Motor Control Demonstration Lab

JIM SIBIGTROTH and EDUARDO MONTAÑEZ
Freescale Semiconductor launched by Motorola, 8/16 Bit MCU Division, Austin, TX 78735, USA.
Email: j.sibigtroth@freescale.com eduardo.montanez@freescale.com

Abstract:
This paper presents an example laboratory exercise using a microcontroller to control a motor. NI ELVIS is used as a convenient platform to connect components, provide power, and analyze circuit operation. A small MCU module with required basic circuitry for crystal, serial I/O, and debugger connections is provided as a small module that plugs into the NI ELVIS breadboard. MCU program development and debug are done with Metrowerks CodeWarrior™ software running on the same host PC as the NI ELVIS and LabVIEW™ software.

INTRODUCTION

Microcontrollers (MCUs) are used in virtually every field of science and engineering. The smallest MCUs have only eight pins and are used in toys, appliances, and for tasks as simple as interfacing a few switches in a car door. More sophisticated MCUs have processing horsepower rivaling that of the most powerful desktop computers and are used to perform complex real-time fuel and spark timing computations in modern automobile engines. MCUs have become so common that they are now considered an essential part of almost all science and engineering programs. As MCU development tools become more user-friendly and evaluation boards come down in price, classes in microcontroller technology are being presented earlier in the engineering curriculum.

This motor control lab exercise was developed to explore how the NI ELVIS system[1] could complement existing microcontroller development tools in the engineering laboratory environment. The Freescale MC9S12C32 MCU[2] is used and Metrowerks CodeWarrior[3] is used as the MCU development tool. NI ELVIS offers an easy way to introduce controlled stimulus signals to the MCU and to monitor outputs from the MCU. The breadboard provides a convenient platform for wiring peripheral circuitry and multiple power supplies for the lab exercise. As an added convenience, NI ELVIS virtual instruments such as the oscilloscope provide a way to look at various signals such as the PWM signal that drives the motor speed.

By using virtual switches from a LabVIEW simulated front panel (SFP) instead of physical switches, certain real-world complications such as switch bounce can be avoided during introductory level exercises. After getting the initial lab exercise to work, the student could be assigned to replace the virtual switches with real switches so they experience, and then solve, the real-world problem of switch bounce.

This paper discusses the implementation of this demonstration lab exercise. The implementation and theory of operation is given for each part of the system. Several possible variations are discussed that could be used to adapt this lab to slightly different audiences. For example, some classes might concentrate on developing a more sophisticated LabVIEW front panel that could monitor and record the duty cycle of the PWM drive signal and the resulting motor speed. Another class might develop a more sophisticated enable/disable mechanism rather than the basic 1-2-3 sequence that we implemented. Yet another class might extend this lab to implement a closed loop motor speed controller using either PID or fuzzy logic techniques.

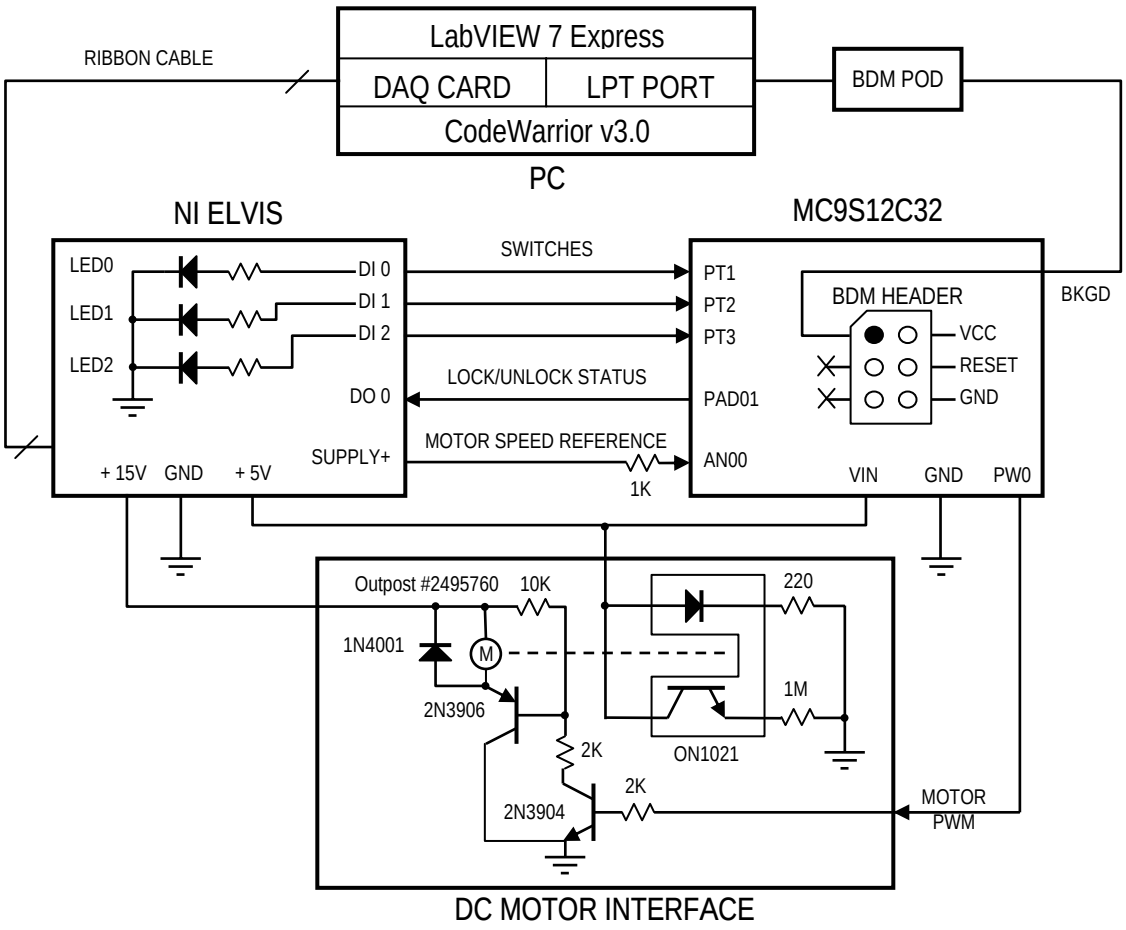

Figure 1. Block Diagram of the Motor Control Demonstration Lab

SYSTEM BLOCK DIAGRAM

Figure 1 shows the overall block diagram of the system. The host PC is shown with the data acquisition card leading to the NI ELVIS system. A background debug (BDM) pod[4] is connected from the host PC to a BDM connector on the microcontroller (MCU) module. CodeWarrior® development software supports debugging through a BDM pod connected to a printer port or USB port on the PC, or a serial I/O connection from a PC serial port to a serial interface on the MCU module may be used in place of the BDM pod (see serial monitor discussion later in this paper). The NI ELVIS block includes power supplies, digital I/O, and indicator LEDs, as well as a large breadboard area for the MCU module and experiment circuitry. The MCU block includes an MC9S12C32 16-bit MCU along with a crystal, RS-232 level shifters, and a BDM connector for programming and debugging the application software. The dc motor interface block includes the motor, a 2-transistor circuit to translate the 0-5v PWM signal from the MCU into a 0-15v signal to drive the motor. Though not used in this demonstration lab, the dc motor interface block also includes an optical interrupter sensor that can be used to monitor motor speed using a slotted disk attached to the motor shaft.

SIMULATED FRONT PANEL

In this example lab exercise, a simulated front panel (SFP) was written in LabVIEW to emulate switches for inputting a code sequence, an LED to indicate the enable/disable state of the PWM output, and a dial to set the desired motor speed. An SFP is part of a LabVIEW virtual instrument (VI). This SFP controls digital I/O on NI ELVIS for the digital signals and the dial controls the positive variable power supply on NI ELVIS to produce an analog voltage level between 0 and 5 volts. Figure 2 shows the motor control SFP.

Normally the NI ELVIS Instrument Launcher would only allow a single VI to run at a time. You could run multiple instruments under LabVIEW as long as they do not try to use the same NI ELVIS resources. Since the front panel for this experiment and the oscilloscope are both VIs, we have included a button on our front panel to invoke the oscilloscope as a sub process. This provides a convenient way to run the front panel and the oscilloscope at the same time.

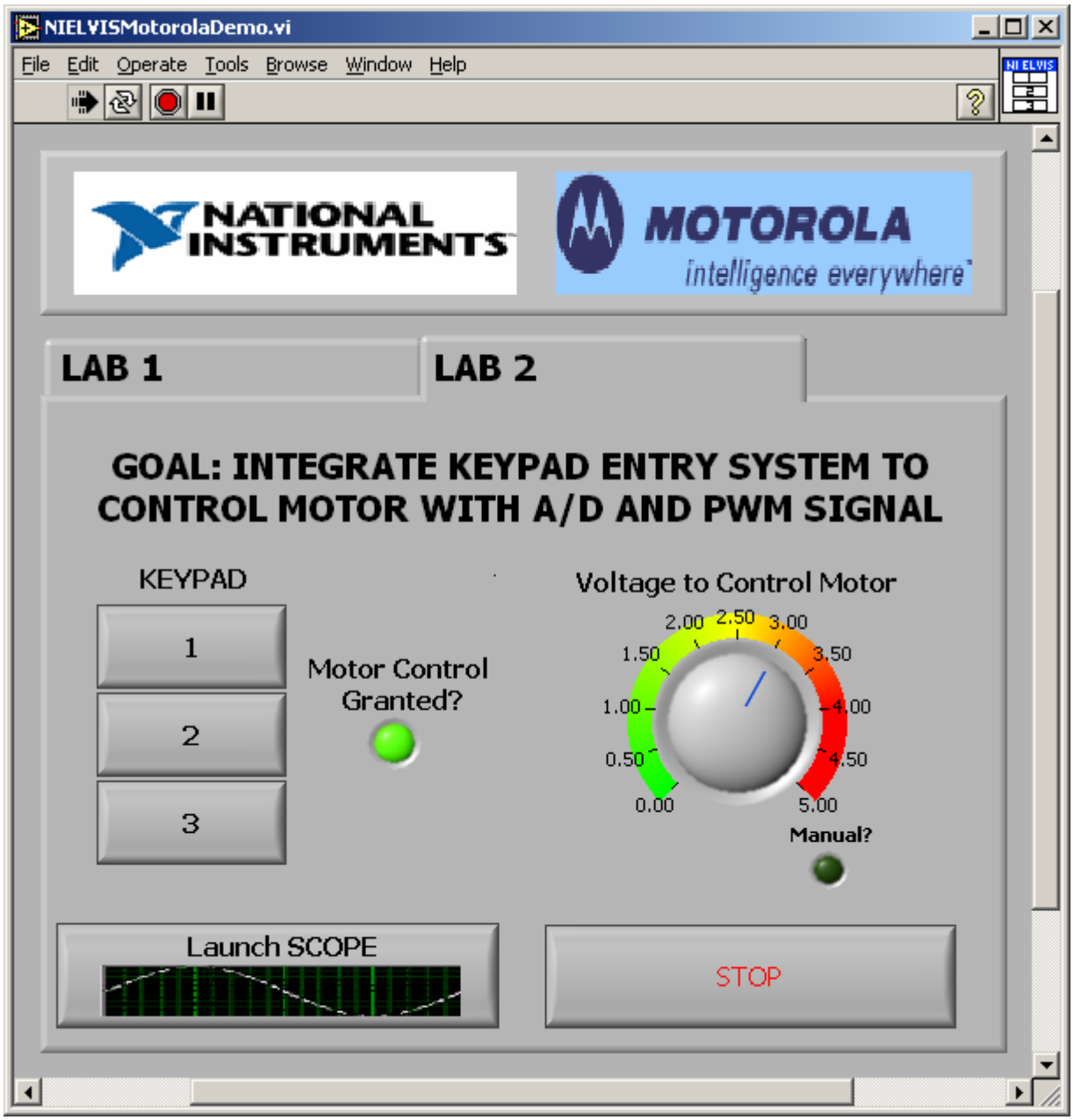

Figure 2. Motor Control Front Panel

The "Manual?" indicator, at the lower right, lights when the positive variable power supply on the NI ELVIS workstation is switched to manual mode. Since the variable power supply can be set to +12v in manual mode (compared to 0-5v with the front panel), a series resistor is installed on the breadboard between this voltage source and the ATD input of the MCU.

Virtual Switches

Each time a user clicks on a switch on the PC screen, the corresponding digital input signal is driven low for a 100 millisecond low-true pulse. These virtual switch signals are different than real switches

because they produce clean signals while real switches typically exhibit switch bounce. Switch bounce causes extra edges as the switch contacts close and open. Dealing with this switch bounce requires extra programming in the microcontroller that may be beyond the ability of a student in an introductory level class. This demonstrates one way the virtual front panel can be used to simplify the solution to problems in an introductory level class. It would also be possible to program the simulated switches to intentionally produce worst-case bounce in order to challenge more advanced students.

Motor Control LED Indicator

This demonstrates how a digital output from the NI ELVIS breadboard can be used to provide signals to the PC where they can be used in a simulated front panel. In this case an output from the MCU is connected to a digital output on the breadboard. This signal will be used to indicate whether the motor control output from the MCU is enabled (granted) or disabled.

Speed Dial

The speed dial on the SFP controls the level of the positive variable power supply to produce an analog voltage between 0 and 5 volts. NI ELVIS also has two DAC signals that could have been used here, but we chose the variable power supply because it can be placed in manual mode. During development of the lab, the student can switch the variable supply to manual to allow control of this voltage with the knob on the front of the NI ELVIS workstation rather than being controlled from the SFP.

Oscilloscope VI Button

This button launches the NI ELVIS oscilloscope VI as a sub process of the front panel. This overcomes a limitation of the Instrument Launcher, so that the front panel and the oscilloscope VI can operate at the same time. Figure 3 shows the oscilloscope VI. The waveform displayed on the oscilloscope is the PWM signal from the MCU. From the settings and measurement annotations on the oscilloscope screen, you can see that the PWM signal has a period of about 500 microseconds so the PWM frequency is about 2 KHz.

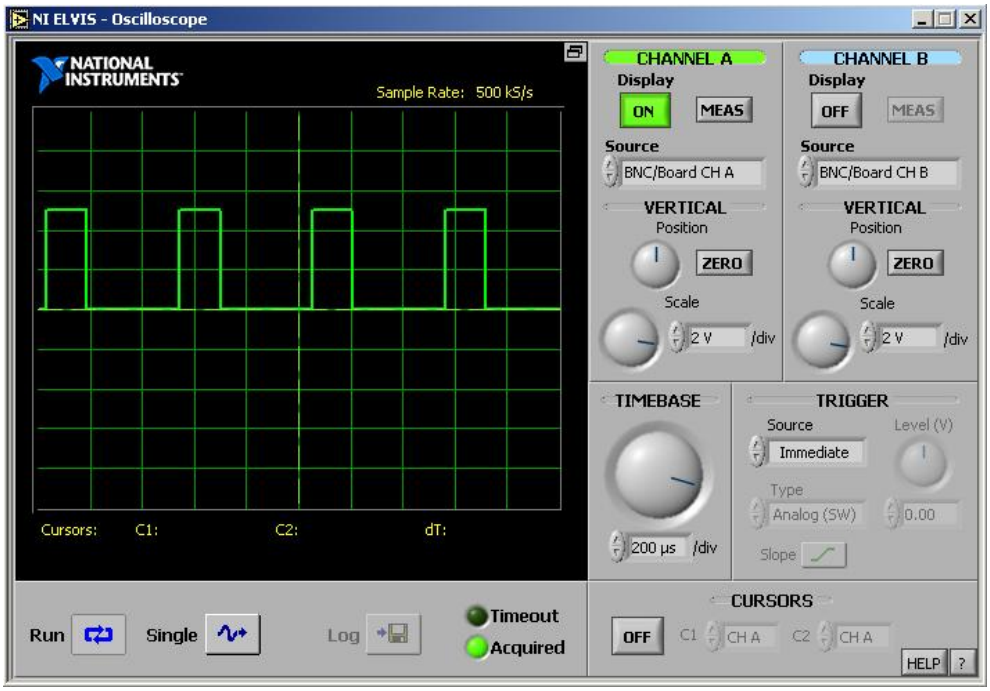

Figure 3. Oscilloscope VI Showing PWM Signal from MCU

The oscilloscope VI is a virtual instrument that appears as an interactive window on the PC screen with buttons and knobs similar to those found on traditional oscilloscopes. The probes for this oscilloscope can be connected to the BNC connectors on the front of the NI ELVIS workstation, or they can be wires from the CH_A+/- and CH_B+/- connectors at the upper left corner of the breadboard. The student can make the same kind of measurements with this virtual oscilloscope as they could with a physical oscilloscope, without the need for a separate piece of test equipment.

MC9S12C32 MCU BLOCK

The MCU block is a pre-assembled module made by Technological Arts[5] that includes the MC9S12C32 MCU and a small amount of common support circuitry. This module is also available from Freescale Semiconductor under the part number M68DKIT912C32. A crystal and a few passive components provide a clock source for the MCU. An RS-232 level shifter converts MCU signals to levels required by the serial I/O standard. The level shifted versions of RxD and TxD are wired to pins 1 and 2 of the MCU block. These can be wired to a DB-9 serial I/O connector to allow the MCU to communicate with a host PC or other computer device.

Figure 4 shows the pin assignments for the MCU block. The block has the same footprint as a standard 32-pin DIP IC. Pin 1 is shown in the upper right corner. Pin names inside the block are the Technological Arts pin names. The names of the signals as used in the application are shown outside the block.

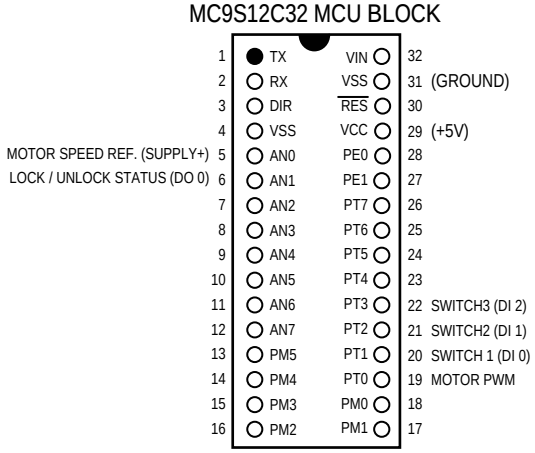

Figure 4. Pin Assignments for the MCU Block

I/O Characteristics

For this lab exercise, we use an analog (ATD) input to measure the SUPPLY+ voltage which is used to set the motor speed. The ATD inputs to the MCU should be limited to 0-5 volts. Because of input protection circuitry inside the MCU, there is effectively a diode to Vdd which conducts when the voltage on an ATD input is a diode drop or more above Vdd. In this application, the front panel VI limits the output on SUPPLY+ to 5 volts, but if the variable power supply is set to manual, it is possible to adjust this voltage to 12 volts. In order to avoid damage to the MCU, a 1K ohm resistor is placed in series between the SUPPLY+ voltage and the ATD input pin. For normal levels on this pin, there is very little current in this resistor so the voltage at the ATD input pin is effectively equal to the SUPPLY+ voltage. When SUPPLY+ is set to 12 volts, the ATD pin clamps at about 5.7 volts so there is (6.3v/1K ohm) or about 6.3 milliamps of current into the ATD input pin. This is not enough current to harm the MCU.

The three virtual switches are connected to input pins PT1, PT2, and PT3 on the MCU. For this application, our software configures these multi-purpose pins as timer input capture pins so they can be used to detect edges that correspond to switch closures. These are standard CMOS inputs that have their switch point near Vdd/2. These pins also have a small amount of hysteresis to help filter out noise when the input level is near its switch point. NI ELVIS 'DI x' signals are TTL compatible so a high level is about 4 volts rather than 5 volts like a CMOS device would normally drive. This level is compatible with the MCUs CMOS input levels. Like the ATD input pins, these general purpose digital I/O pins have internal input protection circuitry that clamps the pin level to no more than a diode drop above Vdd or below Vss. The user is responsible for limiting the current if higher or lower voltages are expected from external circuits.

The Lock/Unlock Status signal shown in the middle of Figure 1 is driven by a general purpose digital output pin on the MCU. These pins typically drive the pin to Vdd for logic one and Vss for logic zero. They are specified to drive up to 2 ma each in reduced drive mode or 10 ma each in full drive mode over the full rated voltage and temperature ranges.

The PWM signal is also a digital output pin with the same characteristics as a general purpose digital output. The 9S12C32 PWM timer can generate 8-bit or 16-bit resolution PWM signals with a clock input frequency of up to 25 MHz. For example, you could produce an 8-bit resolution PWM signal with a frequency of nearly 100 KHz. For this motor control demonstration lab we chose a PWM frequency of just 2 KHz so that the oscilloscope VI could be used to examine the waveform even with a relatively slow DAQ card. In a commercial application, you would probably want to use a PWM frequency of 20 KHz or higher so that you couldn't hear this frequency at low motor speeds. In this demo lab, you can hear the 2 KHz signal a little bit at low motor speeds. You can try this by listening while the PWM is controlling the speed and then drive the motor directly from the variable power supply in manual mode (you will not hear the 2 KHz whistle when the motor is driven from the dc supply.)

The serial port of the PC uses standard RS232 voltage levels (from –X volts to +X volts where X can be from 6 to 15 volts depending upon the particular PC). A level shifter device is included on the MCU block to convert the RS232 levels to the 0-5volt levels that are required at the RxD and TxD pins of the MCU. We have wired the level shifted RxD and TxD pins to the DSUB connector. This connection would be used if you were using the serial monitor rather than a BDM pod to interface CodeWarrior to the MCU. The serial monitor eliminates the need for a BDM pod, but it is not as elegant (unobtrusive) as the BDM pod.

Background Debug Connections

The ideal way to allow CodeWarrior to access the MCU is through the background debug interface. The MCU module includes a 2x3 square-post header for this purpose. This interface uses a single dedicated pin on the MCU plus ground and optional connections to reset and Vdd. CodeWarrior communicates with a BDM pod through a parallel or USB port on the PC. The BDM pod converts commands from CodeWarrior into a custom serial BDM protocol. Using this interface, CodeWarrior can execute primitive commands to read or write memory locations (even while application programs are running), read or write CPU registers, set breakpoints, or trace single instructions. These primitive commands allow CodeWarrior to program the Flash memory and debug user programs.

Serial Monitor Option

The serial monitor[6] is a small 2 Kbyte program that is programmed into the Flash memory of the MC9S12C32. This program emulates primitive debugging commands similar to those available through the background debug interface. CodeWarrior has the ability to choose either a BDM pod or this serial monitor as the interface to the target MCU. All of the Flash programming and debug options of CodeWarrior are available independent of which interface is chosen. The main differences between

these two interface methods are the connection from the PC and the degree to which debugging interferes with the application program in the target MCU.

With the serial monitor, a simple serial cable is used to connect the serial I/O port of the PC to the (level shifted) SCI pins of the target MCU. This eliminates the cost of the BDM pod.

Like traditional ROM monitor programs such as BUFFALO for the M68HC11, the serial monitor is more intrusive than a BDM pod. The serial monitor uses 2 Kbytes of Flash memory, including the vector locations, and takes over control of a serial communications interface (SCI) port. Since the monitor program and the user program share the same CPU and memory, there can be interactions between the monitor and the user programs. These interactions can cause some confusion for a beginner such as a student in an introductory level class. The instructor should consider the tradeoff between ease-of-use and the cost of a BDM pod.

Motor Circuits

This demonstration lab exercise has two small blocks of circuitry associated with the motor as shown at the bottom of Figure 1. A 2-transistor emitter-follower circuit is used to translate the 0-5 volt PWM signal from the MCU to the 0-15 volt levels required for the motor. An optical interrupter sensor (light emitter-sensor pair) monitors the speed of the motor with the help of a slotted disk attached to the shaft of the motor. The motor speed sensor is not used in this demonstration lab, but it would be relatively easy to connect the sensor signal to a timer input to the MCU to measure motor speed. You could also use the oscilloscope VI to monitor the sensor output signal.

The motor is a basic dc brush motor. This motor could be driven by an analog voltage between 0 and 15 volts, or it can be driven by a PWM signal from the MCU. By adjusting the duty cycle of the PWM signal, you can control motor speed. The 1N4001 diode across the motor reduces the amount of noise generated by the motor when it is driven by a PWM signal. If you remove this diode while the motor is running, you should notice the speed drop slightly and the sound will change to be slightly more raspy.

MICROCONTROLLER SOFTWARE

The software for this paper can be downloaded form the Motorola Semiconductor web site. For this lab, the software is built using CodeWarrior software stationery. The software stationery provides all the software necessary to access internal microcontroller register space and program memory through software labels. The stationery associates all register and control bit names from the MCU data sheet with the appropriate address and bit position. The stationery provides the foundation for students to write their own embedded programming routines.

To open the demo software, first extract the file labeled NI_ELVIS_MCU_DEMO.zip to your chosen directory. The extraction will form a directory labeled NI_ELVIS_MCU_DEMO (ex. C:\NI_ELVIS_MCU_DEMO\), which includes a directory labeled NI_ELVIS_DEMO_C32. The NI_ELVIS_DEMO_C32 project directory is built for the MC9S12C32 MCU Block. To proceed, make sure to have Metrowerks CodeWarrior for HC(S)12 installed on your computer. Since this paper focuses on the MC9S12C32 MCU Block, the software description below will also reflect the NI_ELVIS_DEMO_C32 project. Within the NI_ELVIS_DEMO_C32 directory you will find a file labeled NI_ELVIS_DEMO_C32.mcp, which is the Metrowerks CodeWarrior project file for this lab's demo software. Double-click on the file and the demo software project will be loaded in the CodeWarrior development environment. Figure 5 shows the lab's demo software open in the CodeWarrior development environment.

Typical microcontroller software is written in three basic steps. First, you must define specific microcontroller setup parameters. This includes manipulating internal clock prescalers to change the core bus speed, relocating memory space, or specifying the desired microcontroller operating mode.

For this lab, the software uses default clock prescalers (Bus Clock Speed = Crystal Clock Speed / 2) since there is no need for faster cycle execution (lab is not time critical). The MC9S12C32 MCU block is supplied with an 8MHz crystal. Therefore, the default microcontroller bus clock speed equals 4MHz. Also, this introductory level lab does not require any memory relocation or special operating modes.

The second step to developing microcontroller software is configuring the behavior of individual peripherals or multiplexed I/O pins. In software, this is referred to as initialization. Ritual initialization routines specify the tasks to be performed by peripherals or select the functionality of multi-function I/O pins. For this lab, the io_init () function initializes the Lock/Unlock Status signal by configuring a general purpose I/O pin to act as an output with an initial level of 0V (logic 0 = lock; logic 1 = unlock) reflecting the motor status. The pwm_init () function initializes the PWM peripheral to generate a left-aligned, high-true PWM on PWM Ch. 0 (0% duty cycle = continuous low). The routine also configures the PWM period to 2 KHz by adjusting the channel's period and clock prescaler registers. The signal duty cycle is set to a default 0% and the PWM Ch. 0 output remains disabled (Lock/Unlock Status = 0) until later in the demo software where a successful 1-2-3 switch combination enables the Motor PWM signal. Next, the atd_init () function initializes the ATD peripheral to continuously sample the Motor Speed Reference signal (Supply +) that determines the speed of the dc motor. In detail, the function configures an 8-bit resolution conversion sequence on ATD Ch. 0, which generates an interrupt when a sampling sequence is completed. The tim_init () function initializes timer channels 1, 2, and 3 as input capture channels that generate separate interrupts upon an edge detect. These channels are set to detect falling edges generated by switch closures. These switch inputs are generated by virtual switches programmed in the motor control front panel VI. Note that each of the functions mentioned above can be found in their respective peripheralname.c files. Also, note that it is good practice to disable interrupts, while initializing the peripherals, and re-enabling interrupts upon completion.

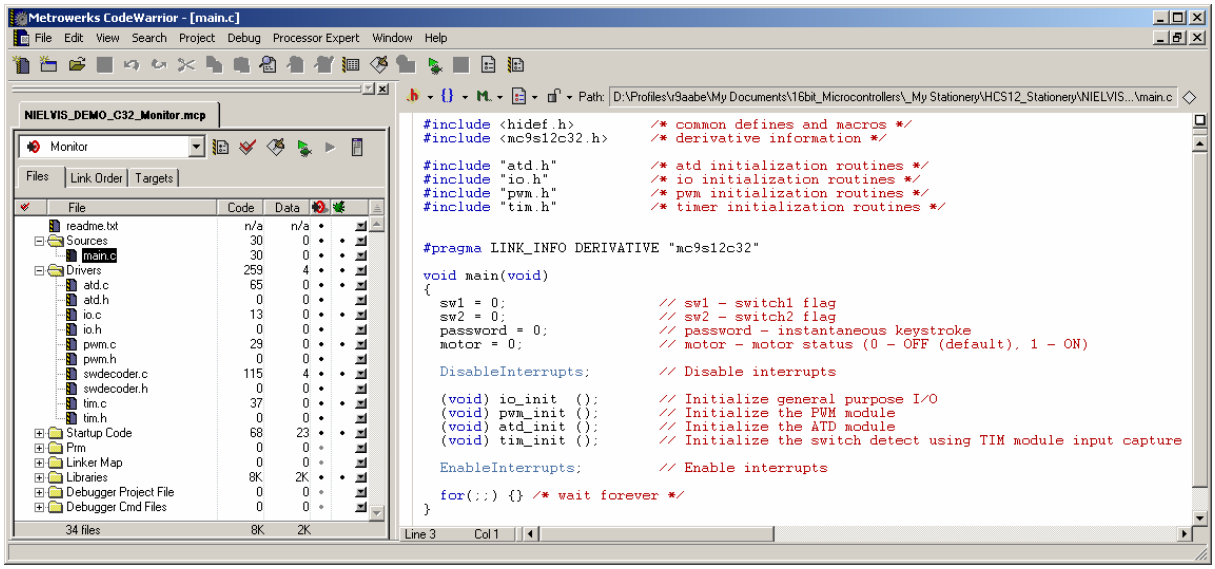

Figure 5. Motor Control Project in CodeWarrior

The final step to developing microcontroller software is to write the main control algorithm by using the results provided by the individual peripherals. For this lab, the microcontroller generates an interrupt when a switch closure is detected that calls a switch_decoder () function. The switch_decoder () function then determines, which switch was pressed and checks whether the valid 1-2-3 sequence has been satisfied. If not, it either saves the last switch value entered to compare with the next switch value input or it clears all three switch values if the 1-2-3 pattern has been violated. Once the correct, 1-2-3 combination is entered the software enables the PWM Ch. 0 output, which drives the Motor PWM and drives the Lock / Unlock Status signal high. If the combination is entered again the PWM Ch. 0 output is disabled turning off the dc motor, and the Lock/Unlock Status signal is cleared to a

locked (0) level. The next control algorithm is the motor speed control. The PWM Ch. 0 duty cycle must reflect the analog voltage Motor Speed Reference signal. The ATD Ch. 0 peripheral continuously samples this reference voltage and generates an ATD interrupt when a conversion sequence is completed. Within this interrupt function, the 8-bit ATD Ch. 0 data result register is copied into the 8-bit PWM Ch. 0 duty cycle register. This causes the microcontroller PWM Ch. 0 to generate a 0 to 100% duty cycle directly proportional to a 0 to 5V reference on ATD Ch. 0. The two software variable windows of the CodeWarrior debugger in Figure 6 show the ATD Ch. 0 data result is copied into the PWM Ch. 0 duty cycle register. These two control algorithms are continuously executed.

The software involved in this lab exercise demonstrates one of many ways to implement the motor control example. This lab uses an interrupt driven procedure, which might not be typical in more complex real world applications. However, this lab simplifies the software process in order to focus on control and embedded interfacing, typically addressed in an introductory microcontroller course. For more involved, time critical applications a routine like the switch_decoder () function would not be implemented within an interrupt service routine. Instead it would be embedded within your main () function and would be serviced periodically through a timed software loop. Overall, the three basic steps to developing microcontroller software remain the same regardless of which control approach you take. For additional information about writing programs for real time MCU applications see *Embedded Microcomputer Systems: Real Time Interfacing*[7] by Jonathan W. Valvano.

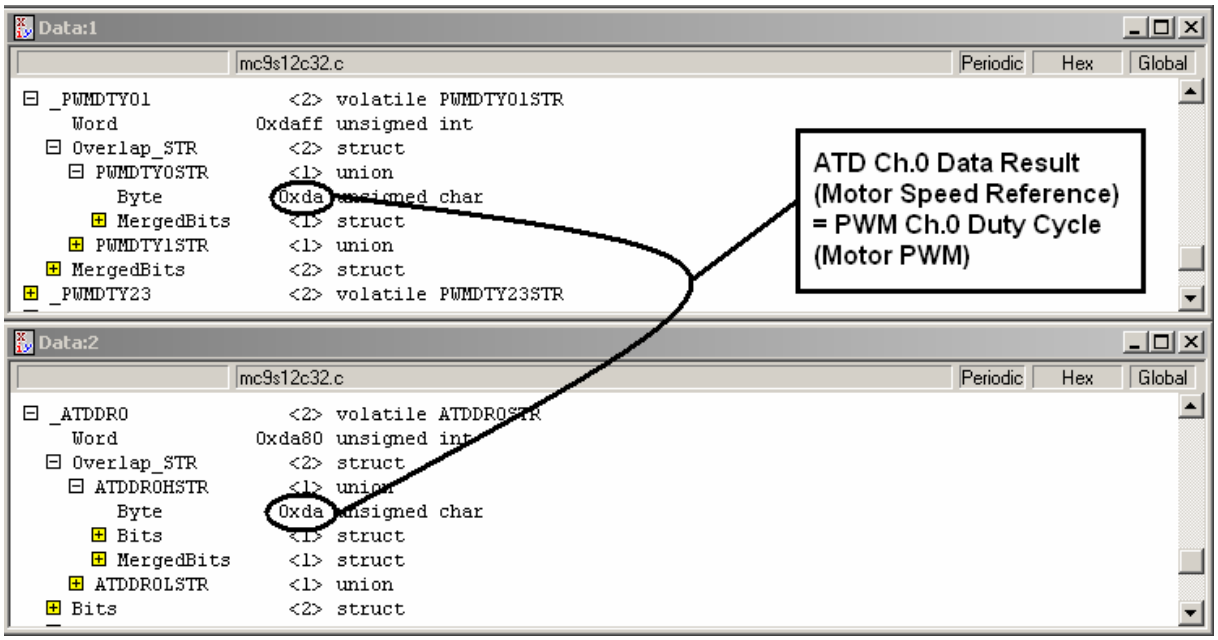

Figure 6. Software Variable Windows in CodeWarrior Debugger

TROUBLESHOOTING

The NI ELVIS system offers many helpful debugging tools in addition to the MCU debugging tools offered in CodeWarrior. NI ELVIS tools range from simple LEDs to the virtual DMM and oscilloscope tools.

Breadboard LEDs

The eight LEDs on the NI ELVIS breadboard can be used as simple logic level probes to test the digital levels present at various circuit nodes in the system. In this lab exercise we wired one LED to each of the digital input signals that are controlled by virtual switches on the SFP. Normally these LEDs remain lit. When a switch is clicked on the PC screen, the corresponding LED will blink off for about 100 milliseconds. This can be a useful debugging tool to make sure the SFP works as expected

and to check the connections to the MCU input pins. You could also use another LED to monitor the level on the MCU output pin that indicates whether or not the motor PWM signal is active.

CodeWarrior®

The CodeWarrior debugger provides a rich graphic user interface for debugging the application program. The debugger screen includes several window panes referred-to as components. The user can add, remove, or rearrange window panes. Figure 7 shows a typical CodeWarrior debugger screen. This debugger allows the programmer to debug their code, including C source-level debugging. You can monitor or change the contents of memory or registers, set breakpoints, trace individual instructions or C statements, monitor or change CPU register contents, and so on. For more information, refer to the CodeWarrior documentation.
The newest HCS08 and HCS12 MCUs, including the MC9S12C32 used in this lab, have an on-chip ICE (in-circuit emulator) [8] that can capture real-time bus information into an on-chip FIFO buffer. This system works with the CodeWarrior debugger to provide the functions of an in-circuit emulator and bus-state analyzer. In order to use the on-chip DBG, you need to add a "Trace" component to the debug screen, and then setup the desired trigger conditions. Bus information is captured in real time based on your trigger settings. Later, CodeWarrior reads the contents of the capture buffer FIFO, decodes the information, and displays the results in the trace component.

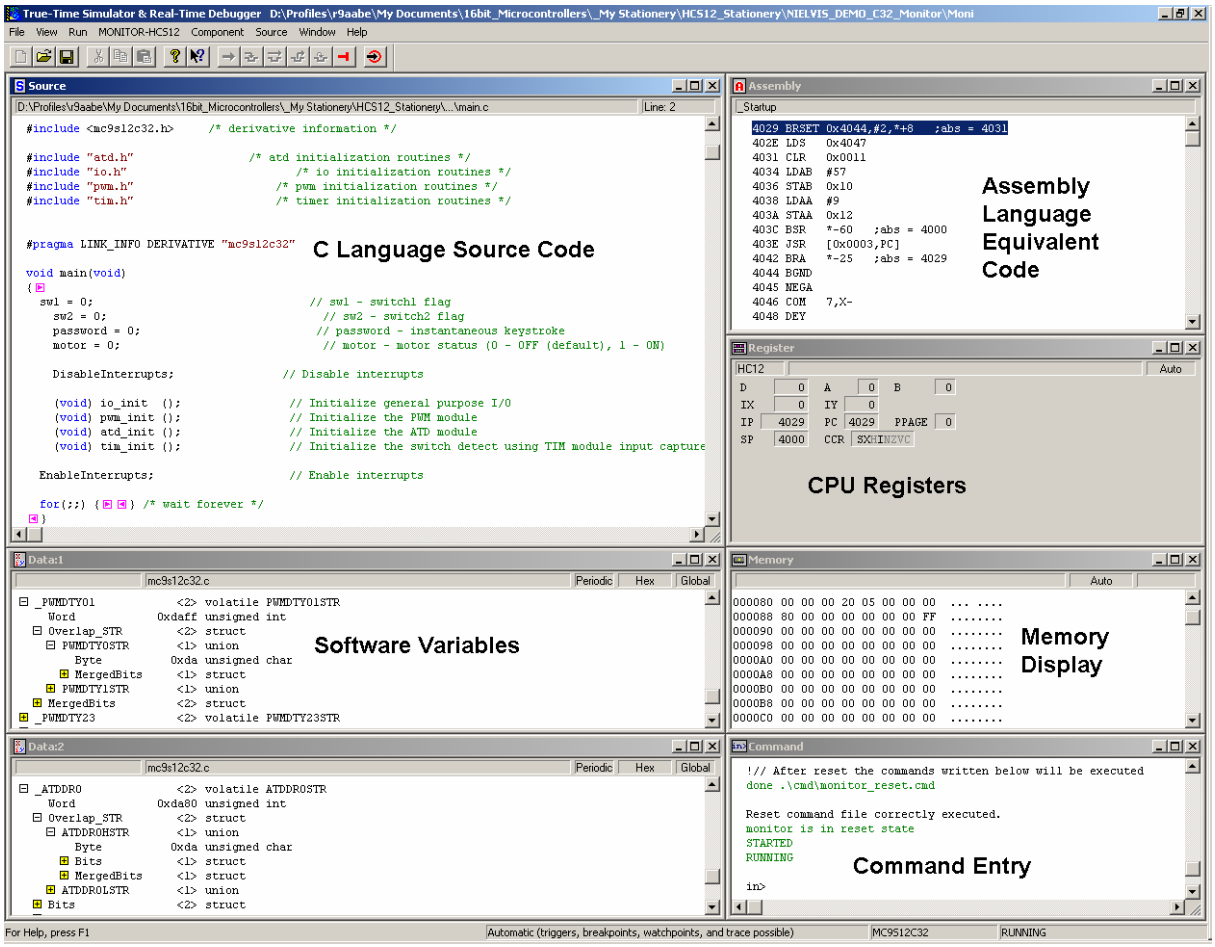

Figure 7. CodeWarrior Debug Screen with Several Components

Oscilloscope VI

The virtual oscilloscope can be used to examine various points in the motor system. The "Launch SCOPE" button on the MotorDemo VI launches the oscilloscope VI (but also keeps the MotorDemo

VI active. This allows you to connect the oscilloscope probe to a circuit point such as the PWM output from the MCU block, and watch the waveform as you adjust the dial on the MotorDemo front panel.

The oscilloscope can be used to troubleshoot other parts of the circuitry in the demonstration lab as it is being built. For example, it could be useful to look at various points in the emitter follower circuit to make sure the circuit is functioning as expected. You could look at the ATD pin on the MCU block as you manually adjust the positive variable power supply to levels above +5v Vdd. This would allow you to confirm the clamping of this input level to a diode drop above Vdd. You could also look at the switch inputs from the "DI 0", "DI 2", and "DI 3", signals from NI ELVIS to confirm levels and to measure the low-true pulse that is generated when you click on a virtual switch on the front panel. The oscilloscope can also be used to examine the output from the optical interrupter that monitors the motor speed.

CONCLUSION

This motor control demonstration lab has shown that NI ELVIS works well with microcontrollers and Metrowerks CodeWarrior software development tools. The NI ELVIS breadboard provides a convenient place to build the circuitry for the lab exercise. LabVIEW front panels can be used to provide digital and analog stimulus to the student's circuitry including the microcontroller. NI ELVIS provides various power supply voltages so there is no need to locate special equipment. Microcontrollers such as the MC9S12C32 are available on MCU blocks that easily plug into the breadboard so they can be integrated with other components to implement laboratory exercises.

The CodeWarrior development software provides useful tools for debugging MCU software, but does not allow a user to debug other external circuits such as the motor circuits in this lab exercise. NI ELVIS provides inexpensive virtual instruments such as the oscilloscope VI which give the student a way to examine and troubleshoot the circuitry that is connected to the MCU.

Including a 16-bit microcontroller like the MC9S12C32 in NI ELVIS lab exercises, allows the development of sophisticated engineering applications. These projects can include both circuit and software components. Working with this sort of application in an engineering course will help prepare students for the kind of projects they will find in industry after they graduate.

REFERENCES

1. National Instruments, *NI Educational Laboratory Virtual Instrumentation Suite (NI ELVIS) User Manual*, National Instruments, Austin, TX (2003).
2. Freescale, *MC9S12C32 Users Guides*, Freescale Semiconductor Inc., Austin, TX (2003).
3. Metrowerks, *CodeWarrior Development Studio for HC(S)12 Microcontrollers (Special Edition)*, Metrowerks, Austin, TX (2003).
4. P&E Microcomputer Systems, *BDM Multilink*, P&E Microcomputer Systems, Boston, MA (2003).
5. Technological Arts, *Using Your M68DKIT912C32 Microcontroller Kit*, Technological Arts, Toronto, Canada (2003).
6. Jim Williams, *AN2548/D Serial Monitor for HCS12 MCUs*, Freescale Semiconductor Inc., Austin, TX (2003).
7. Jonathan W. Valvano, *Embedded Microcomputer Systems: Real Time Interfacing*, Brooks/Cole, Pacific Grove, CA (2000).
8. Eduardo Montañez, *AN2596/D Using the HCS08 Family On-Chip Debug System*, Freescale Semiconductor Inc., Austin, TX (2003).