Requirement	Description
			value should be captured. The overflow value is also captured and other flags are reset.
Sensor2_OnCapture	SENSOR2NOTACTIVATED	3.b	The arm flag for sensor2 is not set. Nothing should be executed.
Sensor1_Frequency_OnCapture	PERIODSEQUALZEROBIG	2.a	The value for icapture1 is 65510, which forces a restart of the measurement. Since the interrupt occurred the LOS1 counter reset.
	PERIODSEQUALZEROVALID	2.a	During the first period, icapture1 is valid (17352), which will reset the overflow counter. Since the interrupt occurred the LOS1 counter reset.
	LARGEPERIODS	2.a	The number of periods is set large with an overflow that is set to 4. This means more periods is needed before all the values for the measurement can be captured.
	SIXOVERLFOWS.001	2.a	The number of periods is set large but the overflow is set to 6. The active flag is set to TRUE so icapture2 is captured and the number of overflows and periods are reassigned.
	SIXOVERFLOWS.002	2.a	The number of periods is set to 2 and overflow is set to 6. The active flag is set to TRUE so icapture2 is captured and the number of overflows and periods are reassigned.
	SIXOVERFLOWS.003	2.a	Period is set to 0 so a valid icapture1 can be obtained. A period of

Function	Test Case Name	Requirement	Description
			1, an overflow of 6, and active flag are then set. A value for icapture2 is obtained and a measurement is ready to be calculated.
Sensor1_Frequency_OnCapture	OVERFLOWSBIGGER	2.a	Overflow is set to 7 with a large period and active set to TRUE. The value for icapture2 should be obtained.
Sensor1_Frequency_OnOverflow	OVERFLOWOCCURRED	2.a; 3.b	The interrupt for an overflow occurred and the previous count number was set to 4 so it should be incremented.
RS485_OnError	RS485_ONERROR.001	1.a	A communication error occurred so this interrupt resets variables used to form the packet.
RS485_OnRxChar	2STARTCHAR	1.a	Two start characters are received and it's verified they are handled correctly.
	1STARTCHAR	1.a	Only one start character is received. Therefore the code should be waiting on another start character otherwise the process must begin.
	INVALIDNUMBER.001	1.a	Two start characters received and then an invalid character (57). The whole process should be reset.
	INVALIDNUMBER.002	1.a	Two start characters are received and then an invalid character (47). The process should be reset.
	NOSTARTCHAR	1.a	No start characters are received, just a valid character. The valid character should be ignored.
	2START1ENDCHAR	1.a	Two start characters are received and then an end character. This

Function	Test Case Name	Requirement	Description
			is allowed even though nothing is transmitted.
RS485_OnRxChar	VALIDCASE	1.a	A valid packet is received and should be handled correctly.
RS485_OnTxComplete	TXCOMPLETE	No Requirement	Once a transmission is completed the transmitter should be disabled and the receiver should be enabled.

iii. Statement Coverage for Modified Main Code – MIP_LC3081709.c

Function	Test Case Name	Requirement	Description
RS485Comm	PDULEN	1.a	The received data length is larger than the calculated data length. The packet should not be processed after received data length is compared to calculated data length.
	VALVE1POSITIONBIG	No Requirement	If Valve1 position is corrupted (not 1 or 0), the valve should be closed.
	VALVE2POSITIONBIG	No Requirement	If Valve2 position is corrupted (not 1 or 0), the valve should be closed.
FrequencyActivation	LARGEFREQUENCY	No Requirement	A frequency above 9999 is calculated. The value should be set to 9999.0.
	SMALLFREQUENCY	No Requirement	A frequency below 0 is calculated. The value should be set to 0.
RateofChange	ROCDATABIG	No Requirement	The same values are placed in the rate of change array. The next value that comes in is larger. The calculation is above 10 RPS/minute. Therefore, it should be set to 9.9999.
	ROCDATASMALL	No Requirement	The same values are placed in the rate of change array. The next value is smaller and the rate of change is calculated to be more negative than -9.9999. The resulting value should be set to -9.9999.
Alarms	BADLOS1	No Requirement	The loss of Sensor1 value is corrupted and is not 0 or 1. The value should be reset to 1.
	BADLOS2	No Requirement	The loss of Sensor2 value is corrupted and is not 0 or 1. The value should be reset to 1.
	BADDECEL	No Requirement	A deceleration alarm has occurred but the value reported is corrupted. The value should then be reset to 0.
	ISOLATIONACTIVATED	5.a	All the valves are open. When isolation occurs verify all the valves close and the positions are reported as closed.

iv. Additional Test Cases for Branch Coverage – MIP_LC3081709.c

Function	Test Case Name	Requirement	Description
RS485Comm	INCORRECTCHECKSUM	1.a	The received checksum is not correct. Once the calculated checksum is compared with the received, the packet should be discarded.
RateofChange	DECELDATACOUNTER.001	5.a.iii.1	The rate of change array is filled with values and the decel counter is set to 1. The decel will trip based off the array element values. The counter should increment and isolation should occur.
	DECELDATACOUNTER.002	5.a.iii.2	The same values as DECELDATACOUNTER.001 are used. However, a decel alarm has already occurred. The counter should still increment but no isolation.
	DECELDATABIG	No Requirement	The value for a parameter used in the decision making within the deceleration alarm was corrupted. The deceleration alarm will be set as normal.
Alarms	FAILUREDETECTION.005	5.c.ii; 6.f.iv.3	Loss of Sensor2 is in alarm and so is the deceleration alarm. The Alarm LED is verified to be on for 100ms.

v. Additional Test Cases for Branch Coverage – Events.c

Function	Test Case Name	Requirement	Description
Timer_OnInterrupt	TIMERS.006	No Requirement	All the timers are set to zero. Those that increment should increase and the others should remain as zero.
Sensor1_Frequency_OnCapture	SIXOVERFLOWS.004	2.a	Overflow is set to 6 but periods are set to 0. Active is also set to 0. No icapture2 value should be obtained.
RS485_OnRxChar	2START1ENDCHARNORXCOUNT	1.a	Two start characters and an end character is received. The Rxcount is set to 0 and should not increment unless data is received.
	NODATAPRESENT	1.a	After the second start character is received no data is received. Therefore, nothing can be processed.

D. Final Source Code

i. DCS Main Code (Communications.c)

```
/** ###################################################################
**     Filename  : Communications.C
**     Project   : Communications
**     Processor : MC9S12XDP512BCPV
**     Version   : Driver 01.12
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 8/12/2010, 9:49 AM
**     Abstract  :
**         Main module.
**         Here is to be placed user's code.
**     Settings  :
**     Contents  :
**         No public methods
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/
/* MODULE Communications */

/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "RS485.h"
#include "RS485_DE.h"
#include "RS485_RE.h"
#include "LED1.h"
#include "Timer.h"
#include "AS1.h"
#include "Bit1.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
```

```
#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include "math.h"
#include "MBdefs.h"
#include "Communications.h"

const char codeVersion[]="Communications v0.2";
int TimerCounter;
word RS485_RxCount;
RS485_TComData RS485_RxBuffer[64];

int GettingChars=0;
int GettingErrors;
RS485_TComData RS485chr;
RS485_TError RS485err;

bool RS485infoReady;
int retValue;
bool STX;
bool ACPT;
word statusTimer=0;
RS485_TComData RS485_TxBuffer[64];
word RS485_nSent;
word RS485_count;
bool send_to_Commboard=FALSE;
word RS485countholder=0;
char RS485singlecmdBuffer[20];
bool RS485continue=FALSE;
word RS485commMeasureTimer=0;
bool RS485_Present=FALSE;
word MDUstartover=0;
int RS485State=0;
int RS485Count=0;
int comm_error=0;
char *packet;
int pducount;
int pdulen;
int chksum;
char hold[4];
int datalen;
double params[12];
int RS485Tx=0;
char workBuffer[64];
```

```
int j, len, i;
int toMainboard, ntoMainboard;
int nCount;
int firstCount;
char checkSum[12];
int check;
int nCount;
int nCountpacket;
char charCount[8];
int a=0;
bool sendpacket1=FALSE;
bool sendpacket2=FALSE;

float sensor1Freq;
float sensor1ROC;
float sensor2Mag;
float sensor2Phase;
float motorRT;
bool motorUp;
bool motorDown;
bool valve1;
bool valve2;
bool valve1command;
bool valve2command;
bool valve1commandtemp;
bool valve2commandtemp;
bool sensor1LO;
bool sensor2LO;
bool sensor1Decel;
char motorUpdir[3];
char motorDowndir[3];
char valve1dir[5];
char valve2dir[5];
char sensor1LOalarm[3];
char sensor2LOalarm[3];
char sensor1Decelalarm[3];

word screenDelay=0;
AS1_TComData AS1_TxBuffer[80];
AS1_TComData AS1_RxBuffer[80];
word AS1_nSent;
word AS1_count;
word AS1_RxCount=0;
bool escape=FALSE;
```

```
bool backspace=FALSE;
word echo=0;
bool commandReady=FALSE;
bool commandActive=FALSE;
char cmdBuffer[80];
char cmdArgument[20];
bool secondPart=FALSE;

void main(void)
{

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  RS485_DE_ClrVal();
  RS485_RE_ClrVal();

  for(;;)  {

  RS485Comm();
  display();
  }
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/


//****************** function to RS485 Communications *************************
void RS485Comm(void) {

  char checkSum[8];
  int a=0;


//***Received Packet Handling***

if(RS485infoReady==TRUE){
    RS485infoReady=FALSE;

//Obtain the length value by calculating the length from the packet received
    datalen=strlen(RS485_RxBuffer)-4;
```

```
   if(datalen>0){  //if length of data is not correct then don't process the data
//Copy the received length fields into the pdulen variable
   if(NULL!=strncpy(hold, RS485_RxBuffer, 2)) pdulen=strtol(hold, NULL, 10);

//Need to ensure pdulen is not bigger than the RxBuffer and is less than 62 (RS485_RxBuffer is
//an array of size 64)
   if(pdulen> strlen(RS485_RxBuffer)&& pdulen<62){
       RS485_RxCount=0;    //The RxCount is reset because pdulen is not correct
       return;
   }

//Copy the received checksum fields into the chksum variable
   if(NULL!=strncpy(hold, RS485_RxBuffer+datalen+2, 2)) chksum=strtol(hold,NULL,10);

//Place a 0 in the first checksums position so that the string-to-token function will parse the data
//correctly
   RS485_RxBuffer[datalen+2]=0;


   check=0;
//Add up all the decimal values of the data to verify the checksum
   for(j=0;j<pdulen;j++) check=check+RS485_RxBuffer[j+2];
   check=check%100;  //mod 100 truncates the total checksum value to keep the last two digits

   if(check==chksum){    //Does the received checksum match the calculated checksum
      if(datalen==pdulen){   //Does the received length match the calculated length
         pducount=0;  //clear the counter so we are looking for the correct number of data fields

//As long as there is a space in the RxBuffer then proceed
         if(NULL!= (packet=strtok(RS485_RxBuffer+2, " ")){

//Uses string-to-double function to convert the first ascii field to a double
            params[pducount]=strtod(packet, NULL);
            pducount++;  //increment the counter
         }

          while(NULL!=(packet=strtok(NULL, " "))){
             params[pducount]=strtod(packet, NULL);   //converts the remaining fields to double
             pducount++;    //increments the counter
          }

//When the counter reaches 12 (the number of data fields that should be present), the packet will
//be evaluated and ready to be sent to the Hyperterminal
          if(pducount==12){
```

```
//Assigns the eight parameter which is a double and casts it to an boolean variable
            valve1=params[0];
            valve2=params[1];
            sensor1Freq=params[2];

//The floating point variable is checked to ensure it is not too large
            if(sensor1Freq>3276.7) sensor1Freq=3276.7;

//Assigns the second parameter sent to a floating point variable
            sensor1ROC=params[3];

//The floating point variable is checked to ensure it is not too large
            if(sensor1ROC>3.2767) sensor1ROC=3.2767;


//the floating point variable is checked to ensure it is not too small
            if(sensor1ROC<-3.2767) sensor1ROC=-3.2767;

//Assigns the forth parameter sent to a floating point variable
            sensor2Phase=params[4];

//The floating point variable is checked to ensure it is not too large
            if(sensor2Phase>360.0) sensor2Phase=0;
            sensor1LO=(bool)params[5];
            sensor1Decel=(bool)params[6];
            sensor2LO=(bool)params[7];
            motorRT=params[8];

//The floating point variable is checked to ensure it is not too large
            if(motorRT>32.767) motorRT=32.767;
            if(motorRT<-32.767) motorRT=-32.767;
            motorUp=(bool)params[9];
            motorDown=(bool)params[10];
            sensor2Mag=params[11];
          }
        }
    }
}

//The RxCount is reset in case the packet doesn't pass some of the integrity tests
RS485_RxCount=0;
}

 //***END of received packet handling***
```

```
}//*** end RS485Comm function ***


//*************** provide a parse function to execute commands *****************
int parseCommand(char *command) {

   if(strcmp(command,"version")==0) {
      retValue=printf("\x1B[16;16f\x1B[K%s",codeVersion);
      screenDelay=3000;//3 seconds
      return(0);
   }

   if(strcmp(command,"valve1o")==0){
      valve1commandtemp=valve1command;
      valve1command=TRUE;
      sendpacket1=TRUE;
      retValue=printf("\x1B[16;16f\x1B[KValve1 is Opening");
      screenDelay=3000;
      return(0);
   }

   if(strcmp(command,"valve1c")==0){
      valve1commandtemp=valve1command;
      valve1command=FALSE;
      sendpacket1=TRUE;
      retValue=printf("\x1B[16;16f\x1B[KValve1 is Closing");
      screenDelay=3000;
      return(0);
   }

   if(strcmp(command,"valve2o")==0){
      valve2commandtemp=valve2command;
      valve2command=TRUE;
      sendpacket2=TRUE;
      retValue=printf("\x1B[16;16f\x1B[KValve2 is Opening");
      screenDelay=3000;
      return(0);
   }

   if(strcmp(command,"valve2c")==0){
      valve2commandtemp=valve2command;
      valve2command=FALSE;
      sendpacket2=TRUE;
      retValue=printf("\x1B[16;16f\x1B[KValve2 is Closing");
```

```
        screenDelay=3000;
        return(0);
    }

return(1);
}
//*** end of parse function ***

//****************** function to display to serial port *************************
void display(void) {
int a=0;


    if(backspace){
        backspace=FALSE;
        AS1_RxCount-=1;
        if(echo>0) echo-=1;
        retValue=printf("\x1B[1D");
    }

    while(echo!=AS1_RxCount) {
        retValue=AS1_SendChar(AS1_RxBuffer[echo]);
        echo+=1;
    }

    if(commandReady==TRUE){
        commandReady=FALSE;
        commandActive=FALSE;
        secondPart=FALSE;
        echo=0;
        firstCount=0;
        for(j=0; j<AS1_RxCount; j++) {
            if(!secondPart){
                cmdBuffer[j]=(char)AS1_RxBuffer[j];
                firstCount+=1;
                if(cmdBuffer[j]==0x20){
                    cmdBuffer[j]='\0';
                    secondPart=TRUE;
                }
            }else cmdArgument[(j-firstCount)]=(char)AS1_RxBuffer[j];
        }
        retValue=parseCommand(cmdBuffer);
        AS1_RxCount=0;
        if(retValue) printf("\x1B[16;7f\x1B[Kinvalid command");
```

```
  }

  if (sendpacket1==TRUE){
     sendpacket1=FALSE;
     toMainboard = sprintf(workBuffer,"%d %d",valve1command, valve2);
     RS485_TxBuffer[0]=2;
     RS485_TxBuffer[1]=2;
    nCountpacket=sprintf(charCount,"%d",toMainboard);
    if(nCountpacket<2) {
        RS485_TxBuffer[2]=(char)'0';
        RS485_TxBuffer[3]=charCount[0];
    }

   for(j=4; j<(toMainboard+4); j++) RS485_TxBuffer[j]=(RS485_TComData)workBuffer[j-4];
    nCount=0;
    for(j=0; j<toMainboard; j++) nCount=nCount+workBuffer[j];
    ntoMainboard=sprintf(checkSum,"%d",nCount);

    if(nCount>=100 && nCount<1000){
        RS485_TxBuffer[toMainboard+4]=checkSum[1];
        RS485_TxBuffer[toMainboard+5]=checkSum[2];
    }

    RS485_TxBuffer[toMainboard+6]=3;
    RS485_count=(word)toMainboard+7;
    RS485_DE_SetVal();//enable RS485 transmitter
    RS485_RE_SetVal();//disable RS485 receiver
    RS485infoReady=FALSE;
    RS485Tx=RS485_SendBlock(RS485_TxBuffer,RS485_count, &RS485_nSent);

   if(RS485Tx==ERR_TXFULL){
       RS485_count=0;
       RS485_ClearTxBuf();
   }
}

   if (sendpacket2==TRUE){
       sendpacket2=FALSE;
       toMainboard = sprintf(workBuffer,"%d %d",valve1, valve2command);
       RS485_TxBuffer[0]=2;
       RS485_TxBuffer[1]=2;
       nCountpacket=sprintf(charCount,"%d",toMainboard);
       if(nCountpacket<2) {
       RS485_TxBuffer[2]=(char)'0';
```

```
        RS485_TxBuffer[3]=charCount[0];
    } else return;

    for(j=4; j<(toMainboard+4); j++) RS485_TxBuffer[j]=(RS485_TComData)workBuffer[j-4];
    nCount=0;
    for(j=0; j<toMainboard; j++) nCount=nCount+workBuffer[j];
    ntoMainboard=sprintf(checkSum,"%d",nCount);

    if(nCount>=100 && nCount<1000){
        RS485_TxBuffer[toMainboard+4]=checkSum[1];
        RS485_TxBuffer[toMainboard+5]=checkSum[2];
    } else return;

    RS485_TxBuffer[toMainboard+6]=3;
    RS485_count=(word)toMainboard+7;
    RS485_DE_SetVal();//enable RS485 transmitter
    RS485_RE_SetVal();//disable RS485 receiver
    RS485infoReady=FALSE;
    RS485Tx=RS485_SendBlock(RS485_TxBuffer,RS485_count, &RS485_nSent);

    if(RS485Tx==ERR_TXFULL){
        RS485_count=0;
        RS485_ClearTxBuf();
     }

  }

   if(statusTimer!=0) return;
   else statusTimer=1000;

   if(commandActive==TRUE || screenDelay) return;

   retValue=printf("\x1B[1;1f\x1B[KAlarms: Sensor1 %d   Sensor2 %d   Deceleration %d\
                \x1B[3;1f\x1B[KSensor1: Freq %4.1f   Rate of Change %1.4f\
                \x1B[5;1f\x1B[KSensor2: Magnitude %3.2f   Phase %3.1f"\
                ,sensor1LO, sensor2LO, sensor1Decel, sensor1Freq, sensor1ROC, sensor2Mag,
sensor2Phase);

   retValue=printf("\x1B[7;1f\x1B[KMotor: Up %d   Down %d   Run Time %2.3f\
                \x1B[9;1f\x1B[KValves: Valve1 %d   Valve2 %d\
                \x1B[11;1f\x1B[KValve Command: Valve1 %d   Valve2 %d"
                ,motorUp,motorDown, motorRT, valve1, valve2, valve1command,
valve2command);
```

```
   retValue=printf("\x1B[16;1f\x1B[KCmd-> ");//set initial cursor position
}//*** end display function ***

/* END Communications */
/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.97 [03.83]
**     for the Freescale HCS12X series of microcontrollers.
**
** ###################################################################
*/
```

ii. DCS Main Code Header File (Communications.h)

```
/** ###################################################################
**     Filename  : Communications.h
**     Project   : Communications
**     Processor : MC9S12XDP512BCPV
**     Beantype  : none
**     Version   :
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 8/12/2010, 9:49 AM
**     Abstract  :
**             :General delclarations
**
**     Settings  :
**     Contents  :
**
**
** ###################################################################*/

#ifndef __Communications_H
#define __Communications_H
/* MODULE main */

//*** function declaratons ***
void RS485Comm(void);
int parseCommand(char *command);
void display(void);

/*definitions*/
#define RxBufferSize 64
#define TxBufferSize 64

/* END */
#endif /* __Communications_H*/

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.96 [03.76]
**     for the Freescale HCS12X series of microcontrollers.
**
** ###################################################################
*/
```

iii. DCS Interrupt Code (Events.c)

```
/** ###################################################################
**     Filename  : Events.C
**     Project   : Communications
**     Processor : MC9S12XDP512BCPV
**     Beantype  : Events
**     Version   : Driver 01.04
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 8/12/2010, 9:49 AM
**     Abstract  :
**         This is user's event module.
**         Put your event handler code here.
**     Settings  :
**     Contents  :
**         RS485_OnError  - void RS485_OnError(void);
**         RS485_OnRxChar - void RS485_OnRxChar(void);
**         RS485_OnTxChar - void RS485_OnTxChar(void);
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2006
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/
/* MODULE Events */


#include "Cpu.h"
#include "Events.h"
#include "Communications.h"

#pragma CODE_SEG DEFAULT

extern int TimerCounter;
extern word RS485_RxCount;
extern RS485_TComData RS485_RxBuffer[64];
extern RS485_TComData RS485chr;
extern bool RS485infoReady;
extern RS485_TError RS485err;
extern int retValue;
```

```
extern bool STX;
extern bool ACPT;
extern word statusTimer;
extern unsigned int screenDelay;
extern word AS1_RxCount;
AS1_TComData chr;
extern AS1_TComData AS1_RxBuffer[80];
extern bool escape;
extern bool backspace;
extern bool commandReady;
extern bool commandActive;
extern int GettingChars;
extern int GettingErrors;


/*
**
===================================================================
**     Event       :  RS485_OnError (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
**     Description :
**         This event is called when a channel error (not the error
**         returned by a given method) occurs. The errors can be
**         read using <GetError> method.
**         The event is available only when the <Interrupt
**         service/event> property is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
void RS485_OnError(void){
   RS485_RxCount=0;
   STX=FALSE;
   ACPT=FALSE;
   retValue=RS485_GetError(&RS485err);
}
/*
**
===================================================================
**     Event       :  RS485_OnRxChar (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
```

```
**     Description :
**         This event is called after a correct character is
**         received.
**         The event is available only when the <Interrupt
**         service/event> property is enabled and either the
**         <Receiver> property is enabled or the <SCI output mode>
**         property (if supported) is set to Single-wire mode.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void RS485_OnRxChar(void){
//Get characters in the receive buffer if there is something there
   while(RS485_GetCharsInRxBuf()!=0){
   retValue=RS485_RecvChar(&RS485chr);
   if(RS485chr==2){       //if the STX character (2) is received set the STX flag and exit
      if(STX){
         RS485_RxCount=0;  //if the second STX character is received clear the RxCount and exit
         ACPT=TRUE;
      } else{
      STX=TRUE;
      ACPT=FALSE;
   }
   return;
   }else STX=FALSE;            //if there is no STX character then the STX flag should be cleared

   if(ACPT==FALSE) return;

   if(RS485chr==3) {

//If the EOF character (3) is received and the RxCount has indicated that data has been received
then the DCS has established communications and the EOF character is replaced by a 0 in the
RxBuffer
      if(RS485_RxCount>1){
         RS485infoReady=TRUE;
         GettingChars++;
         RS485_RxBuffer[RS485_RxCount]='\0';
         ACPT=FALSE;
      }
      return;    //if the EOF character is received without a packet body; then just exit
   }

//Validating the remaining packet for valid characters (0-9, spaces, and a decimal)
```

```
//If the character received is greater than an ASCII 9 (a value of 57 in decimal) then clear the
//RxCounter and exit
   if(RS485chr>'9'){
      RS485_RxCount=0;
      return;
   }

//If the character received is less than an ASCII 0 (a value of 48 in decimal) and it's not a space
//(a value of 32 in decimal) and it's not a decimal (a value of 46 in decimal) and it's not a minus
//sign (a value of 45 in decimal) then clear the RxCounter and exit
   if(RS485chr<48 && RS485chr!=32 && RS485chr!=46 && RS485chr!=45){
      RS485_RxCount=0;
      return;
   }

//If the character passes all the tests for a valid character then place it in the RxBuffer
   RS485_RxBuffer[RS485_RxCount]=RS485chr;

//Increment the RxCounter as long as it is less than the size of the RxBuffer
   if(RS485_RxCount<sizeof(RS485_RxBuffer)) RS485_RxCount+=1;
   }//end of while
}
/*
**
===================================================================
**     Event      :  Timer_OnInterrupt (module Events)
**
**     From bean   :  Timer [TimerInt]
**     Description :
**         When a timer interrupt occurs this event is called (only
**         when the bean is enabled - "Enable" and the events are
**         enabled - "EnableEvent").
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
void Timer_OnInterrupt(void){
   TimerCounter += 1;

   if(TimerCounter == 500) {//500 = 1/2 sec
      LED1_PutVal(TRUE);
      Bit1_PutVal(TRUE);
   }
```

```
   if(TimerCounter == 1000) {//1000 = 1 sec
      LED1_PutVal(FALSE);
      Bit1_PutVal(FALSE);
      TimerCounter = 0;
   }

    if(screenDelay) screenDelay-=1;
    if(statusTimer) statusTimer -=1;

}
/*
**
============================================================================
**     Event      :  RS485_OnTxComplete (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
**     Description :
**         This event indicates that the transmitter is finished
**         transmitting all data, preamble, and break characters and
**         is idle. It can be used to determine when it is safe to
**         switch a line driver (e.g. in RS-485 applications).
**         The event is available only when both <Interrupt
**         service/event> and <Transmitter> properties are enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void RS485_OnTxComplete(void){
   RS485_DE_ClrVal();//disable MB_RS485 transmitter
   RS485_RE_ClrVal();//enable MB_RS485 receiver
}
/*
**
============================================================================
**     Event      :  AS1_OnRxChar (module Events)
**
**     From bean   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after a correct character is
**         received.
**         The event is available only when the <Interrupt
**         service/event> property is enabled and either the
```

```
**         <Receiver> property is enabled or the <SCI output mode>
**         property (if supported) is set to Single-wire mode.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void AS1_OnRxChar(void){
   commandActive=TRUE;

   while(AS1_GetCharsInRxBuf()!=0){
      retValue=AS1_RecvChar(&chr);

      if(chr==0x08 && AS1_RxCount>0) { //handle backspace
         backspace=TRUE;
      }

      AS1_RxBuffer[AS1_RxCount]=chr;

      if(chr=='\r') {
         commandReady=TRUE;
        AS1_RxBuffer[AS1_RxCount]='\0';
      }

      if(AS1_RxCount<sizeof(AS1_RxBuffer) && chr!=0x08) AS1_RxCount+=1;

   }//end of while
}
/* END Events */
/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.97 [03.83]
**     for the Freescale HCS12X series of microcontrollers.
**
** ###################################################################
*/
```

iv. IMMI Main Code (MIP_LC3081709.c)

```
/** ###################################################################
**     Filename  : MIP_LC3081709.C
**     Project   : IMMI
**     Processor : MC9S12XDP512BCPV
**     Version   : Driver 01.1
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 9/25/2010, 10:41 AM
**     Abstract  :
**         Main module.
**         Here is to be placed user's code.
**     Settings  :
**     Contents  :
**         No public methods
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2005
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/
/* MODULE MIP_LC3081709 */

/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "LED1.h"
#include "Timer.h"
#include "RS485.h"
#include "RS485_RE.h"
#include "RS485_DE.h"
#include "AD1.h"
#include "Sensor1_Frequency.h"
#include "Sensor2.h"
#include "Sensor1.h"
#include "Open_Valve1.h"
#include "Close_Valve1.h"
#include "Open_Valve2.h"
#include "Close_Valve2.h"
#include "Valve1_Switch.h"
```

```
#include "Valve2_Switch.h"
#include "up_rising.h"
#include "up_falling.h"
#include "down_rising.h"
#include "Alarm_LED.h"
#include "down_falling.h"

/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "stdlib.h"
#include "stdio.h"
#include "math.h"
#include "MBdefs.h"
#include "MIP_LC3081709.h"
#include "string.h"

/*declarations*/
int TimerCounter;
word RS485_RxCount;
RS485_TComData RS485_RxBuffer[20];
bool RS485infoReady;
int retValue;
bool STX;
bool ACPT;
int statusTimer=0;
RS485_TComData RS485_TxBuffer[80];
word RS485_nSent;
word RS485_count;
bool send_to_Commboard=FALSE;
int RS485State=0;
int RS485Count=0;
char *packet;
int pducount;
int pdulen;
int chksum;
char hold[4];
int datalen;
double params[4];
int RS485Tx=0;
char workBuffer[74];
int j, i;
```

```
char checkSum[12];
int check;
int nCount;
int nCountpacket;
char charCount[8];
int toCommboard, ntoCommboard;


//Valve operation
bool Valve1Sw=FALSE;
bool Valve2Sw=FALSE;
int Valve1Count=0;
int Valve2Count=0;
bool Valve1Position=FALSE;
bool Valve2Position=FALSE;
bool switcheswork=TRUE;
bool Valve1Command=FALSE;
bool Valve2Command=FALSE;
bool Valve1commandedposition;
bool Valve2commandedposition;
word Valve1Timer=0;
word Valve1TimeOut=0;
int Valve1State=0;
int Valve1Valid=0;
word Valve2Timer=0;
word Valve2TimeOut=0;
int Valve2State=0;
int Valve2Valid=0;

//Motor interval time
word uptimer=0;
word downtimer=0;
word upTimerCalc=0;
word downTimerCalc=0;
float upSec=0;
float downSec=0;
bool up=FALSE;
bool down=FALSE;
float motorRT=0;
int motorUp=0;
int motorDown=0;

//Sensor1 vs. Sensor2 Phase measurement variables
word sample_counter_fixed=16;
```

```
bool phaseStart=TRUE;
word phaseMeasureTimer=0;
word sensor1pulse=0;
word sensor1OverFlows=0;
word period=0;
word sensor1ExactOverFlows=0;
word icapture3;
word icapture4;
word icapture5;
bool sensor2armed=FALSE;
bool sensor1=FALSE;
bool sensor2=FALSE;
word sensor2ExactOverFlows=0;
word phase_sample_counter=0;
float sensor2count;
float sensor1count;
float sensor1_sensor2_Fraction;
float Bin_1_Sum;
float Bin_2_Sum;
float Bin_3_Sum;
float Bin_4_Sum;
float Bin_5_Sum;
float Bin_1_Counter;
float Bin_2_Counter;
float Bin_3_Counter;
float Bin_4_Counter;
float Bin_5_Counter;
float Total_Fraction;
float sensor1_sensor2_Degree;
float Degree1;
float whole_number;
float sensor1_sensor2_phase;
bool phase_value=FALSE;

//ADC
bool adValueReady1=FALSE;
word adValue1;
byte channel1=0;
float sensor2Mag=0;

//Frequency measurement variables
word nPeriods=0;
word nOverFlows=0;
word icapture1;
```

```
word icapture2;
bool pDone=FALSE;
bool firstPeriod=TRUE;
word freqMeasureTimer=0;
float count;
float f1=0;
word realoverflow=0;
word realperiod=0;
bool active=TRUE;
float sensor1Freq;

//Rate of Change variables
float Rate_of_Change_array[63];
float array_shift[63];
float Rate_of_Change_orig;
bool min_past=FALSE;
int rate_of_change_counter=0;
word ROCtimer=1000;//start at 1 second
bool freqready=FALSE;
float sensor1ROC;

//Decel variables
float Decel_orig;
float Decel_product;
bool sec_past=FALSE;
int decel_counter=0;
word DecelTimer=0;//start at 0
bool decelactivated=FALSE;
bool decelhappened=FALSE;
bool sensor1Decel=FALSE;

//Alarms
bool LOS1=FALSE;
bool LOS2=FALSE;
bool Decelalarm=FALSE;
int isocount=0;
word LOS1_Timer=100;
word LOS2_Timer=100;
word AlarmTimer=0;
int AlarmState=0;
bool iso=FALSE;


void main(void)
```

```
{
/*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
/*** End of Processor Expert internal initialization.                    ***/
  RS485_DE_SetVal();
  RS485_RE_SetVal();
  retValue=AD1_MeasureChan(TRUE, channel1);
  retValue=AD1_GetChanValue(channel1, &adValue1);
  TSCR1_TEN=1;
  TSCR1_TFFCA=0;//normal interrupt clearing
  TIE_C3I=1;//enable capture 3 interrupt
  TIE_C1I=1;//enable capture 1 interrupt
  TIE_C2I=1;//enable capture 2 interrupt
  TSCR2_TOI=1;//enable timer overflow interrupt
  TCTL4_EDG3A=1;//capture on rising edges only using register 3
  TCTL4_EDG3B=0;; //Disable and set to 0xFFFF
  TCTL4_EDG1A=1;//capture on rising edges only using register 1
  TCTL4_EDG1B=0;; //Disable and set to 0xFFFF
  TCTL4_EDG2A=1;//capture on rising edges only using register 2
  TCTL4_EDG2B=0;; //Disable and set to 0xFFFF
  TFLG2_TOF = 1;                    /* Reset overflow interrupt request flag */
  freqMeasureTimer=FREQ_TIME_OUT;//set maximum time for measurement
  rate_of_change_counter=0;
  min_past=FALSE;
  sec_past=FALSE;
  switcheswork=TRUE;
  Valve1Position=FALSE;
  Valve2Position=FALSE;
  Close_Valve1_ClrVal();
  Close_Valve2_ClrVal();

  for(;;)  {
    RS485Comm();
    Switches();
    FrequencyActivation();
    RateofChange();
    PhaseMeasure();
    Alarms();
    MotorTimer();
    ADCMeasurements();
  }

/*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
```

```
/*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/

//**********************Functions start here**********************************

//****************** function to RS485 Communications *************************
void RS485Comm(void) {

//***Received Packet Handling***

   if(RS485infoReady==TRUE){
      RS485infoReady=FALSE;

//Obtain the length value by calculating the length from the packet received
      datalen=strlen(RS485_RxBuffer)-4;
      if(datalen>0){  //if length of data is not correct then don't process the data

//Copy the received length fields into the pdulen variable
         if(NULL!=strncpy(hold, RS485_RxBuffer, 2)) pdulen=strtol(hold, NULL, 10);

//Need to ensure pdulen is not bigger than the RxBuffer and it is less than 62 (RS485_RxBuffer
//is an array of size 64)
         if(pdulen> (datalen+4) && pdulen<18){
            RS485_RxCount=0;    //The RxCount is reset because pdulen is not correct
            return;
         }

//Copy the received checksum fields into the chksum variable
         if(NULL!=strncpy(hold, RS485_RxBuffer+datalen+2, 2))chksum=strtol(hold,NULL,10);

//Place a 0 in the first checksums position so the string-to-token function will parse the data
//correctly
         RS485_RxBuffer[datalen+2]=0;
         check=0;

//Adding up all the decimal values of the data to be able to verify the checksum
         for(j=0;j<pdulen;j++) check=check+RS485_RxBuffer[j+2];

//Mod 100 truncates the total checksum value to just keep the last two digits
         check=check%100;
         if(check==chksum){    //Does the received checksum match the calculated checksum
            if(datalen==pdulen){   //Does the received length match the calculated length

//Clear the counter so that we are looking for the correct number of data fields
```

```
                pducount=0;

//As long as there is a space in the RxBuffer then proceed
                if(NULL!= (packet=strtok(RS485_RxBuffer+2, " "))){

//Use string-to-double function to convert the first ascii field to a double
                    params[pducount]=strtod(packet, NULL);
                    pducount++;  //increment the counter
                }

                while(NULL!=(packet=strtok(NULL, " "))){

//Convert the remaining fields to double
                    params[pducount]=strtod(packet, NULL);
                    pducount++;    //increments the counter
                 }

//When the counter reaches 2 (the number of data fields that should be present), the packet will
//be evaluated and ready to be sent to the DCS
                 if(pducount==2){

//Assign the first parameter sent to a floating point variable
                     Valve1commandedposition=(bool)params[0];

//The floating point variable should be checked to ensure it is not too large
                     if(Valve1commandedposition!= Valve1Position) Valve1Command=TRUE;

//Assign the second parameter sent to a floating point variable
                     Valve2commandedposition=(bool)params[1];

//The floating point variable should be checked to ensure it is not too large
                     if(Valve2commandedposition!=Valve2Position) Valve2Command=TRUE;

                 }
             }
         }
     }

//The RxCount should be reset in case the packet doesn't pass some of the integrity tests
        RS485_RxCount=0;
    }

//***END of received packet handling***
```

```
    if(statusTimer!=0) return;
    else statusTimer=1000;

    if(motorRT>0) motorUp=1;
    else motorUp=0;
    if(motorRT<0) motorDown=1;
    else motorDown=0;
    if(Valve1Position<0 ||Valve1Position>1){
        Valve1Position=FALSE;    //Change position of valve1 to CLOSE
        Close_Valve1_ClrVal();
        Open_Valve1_SetVal();     //Turn off Open LED
        Valve1Sw=FALSE;           //Switch activity is complete
        Valve1Count=0;
    }
    if(Valve2Position<0 ||Valve2Position>1){
        Valve2Position=FALSE;   //Change position of valve2 to CLOSE
        Close_Valve2_ClrVal();
        Open_Valve2_SetVal();    //Turn off Open LED
        Valve2Sw=FALSE;          //Switch activity is complete
        Valve2Count=0;
    }
    toCommboard = sprintf(workBuffer,"%d %d %4.1f %1.4f %4.1f %d %d %d %2.3f %d %d
%3.2f",Valve1Position, Valve2Position, sensor1Freq, sensor1ROC, sensor1_sensor2_phase,
LOS1, sensor1Decel, LOS2, motorRT, motorUp, motorDown, sensor2Mag);
    RS485_TxBuffer[0]=2;
    RS485_TxBuffer[1]=2;
    nCountpacket=sprintf(charCount,"%d",toCommboard);
    if(nCountpacket>=2) {
        RS485_TxBuffer[2]=charCount[0];
        RS485_TxBuffer[3]=charCount[1];
    } else return;

    for(j=4; j<(toCommboard+4); j++) RS485_TxBuffer[j]=(RS485_TComData)workBuffer[j-4];
    nCount=0;
    for(j=0; j<toCommboard; j++) nCount=nCount+workBuffer[j];
    ntoCommboard=sprintf(checkSum,"%d",nCount);

    if(nCount>=1000){
        RS485_TxBuffer[toCommboard+4]=checkSum[2];
        RS485_TxBuffer[toCommboard+5]=checkSum[3];
    } else return;

    RS485_TxBuffer[toCommboard+6]=3;
    RS485_count=(word)toCommboard+7;
```

```
   RS485_DE_SetVal();//enable RS485 transmitter
   RS485_RE_SetVal();//disable RS485 receiver
   RS485Tx=RS485_SendBlock(RS485_TxBuffer,RS485_count, &RS485_nSent);

   if(RS485Tx==ERR_TXFULL){
      RS485_count=0;
      RS485_ClearTxBuf();
   }
}//*** end RS485Comm function ***

//*************** function for switches ******************
void Switches(void){

   if(switcheswork==TRUE){
      if(Valve1Sw==TRUE){
         if(Valve1Position==TRUE){   //position of valve1 (OPEN)
            Valve1Position=FALSE;     //Change position of valve1 to CLOSE
            Close_Valve1_ClrVal();
            Open_Valve1_SetVal();      //Turn off Open LED
            Valve1Sw=FALSE;            //Switch activity is complete
            Valve1Count=0;
         } else{
            if(Valve1Count==1) Valve1TimeOut=5000;    //If button is pressed once then set timer

//If the valve is in the CLOSED position and you waited 3 seconds but 5 seconds has not elapsed
//and it is your second press
            if(Valve1Position==FALSE && Valve1TimeOut<2000 && Valve1TimeOut!=0 &&
Valve1Count==2){
               Valve1Valid=1;

//If the valve is in the CLOSED position and didn't wait 3 seconds and it is your second press
            } else if(Valve1Position==FALSE && Valve1TimeOut>2000 && Valve1Count==2){
               Valve1Valid=2;

//If the valve is in the CLOSED position and 5 seconds has elapsed
            } else if(Valve1Position==FALSE && Valve1TimeOut==0){
               Valve1Valid=2;

//Otherwise, wait for another press of the button
            } else{
               Valve1Valid=0;
               Valve1Sw=FALSE;
            }
```

```
//If buttons were operated successfully then do the following
            if(Valve1Valid==1){
               Valve1Sw=FALSE;
               Valve1Position=TRUE;    //position of the valve is now OPEN
               Open_Valve1_ClrVal();   //Open LED is turned on
               Close_Valve1_SetVal();  //Closed LED is turned off
               Valve1Count=0;
               Valve1Valid=0;
            } else if(Valve1Valid==2)   {
               Valve1Sw=FALSE;
               Valve1Position=FALSE;    //position of the valve is now CLOSED
               Open_Valve1_SetVal();     //Open LED is turned off
               Close_Valve1_ClrVal();    //Closed LED is turned on
               Valve1Count=0;
               Valve1Valid=0;
            }
         }
      }

      if(Valve2Sw==TRUE){
         if(Valve2Position==TRUE){   //position of valve2 (OPEN)
            Valve2Position=FALSE;     //Change position of valve2 to CLOSE
            Close_Valve2_ClrVal();      //Turn on Purge Closed LED
            Open_Valve2_SetVal();      //Turn off Purge Open LED
          Valve2Sw=FALSE;              //Switch activity is complete
          Valve2Count=0;
       } else{
          if(Valve2Count==1) Valve2TimeOut=5000;    //If button is pressed once then set timer

//If the valve is in the CLOSED position and waited 3 seconds but 5 seconds has not elapsed and
//it is your second press
          if(Valve2Position==FALSE && Valve2TimeOut<2000 && Valve2TimeOut!=0 &&
Valve2Count==2){
             Valve2Valid=1;

//If the valve is in the CLOSED position and didn't wait 3 seconds and it is your second press
          } else if(Valve2Position==FALSE && Valve2TimeOut>2000 && Valve2Count==2){
             Valve2Valid=2;

//If the valve is in the CLOSED position and 5 seconds has elapsed
          } else if(Valve2Position==FALSE && Valve2TimeOut==0){
             Valve2Valid=2;

//Otherwise, wait for another press of the button
```

```
            } else{
               Valve2Valid=0;
               Valve2Sw=FALSE;
            }

//If buttons were operated successfully then do the following
            if(Valve2Valid==1){
               Valve2Sw=FALSE;
               Valve2Position=TRUE;     //position of the valve is now OPEN
               Open_Valve2_ClrVal();    //Open LED is turned on
               Close_Valve2_SetVal();   //Closed LED is turned off
               Valve2Count=0;
               Valve2Valid=0;
            } else if(Valve2Valid==2)   {
               Valve2Sw=FALSE;
               Valve2Position=FALSE;   //position of the valve is now CLOSED
               Open_Valve2_SetVal();    //Open LED is turned off
               Close_Valve2_ClrVal();   //Closed LED is turned on
               Valve2Count=0;
               Valve2Valid=0;
            }
         }
      }

//If the valve is in the CLOSED position and the button has been pressed to move the valve
//OPEN and the blink timer has expired, do the following:
      if(Valve1Position==FALSE && Valve1Count>0 && Valve1Timer==0){
         Valve1Timer=500;       //set the blink timer
         Valve1State=~Valve1State;       //toggle the state
         if(Valve1State) Open_Valve1_ClrVal(); //if the state is 1 then turn off the Open LED
         else Open_Valve1_SetVal();      //if the state is 0 then turn on the Open LED
      }

//If the valve is in the CLOSED position and the button has been pressed to move the valve
//OPEN and the blink timer has expired do the following:
      if(Valve2Position==FALSE && Valve2Count>0 && Valve2Timer==0){
         Valve2Timer=500;
         Valve2State=~Valve2State;
         if(Valve2State) Open_Valve2_ClrVal();
         else Open_Valve2_SetVal();
      }

//If a DCS command is sent to CLOSE the valve and IMMI has granted permission for that to
//occur then do the following:
```

```
    if(Valve1Command==TRUE && Valve1commandedposition==FALSE){
       Valve1Position=FALSE;      //position of the valve is now CLOSED
       Open_Valve1_SetVal();      //Open LED is turned off
       Close_Valve1_ClrVal();     //Closed LED is turned on
       Valve1Command=FALSE;

//If a DCS command is sent to OPEN the valve and IMMI has granted permission for that to
//occur then do the following:
    } else if(Valve1Command==TRUE && Valve1commandedposition==TRUE){
       Valve1Position=TRUE;      //position of the valve is now OPEN
       Open_Valve1_ClrVal();     //Open LED is turned on
       Close_Valve1_SetVal();    //Closed LED is turned off
       Valve1Command=FALSE;
    }

//If a DCS command is sent to CLOSE the valve and IMMI has granted permission for that to
//occur then do the following:
    if(Valve2Command==TRUE && Valve2commandedposition==FALSE){
       Valve2Position=FALSE;      //position of the valve is now CLOSED
       Open_Valve2_SetVal();      //Open LED is turned off
       Close_Valve2_ClrVal();     //Closed LED is turned on
       Valve2Command=FALSE;

//If a DCS command is sent to OPEN the valve and IMMI has granted permission for that to
//occur then do the following:
    } else if(Valve2Command==TRUE && Valve2commandedposition==TRUE){
       Valve2Position=TRUE;         //position of the valve is now OPEN
       Open_Valve2_ClrVal();     //Open LED is turned on
       Close_Valve2_SetVal();   //Closed LED is turned off
       Valve2Command=FALSE;
    }
  } else {
    Valve1Sw=FALSE;
    Valve2Sw=FALSE;
    Valve1Count=0;
    Valve2Count=0;
  }
} //*** end of function for switches ***

//***************** update frequency measurement function **********************
float FrequencyMeasurement(void) {

  count=(realoverflow*65535)+icapture2-icapture1;
  f1=((BUSCLOCK/PRESCALE)*(realperiod-1))/count;
```

```
   return(f1);

}//*** end of update frequency measurement function ***

//********** frequency measurement function ************************
void FrequencyActivation(void) {

   if(firstPeriod==TRUE || nOverFlows==8){
      if(pDone==FALSE && nPeriods>=2){
         nPeriods=0;
         nOverFlows=0;
         firstPeriod=FALSE;
         active=TRUE;
         freqMeasureTimer=FREQ_TIME_OUT;//set maximum time for measurement
         return;
      }
   }

   if(pDone==TRUE){
      sensor1Freq=FrequencyMeasurement();
      if(sensor1Freq>9999) sensor1Freq=9999;
      if(sensor1Freq<0) sensor1Freq=0;
      pDone=FALSE;
      freqready=TRUE;
      firstPeriod=TRUE;
      return;
   }

   if(LOS1==TRUE){
      sensor1Freq=0.0;
      freqready=TRUE;
      return;
   }

   if(freqMeasureTimer==0){
      firstPeriod=TRUE;//force restart
      pDone=FALSE;
      sensor1Freq=0.0;
      freqready=TRUE;
      return;
   }
}//*** end of frequency measurement function ***

//******************** Rate of Change function ************************
```

```
void RateofChange(void) {

   if(freqready){//wait for Rate of Change Timer to expire
      if(sensor1Freq==0) {
         if(ROCtimer!=0) return;
      }

      Rate_of_Change_array[0]=sensor1Freq;
      for(i=61;i>-1;i--){
         array_shift[i+1]=Rate_of_Change_array[i];
         Rate_of_Change_array[i+1]=array_shift[i+1];
      }

      ++rate_of_change_counter;
      if(rate_of_change_counter==11||sec_past==TRUE){
         sec_past=TRUE;
         Decel_orig=(array_shift[11]- array_shift[1])/10;
         Decel_product=sensor1Freq*Decel_orig;
         if(Decel_product>=DECELTRIP && decelhappened==FALSE){
            decel_counter++;
            Decelalarm=TRUE;
            sensor1Decel=TRUE;
            iso=TRUE;
           decelhappened=TRUE;
           if(decel_counter==1) DecelTimer=7500; //at least 7 second hold up timer for alarm
           Alarms();
         } else if(Decel_product>=DECELTRIP && decelhappened==TRUE){
           decel_counter++;
           Decelalarm=TRUE;
           if(decel_counter==1) DecelTimer=7500; //at least 7 second hold up timer for alarm
           Alarms();
         } else if(Decel_product<DECELTRIP){
           Decelalarm=FALSE;
           decelhappened=FALSE;
         }
      }

      if(rate_of_change_counter==61||min_past==TRUE){
         min_past=TRUE;

//Don't need to divide by 60 since each element represents 1 second
         Rate_of_Change_orig=(array_shift[1]- array_shift[61]);
         sensor1ROC=Rate_of_Change_orig;
         if(sensor1ROC>=10) sensor1ROC=9.9999;
```

```
        if(sensor1ROC<=-10) sensor1ROC=-9.9999;
        rate_of_change_counter=65;
      }
      freqready=FALSE;
      ROCtimer=1000;
      return;
   }
}

//********** DRO Phase measurement function ************************
void PhaseMeasure(void) {

   if(phaseStart==TRUE){
      phaseMeasureTimer=PHASE_TIME_OUT;
      sensor1pulse=0;
      sensor1OverFlows=0;
      sensor1=FALSE;
      sensor2armed=FALSE;
      sensor2=FALSE;
      phaseStart=FALSE;
      return;
   }

//If sample is less than 16 samples the capture for sensor1 and sensor2 are enabled
   if(phase_sample_counter<sample_counter_fixed){
      if(sensor1==TRUE){     //If we have received 3 sensor1 signals then calculation begins
         sensor1=FALSE;
         phase_sample_counter+=1;  //increase sample number

//Determines the sensor1 count
         sensor1count=(icapture5-icapture3)+(sensor1ExactOverFlows*65535);
         sensor2count=(icapture4-icapture3)+(sensor2ExactOverFlows*65535);

         if(sensor2!=TRUE){
            sensor2count=sensor1count;
         }
         sensor2=FALSE;
         sensor1_sensor2_Fraction=sensor2count/sensor1count;
         if(sensor1_sensor2_Fraction>=0 && sensor1_sensor2_Fraction<=0.125){
            Bin_1_Sum=Bin_1_Sum+sensor1_sensor2_Fraction;
            Bin_5_Sum=Bin_5_Sum+sensor1_sensor2_Fraction+0.500;
            Bin_1_Counter=Bin_1_Counter+1;
            Bin_5_Counter=Bin_5_Counter+1;
         } else if(sensor1_sensor2_Fraction>0.125 && sensor1_sensor2_Fraction<=0.250){
```

```
            Bin_2_Sum=Bin_2_Sum+sensor1_sensor2_Fraction;
            Bin_2_Counter=Bin_2_Counter+1;
         } else if(sensor1_sensor2_Fraction>0.250 && sensor1_sensor2_Fraction<=0.375){
            Bin_3_Sum=Bin_3_Sum+sensor1_sensor2_Fraction;
            Bin_3_Counter=Bin_3_Counter+1;
         } else if(sensor1_sensor2_Fraction>0.375 && sensor1_sensor2_Fraction<=0.500){
            Bin_4_Sum=Bin_4_Sum+sensor1_sensor2_Fraction;
            Bin_4_Counter=Bin_4_Counter+1;
         } else if(sensor1_sensor2_Fraction>0.500 && sensor1_sensor2_Fraction<=0.625){
            Bin_5_Sum=Bin_5_Sum+sensor1_sensor2_Fraction;
            Bin_5_Counter=Bin_5_Counter+1;
         } else if(sensor1_sensor2_Fraction>0.625){
            sample_counter_fixed=sample_counter_fixed-1;
            phase_sample_counter=phase_sample_counter-1;
         }

         phaseStart=TRUE;
      }
      phaseMeasureTimer=PHASE_TIME_OUT; //start time-out counter over
   }

   if(phase_sample_counter==sample_counter_fixed){
      phase_sample_counter=0;
      if((Bin_1_Counter+Bin_2_Counter)>0){
         Total_Fraction=Bin_1_Sum+Bin_2_Sum;
         sample_counter_fixed=(word)(Bin_1_Counter+Bin_2_Counter);
      }
      if((Bin_2_Counter+Bin_3_Counter)>(Bin_1_Counter+Bin_2_Counter)){
         Total_Fraction=Bin_2_Sum+Bin_3_Sum;
         sample_counter_fixed=(word)(Bin_2_Counter+Bin_3_Counter);
      }
      if(((Bin_3_Counter+Bin_4_Counter)>(Bin_2_Counter+Bin_3_Counter)) &&
((Bin_3_Counter+Bin_4_Counter)>(Bin_1_Counter+Bin_2_Counter))){
         Total_Fraction=Bin_3_Sum+Bin_4_Sum;
         sample_counter_fixed=(word)(Bin_3_Counter+Bin_4_Counter);
      }
      if(((Bin_4_Counter+Bin_5_Counter)>(Bin_3_Counter+Bin_4_Counter)) &&
((Bin_4_Counter+Bin_5_Counter)>(Bin_2_Counter+Bin_3_Counter)) &&
((Bin_4_Counter+Bin_5_Counter)>(Bin_1_Counter+Bin_2_Counter))){
         Total_Fraction=Bin_4_Sum+Bin_5_Sum;
         sample_counter_fixed=(word)(Bin_4_Counter+Bin_5_Counter);
      }
      if(sample_counter_fixed<4) phase_value=FALSE;
      sensor1_sensor2_Degree=Total_Fraction*(720.00/sample_counter_fixed);
```

```
      if(sensor1_sensor2_Degree>360.00) sensor1_sensor2_Degree=sensor1_sensor2_Degree-
360.00;
      Degree1=modff(sensor1_sensor2_Degree,&whole_number);
      if(whole_number==0){
         if(Degree1<0.050){
            sensor1_sensor2_Degree=360.0;
         } else if(Degree1>0.050 && Degree1<0.1) sensor1_sensor2_Degree=0.1;
      }

      if(phase_value==FALSE){
         phase_value=TRUE;
         sensor1_sensor2_phase=0;
      } else{
         sensor1_sensor2_phase=sensor1_sensor2_Degree;
      }

      Bin_1_Sum=0;
      Bin_2_Sum=0;
      Bin_3_Sum=0;
      Bin_4_Sum=0;
      Bin_5_Sum=0;
      Bin_1_Counter=0;
      Bin_2_Counter=0;
      Bin_3_Counter=0;
      Bin_4_Counter=0;
      Bin_5_Counter=0;
      sensor1_sensor2_Fraction=0;
      sample_counter_fixed=16;
   }

   if(phaseMeasureTimer==0){
      phaseStart=TRUE;//force restart
      sensor1=FALSE;
      sensor2=FALSE;
      sensor1_sensor2_phase=0;
      return;
   }
   return;

}//*** end of DRO Phase measurement function ***

//**************** function for Alarms ******************
void Alarms(void){
```

```
   if((LOS1==TRUE && LOS2==TRUE) || (LOS2==TRUE && Decelalarm==TRUE))  {
      switcheswork=FALSE;

//Clear command from DCS in case process valve is commanded to open
      Valve1Command=FALSE;

//Clear command from DCS in case purge valve is commanded to open
      Valve2Command=FALSE;
      isolate();
   } else switcheswork=TRUE;

   if(iso==TRUE){
      iso=FALSE;
      isolate();
      isocount++;
   }

   if(LOS1==TRUE){
      if(LOS1_Timer!=0){
         LOS1=FALSE;
      }
   }

   if(LOS2==TRUE){
      if(LOS2_Timer !=0){
         LOS2=FALSE;
      }
   }

   if(Decelalarm && DecelTimer!=0){
      sensor1Decel=TRUE;
      decelactivated=TRUE;
   } else if(decelactivated && Decelalarm==FALSE){
      Decelalarm=FALSE;
      sensor1Decel=FALSE;
      decel_counter=0;
      decelactivated=FALSE;
   } else if(Decelalarm && decelactivated){
      sensor1Decel=TRUE;
   } else if(Decelalarm==FALSE) {
      Decelalarm=FALSE;
      sensor1Decel=FALSE;
      decel_counter=0;
   }
```

```
   if((LOS1==TRUE && LOS2==TRUE) || (LOS2==TRUE && Decelalarm==TRUE)){
      if(AlarmTimer==0){
         AlarmTimer=100;
         AlarmState=~AlarmState;
         if(AlarmState) Alarm_LED_ClrVal();
         else Alarm_LED_SetVal();
      }
   } else if(LOS1 || LOS2 ||   sensor1Decel==TRUE) Alarm_LED_ClrVal();
   else Alarm_LED_SetVal();

   if(LOS1<0 || LOS1>1) LOS1=TRUE;
   if(LOS2<0 || LOS2>1) LOS2=TRUE;
   if(sensor1Decel<0 || sensor1Decel>1) sensor1Decel=FALSE;

}  //*** end of function for alarms ***

//**************** function for Isolation ******************
void isolate(void){

   Valve1Position=FALSE;      //position of the valve is now CLOSED
   Open_Valve1_SetVal();      //Open LED is turned off
   Close_Valve1_ClrVal();     //Closed LED is turned on
   Valve1Command=FALSE;
   Valve2Position=FALSE;      //position of the valve is now CLOSED
   Open_Valve2_SetVal();      //Open LED is turned off
   Close_Valve2_ClrVal();     //Closed LED is turned on
   Valve2Command=FALSE;

}   //*** end of function for isolation ***

//**************** function measuring time a motor is operational ******************
void MotorTimer(void){

   if(up==TRUE){
      upSec=(float)(upTimerCalc/1000.00);
      motorRT=upSec;
      if(motorRT>=100) motorRT=99.999;
      if(motorRT<0) motorRT=0;
      up=FALSE;
   }

   if(down==TRUE){
      downSec=(float)(downTimerCalc/1000.00);
```

```
      motorRT=downSec*(-1);
      if(motorRT<=-100) motorRT=-99.999;
      if(motorRT>0) motorRT=0;
      down=FALSE;
   }

} //*** end of function measuring time a motor is operational ***

//****************** function to obtain ADC Values *************************
void ADCMeasurements(void){

   if (adValueReady1==TRUE){
      if(AD1_GetChanValue(channel1, &adValue1)==ERR_OK){
         switch(channel1){
            case 0:
            if(phase_value==TRUE){
               sensor2Mag=(adValue1);
               if(sensor2Mag>=1000) sensor2Mag=999.99;
               if(sensor2Mag<0) sensor2Mag=0;
            }
            break;
         }

         adValueReady1=FALSE;
         channel1++;
         if (channel1>0) channel1=0;
         retValue=AD1_MeasureChan(FALSE, channel1);
      }
   }

}//*** end of function to obtain ADC values ***

/* END MIP_LC3081709 */
/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.96 [03.76]
**     for the Freescale HCS12X series of microcontrollers.
**
** ###################################################################
*/
```

v. IMMI Main Code Header File (MIP_LC3081709.h)

```
/** ###################################################################
**     Filename  : MIP_LC3081709.h
**     Project   : IMMI
**     Processor : MC9S12XDP512BCPV
**     Beantype  : none
**     Version   :
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 9/25/2010, 10:41 AM
**     Abstract  :
**             :General delclarations
**
**     Settings  :
**     Contents  :
**
**
** ###################################################################*/


#ifndef __Slave_H
#define __Slave_H
/* MODULE main */


//*** function declaratons ***
void Switches(void);
void RS485Comm(void);
float FrequencyMeasurement(void);
void FrequencyActivation(void);
void RateofChange(void);
void PhaseMeasure(void);
void Alarms(void);
void isolate(void);
void MotorTimer(void);
void ADCMeasurements(void);

/*definitions*/
#define RxBufferSize 64
#define TxBufferSize 64
#define HOLDUP 100
#define PHASE_TIME_OUT 8000 //1000=1 second
#define SENSOR2SCALED 5.859 //DRO scaling
```

```
#define BUSCLOCK 8000000 //oscillator/2
#define PRESCALE 16
#define FREQ_TIME_OUT 8000 //1000=1 second
#define DECELTRIP 50.00 //Decel trip point

/* END */
#endif /* __Slave_H*/

/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.96 [03.76]
**     for the Freescale HCS12X series of microcontrollers.
**
** ###################################################################
*/
```

vi. IMMI Interrupt Code (Events.c)

```
/** ###################################################################
**     Filename  : Events.C
**     Project   : IMMI
**     Processor : MC9S12XDP512BCPV
**     Beantype  : Events
**     Version   : Driver 01.04
**     Compiler  : CodeWarrior HCS12X C Compiler
**     Date/Time : 9/25/2010, 10:41 AM
**     Abstract  :
**         This is user's event module.  Put your event handler code here.
**     Settings  :
**     Contents  :
**         MB_RS485_OnError     - void MB_RS485_OnError(void);
**         MB_RS485_OnRxChar    - void MB_RS485_OnRxChar(void);
**         MB_RS485_OnTxChar    - void MB_RS485_OnTxChar(void);
**         MB_RS485_OnFullRxBuf - void MB_RS485_OnFullRxBuf(void);
**         MB_RS485_OnFreeTxBuf - void MB_RS485_OnFreeTxBuf(void);
**         Timer_OnInterrupt   - void Timer_OnInterrupt(void);
**         AS1_OnError       - void AS1_OnError(void);
**         AS1_OnRxChar      - void AS1_OnRxChar(void);
**         AS1_OnTxChar      - void AS1_OnTxChar(void);
**         AS1_OnFullRxBuf   - void AS1_OnFullRxBuf(void);
**         AS1_OnFreeTxBuf   - void AS1_OnFreeTxBuf(void);
**
**     (c) Copyright UNIS, spol. s r.o. 1997-2005
**     UNIS, spol. s r.o.
**     Jundrovska 33
**     624 00 Brno
**     Czech Republic
**     http      : www.processorexpert.com
**     mail      : info@processorexpert.com
** ###################################################################*/
/* MODULE Events */

#include "Cpu.h"
#include "Events.h"
#include "MIP_LC3081709.h"
#pragma CODE_SEG DEFAULT

extern int TimerCounter;
extern word RS485_RxCount;
```

```
extern RS485_TComData RS485_RxBuffer[64];
RS485_TComData RS485chr;
extern bool RS485infoReady;
RS485_TError RS485err;
extern int retValue;
extern bool STX;
extern bool ACPT;
extern int statusTimer;

//Motor Run Time
extern word uptimer;
extern word downtimer;
extern word upTimerCalc;
extern word downTimerCalc;
extern bool up;
extern bool down;

//Sensor1 vs. Sensor2 Phase measurement variables
extern bool phaseStart;
extern word sensor1pulse;
extern word sensor1OverFlows;
extern word period;
extern word sensor1ExactOverFlows;
extern word icapture3;
extern word icapture4;
extern word icapture5;
extern bool sensor2armed;
extern bool sensor1;
extern bool sensor2;
extern word sensor2ExactOverFlows;
int CaptureValueSensor1;
int *pCaptureValueSensor1=&CaptureValueSensor1;
int CaptureValueSensor2;
int *pCaptureValueSensor2=&CaptureValueSensor2;

//ADC
extern bool adValueReady1;
extern word adValue1;
extern byte channel1;

//Sensor1 frequency measurement
extern unsigned int freqMeasureTimer;
int CaptureValue;
int *pCaptureValue=&CaptureValue;
```

```
bool firstPass=TRUE;
extern word nOverFlows;
extern word icapture1;
extern word icapture2;
extern bool pDone;
extern bool firstPeriod;
extern word nPeriods;
extern word ROCtimer;
extern bool active;
extern word realoverflow;
extern word realperiod;
extern bool Valve1Sw;
extern bool Valve2Sw;
extern int Valve1Count;
extern int Valve2Count;
extern word Valve1Timer;
extern word Valve1TimeOut;
extern word Valve2Timer;
extern word Valve2TimeOut;
extern word LOS1_Timer;
extern bool LOS1;
extern word LOS2_Timer;
extern bool LOS2;
extern bool Valve1Position;
extern bool Valve2Position;
extern bool Valve1Command;
extern bool Valve2Command;
extern word AlarmTimer;
extern bool iso;

/*
**
============================================================================
**    Event       :  Timer_OnInterrupt (module Events)
**
**    From bean   :  Timer [TimerInt]
**    Description :
**        When a timer interrupt occurs this event is called (only
**        when the bean is enabled - "Enable" and the events are
**        enabled - "EnableEvent").
**    Parameters  : None
**    Returns     : Nothing
**
============================================================================
```

```
*/
void Timer_OnInterrupt(void){
   TimerCounter += 1;
   if(TimerCounter == 500) {//500 = 1/2 sec
      LED1_PutVal(TRUE);
   }
   if(TimerCounter == 1000) {//1000 = 1 sec
      LED1_PutVal(FALSE);
      TimerCounter = 0;
   }

   if(statusTimer) statusTimer -=1;
   if(uptimer) uptimer +=1;
   if(downtimer) downtimer +=1;
   if(Valve1Timer) Valve1Timer -=1;
   if(Valve1TimeOut) Valve1TimeOut -=1;
   if(Valve2Timer) Valve2Timer -=1;
   if(Valve2TimeOut) Valve2TimeOut -=1;
   if(LOS1_Timer) LOS1_Timer -=1;
   if(LOS2_Timer) LOS2_Timer -=1;
   if(AlarmTimer) AlarmTimer -=1;

   if(LOS1_Timer==0 && LOS1==FALSE){
      LOS1=TRUE;
      iso=TRUE;
   }

   if(LOS2_Timer==0 && LOS2==FALSE){
      LOS2=TRUE;
      iso=TRUE;
   }

}

/*
**
===================================================================
**     Event      :  AD1_OnEnd (module Events)
**
**     From bean   :  AD1 [ADC]
**     Description :
**         This event is called after the measurement (which
**         consists of <1 or more conversions>) is/are finished.
**         The event is available only when the <Interrupt
```

```
**         service/event> property is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void AD1_OnEnd(void){
   retValue=AD1_GetChanValue(channel1, &adValue1);
   adValueReady1=TRUE;
}

/*
**
============================================================================
**     Event       :  Valve2_Switch_OnInterrupt (module Events)
**
**     From bean   :  Valve2_Switch [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void Valve2_Switch_OnInterrupt(void){
   if(Valve2_Switch_GetVal()){
      Valve2Sw=TRUE;
      Valve2Count++;
   } else{
      Valve2Sw=FALSE;
   }
}

/*
**
============================================================================
**     Event       :  Valve1_Switch_OnInterrupt (module Events)
**
**     From bean   :  Valve1_Switch [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
```

```
**     Returns     : Nothing
**
============================================================================
*/
void Valve1_Switch_OnInterrupt(void){
   if(Valve1_Switch_GetVal()){
      Valve1Sw=TRUE;
      Valve1Count++;
   } else{
      Valve1Sw=FALSE;
   }
}

/*
**
============================================================================
**     Event       :  down_falling_OnInterrupt (module Events)
**
**     From bean   :  down_falling [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void down_falling_OnInterrupt(void){
   if(down_falling_GetVal()==FALSE){
      downTimerCalc=downtimer;
      down=TRUE;
   }

}

/*
**
============================================================================
**     Event       :  down_rising_OnInterrupt (module Events)
**
**     From bean   :  down_rising [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
```

```
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void down_rising_OnInterrupt(void){
   if(down_rising_GetVal()){
      downtimer=1;
   }
}

/*
**
============================================================================
**     Event       :  up_falling_OnInterrupt (module Events)
**
**     From bean   :  up_falling [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void up_falling_OnInterrupt(void){
   if(up_falling_GetVal()==FALSE){
      upTimerCalc=uptimer;
      up=TRUE;
   }
}

/*
**
============================================================================
**     Event       :  up_rising_OnInterrupt (module Events)
**
**     From bean   :  up_rising [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
```

```
**
============================================================================
*/
void up_rising_OnInterrupt(void){
   if(up_rising_GetVal()){
      uptimer=1;
   }
}

/*
**
============================================================================
**     Event      :  Sensor1_OnCapture (module Events)
**
**     From bean   :  Sensor1 [Capture]
**     Description :
**         This event is called on capturing of Timer/Counter actual
**         value (only when the bean is enabled - <"Enable"> and the
**         events are enabled - <"EnableEvent">.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
void Sensor1_OnCapture(void){
   Sensor1_GetCaptureValue(pCaptureValueSensor1);
   if(sensor1pulse==0){    //if first capture store it in icapture3
      icapture3=*pCaptureValueSensor1;
      if(icapture3>65000){  //are we to close to an overflow?
         phaseStart=TRUE;//force to start again
         return;
      } else{
         sensor1OverFlows=0;
         sensor2ExactOverFlows=0;
         sensor2armed=TRUE;
      }
   } else {
      if(sensor1pulse==2){
         icapture5=*pCaptureValueSensor1;
         sensor1=TRUE;
         sensor1ExactOverFlows=sensor1OverFlows;
         period=sensor1pulse;
      }
   }
```

```
  sensor1pulse +=1;
}

/*
**
============================================================================
**    Event      :  Sensor2_OnCapture (module Events)
**
**    From bean   :  Sensor2 [Capture]
**    Description :
**       This event is called on capturing of Timer/Counter actual
**       value (only when the bean is enabled - <"Enable"> and the
**       events are enabled - <"EnableEvent">.
**    Parameters  : None
**    Returns     : Nothing
**
============================================================================
*/
void Sensor2_OnCapture(void){
  Sensor2_GetCaptureValue(pCaptureValueSensor2);
  LOS2_Timer=100;
  if(sensor2armed==TRUE){
     icapture4=*pCaptureValueSensor2;
     sensor2=TRUE;
     sensor2ExactOverFlows=sensor1OverFlows;
     sensor2armed=FALSE;
  }
}

/*
**
============================================================================
**    Event      :  Sensor1_Frequency_OnCapture (module Events)
**
**    From bean   :  Sensor1_Frequency [Capture]
**    Description :
**       This event is called on capturing of Timer/Counter actual
**       value (only when the bean is enabled - <"Enable"> and the
**       events are enabled - <"EnableEvent">.
**    Parameters  : None
**    Returns     : Nothing
**
============================================================================
*/
```

```
void Sensor1_Frequency_OnCapture(void){
   Sensor1_Frequency_GetCaptureValue(pCaptureValue);
   LOS1_Timer=100;
   if(nPeriods==0){   //if first capture store it in icapture1
      icapture1=*pCaptureValue;
      if(icapture1>65000){   //are we to close to an overflow?
         firstPeriod=TRUE;//force to start again
         return;
      } else {
         nOverFlows=0;
      }
   }

   nPeriods +=1;

   if(nOverFlows>=6){
      if(active==TRUE && nPeriods>=2){
         icapture2=*pCaptureValue;
         realoverflow=nOverFlows;
         realperiod=nPeriods;
         pDone=TRUE;//set period done flag
         active=FALSE;
      }
   }
}

/*
**
===================================================================
**     Event      :  Sensor1_Frequency_OnOverflow (module Events)
**
**     From bean   :  Sensor1_Frequency [Capture]
**     Description :
**         This event is called if counter overflows (only when the
**         bean is enabled - <"Enable"> and the events are enabled -
**         <"EnableEvent">.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
void Sensor1_Frequency_OnOverflow(void){
   TFLG2_TOF=1;   //Reset overflow interrupt request flag
   nOverFlows +=1;
```

```
   sensor1OverFlows +=1;
}

/*
**
===================================================================
**     Event       :  RS485_OnError (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
**     Description :
**         This event is called when a channel error (not the error
**         returned by a given method) occurs. The errors can be
**         read using <GetError> method.
**         The event is available only when the <Interrupt
**         service/event> property is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
void RS485_OnError(void){
   RS485_RxCount=0;
   STX=FALSE;
   ACPT=FALSE;
   retValue=RS485_GetError(&RS485err);
}

/*
**
===================================================================
**     Event       :  RS485_OnRxChar (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
**     Description :
**         This event is called after a correct character is
**         received.
**         The event is available only when the <Interrupt
**         service/event> property is enabled and either the
**         <Receiver> property is enabled or the <SCI output mode>
**         property (if supported) is set to Single-wire mode.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
```

```
*/
void RS485_OnRxChar(void){

//Get characters in the receive buffer if there is something there
   while(RS485_GetCharsInRxBuf()!=0){
   retValue=RS485_RecvChar(&RS485chr);
   if(RS485chr==2){     //if the STX character (2) is received set the STX flag and exit
      if(STX){

//If the second STX character is received clear the RxCount and exit
         RS485_RxCount=0;
         ACPT=TRUE;
      } else{
         STX=TRUE;
         ACPT=FALSE;
      }
      return;
   }else STX=FALSE;        //if there is no STX character then the STX flag should be cleared

   if(ACPT==FALSE) return;
   if(RS485chr==3) {

//If the EOF character (3) is received and the RxCount has indicated that data has been received
//then the DCS has established communications and the EOF character is replaced by a 0 in the
//RxBuffer
      if(RS485_RxCount>1){
         RS485infoReady=TRUE;
         RS485_RxBuffer[RS485_RxCount]='\0';
         ACPT=FALSE;
      }
      return;      //if the EOF character is received without a packet body; then just exit
   }

//Validating the remaining packet for valid characters (0-9, spaces, and a decimal)
//If the character received is greater than an ASCII 9 (a value of 57 in decimal) then clear the
//RxCounter and exit

   if(RS485chr>'9'){
      RS485_RxCount=0;
      return;
   }
```

```
//If the character received is less than an ASCII 0 (a value of 48 in decimal) and it's not a space
//(a value of 32 in decimal) and it's not a decimal (a value of 46 in decimal) then clear the
//RxCounter and exit
   if(RS485chr<48 && RS485chr!=32 && RS485chr!=46){
      RS485_RxCount=0;
      return;
   }

//If the character passes all the tests for a valid character then place it in the RxBuffer
   RS485_RxBuffer[RS485_RxCount]=RS485chr;

//Increment the RxCounter as long as it is less than the size of the RxBuffer
   if(RS485_RxCount<sizeof(RS485_RxBuffer)) RS485_RxCount+=1;
   }//end of while
}

/*
**
===================================================================
**     Event       :  RS485_OnTxComplete (module Events)
**
**     From bean   :  RS485 [AsynchroSerial]
**     Description :
**         This event indicates that the transmitter is finished
**         transmitting all data, preamble, and break characters and
**         is idle. It can be used to determine when it is safe to
**         switch a line driver (e.g. in RS-485 applications).
**         The event is available only when both <Interrupt
**         service/event> and <Transmitter> properties are enabled.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
void RS485_OnTxComplete(void){
   RS485_DE_ClrVal();//disable RS485 transmitter
   RS485_RE_ClrVal();//enable RS485 receiver
}

/* END Events */
/** ###################################################################
**     This file was created by UNIS Processor Expert 2.96 [03.76]
**     for the Freescale HCS12X series of microcontrollers.
** ###################################################################*/
```

Vita

Christina Ward was born in Gainesville, FL on September 10th 1980. She graduated from Fernandina Beach High School in 1999 and continued her education while on an athletic scholarship at Hillsborough Community College. After graduating in May 2001 with an Associate of Arts in Engineering, she transferred to the Electrical and Computer Engineering department at the University of Florida. Christina graduated from UF in December 2003 with a BSEE degree. She has been working for Oak Ridge National Laboratory since January 2005 in the Electronic and Embedded Systems group (formally the Analog and Digital Systems group). She was awarded her MSEE degree in May 2011 from the Electrical Engineering and Computer Science department from the University of Tennessee, Knoxville.