UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CMPE118/218/L: INTRODUCTION TO MECHATRONICS

LABORATORY ASSIGNMENT NUMBER 3 FOR CMPE 118/218

Due by 5:00pm on Wednesday, February 2, 2011
Pre-Lab Due by **5:00pm** on Thursday, January 27, 2011

Purpose: This lab is intended to acquaint you with:

Controlling a DC motor.
Controlling a stepper motor.
Controlling an R/C servo.
Finding information in data sheets.
Pulse width modulation.
Phenomena encountered with a DC motor.
Limitations of techniques implemented purely in software.

Minimum Parts Required: 1 each: DC motor, R/C servo, and stepper motor (provided at each lab station), 1N4744A 15 Volt zener diode, 50K potentiometer. CMPE-118 I/O Board, CMPE-118 Dual H-Bridge, DS3568 High Current Drive, and CMPE-118 Stepper Motor Driver. Ribbon cable connectors and stake headers for the I/O board. See a TA/tutor for any parts you need that are not in your parts kit.

Pre-Lab:

Complete the following exercises AFTER you have read through the lab assignment and BEFORE coming in to complete the lab.

☐ 0.1) Decide which port and bit(s) you will use to control the motors (all parts of lab).

☐ 0.2) Decide what, if any, initialization is required for that port.

☐ 0.3) Describe the mode in which the A/D converter will be operating.

☐ 0.4) Determine which pins on which cables/connectors you need to hook up.

☐ 0.5) Determine which ribbon connectors you will need, and how to wire them in.

☐ 0.6) Draw a fully detailed state diagram for a PWM software driver. Include an example plot of the expected output signal for a given frequency and duty cycle. HINT: using timer events may assist you getting the proper duty cycles for a given frequency.

In the report: Include a description of which of the port lines on the 9S12C32 you used to implement the control as well as a justification for why you chose to use that (those) particular line(s). Include the answer to question 0.3.

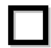
Part 1 Interfacing to a DC Motor and Potentiometer

Reading: CKO Ch. 8. ADS12 and PWMS12 reference documentation, the CMPE118 HC12 IO Board documentation, the CMPE118 DS3658 Module board documentation, the Metrowerks CodeWarrior Quickstart manual.

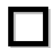
Assignment:

You are to design the software necessary for the 9S12C32 to drive the supplied DC motor. Motor drive should use Pulse Width Modulation using the provided PWMS12 software module. The speed at which you run the motor should be determined by the setting of an external potentiometer, which your software should read using the A/D converter on the 9S12C32. Your program should continuously read the voltage set by the potentiometer and set the motor speed directly proportional to the voltage (full scale A/D is full speed). You should be able to exit the program by pressing any key. When you have completed this task, you will be prepared to continue with the laboratory exercises in Part 2. At this point find a tutor or TA and demonstrate Part 1. **BE CAREFUL HERE**: Do **NOT** hook power and ground up with the power supply turned on. It is the easiest way to fry microcontrollers, followed by reversing power and ground.

Set-Up:

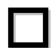


1.1) For this, and the next, part you will be using the CMPE118 DS3658 Module board to drive the DC motor. Since the DS3658 has open collector outputs, you will need to supply the power to the motor separately. To bring the power to your proto-board for connection to the motor, use your banana plug jumpers to bring the power from the positive adjustable outputs to your proto-board.

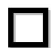


1.2) Use the +12V voltage source from the power supply for the positive output. Now connect the supply (at the proto-board) to one lead of the DC motor using the connector from the motor assembly. Also, you will need to supply power to the CMPE118 DS3658 Module board, so connect the +12V supply to J4, pin 1 on the DS3658 module, and connect J4, pin 2 to the GND terminal on the power supply.

1.3) Now connect the DS3658's clamp diode for Channels A & B to the motor supply at the proto-board, and the output for Channel A to the remaining lead of the DC motor. The motor should not be rotating at this point, right? If the motor is rotating, this is due to the fact that you haven't hooked up anything to the input of the DS3658, so the state of the input is not guaranteed – it's floating!



1.4) Hook up the 9S12C32 output(s) you selected in the pre-lab to the input of the channel you are using on the DS3658 (use ribbon cable connectors between the IO board and the protoboard).



1.5) Hook the two outer leads from the 50K potentiometer to the **power supply's +5V & GND** terminals. Be **very** careful **NOT** to use the power going to the motor, since that is +12V and will destroy the 9S12C32 once you connect it. After doing this, check to see that the voltage at the wiper (the remaining terminal on the potentiometer) swings between 0 and +5V. You will also use this circuit in Part 5. Check the voltage levels **BEFORE** you hook up the IO board.

1.6) Using one of the jumpers that goes from a wire to a push-on connector, make the connection from the wiper to the pin labeled AN0 on the HC12 IO Board (it's on J12). Make sure you go through the follower board and not directly to the pin. What are the limitations that this imposes? What does it do for the microcontroller? Why might you not want to do this?

1.7) It's finally time to try out your code. In CW12, build your project into an executable file. Then, with the uBug12 window active, turn on (or reset) the 9S12C32 board and type "CON 1" <ENTER>. You should get the connection message. If not, ensure SW1 is in the boot position. At this point you are ready to download your code. Downloading and testing your code should proceed exactly as it did in Lab 0. Before you can download your code, you need to clear the FLASH, so you must type "fbulk" and press <ENTER>. Now, everything is ready for the new code to be downloaded. You need to tell the Terminal window which file to download, so type "fload ;b" and select your file, which should be named something like MOTORLAB.ABS.S19 (the *.S19 file is compiled code in the proper format for downloading). Once your code is finished downloading, the You are now ready to execute your code. Type "go" or restart the 9S12C32 with SW1 in the "Run" position and connect using Tera Term to use serial IO. The proper settings are at 115.2Kbaud, no flow control, and be sure to init the SCI using TERMIO_Init().

In the report: You must include the wiring diagrams of the circuits you used, noting which bits were used and what connections were made on the HC12 IO Board and the CMPE118 DS3658 Module board, the sign-off sheet, the answers to the questions in part 1.6, as well as a listing of the software used to control the motor. All wiring diagrams and schematics should be completed using the OrCAD Capture program provided on each workstation.

Part 2 Exploring DC Motors

Reading: CKO Ch. 22, 23, 24, 26 and 27

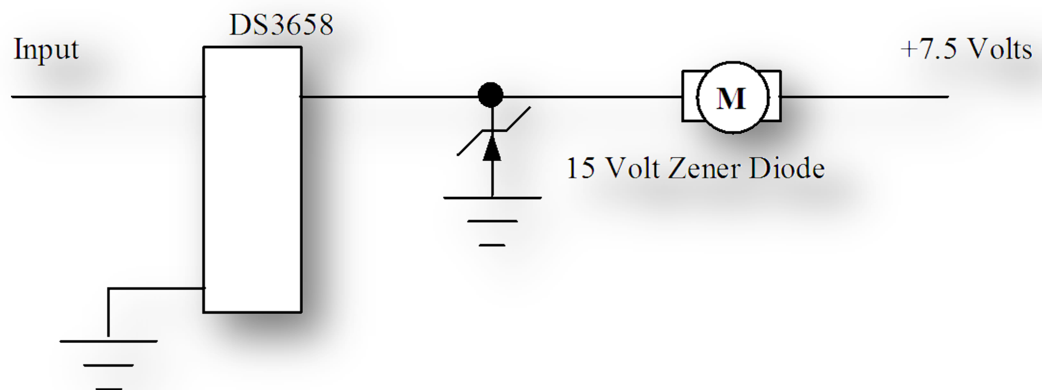
Assignment: Complete the following exercises:

Basic Waveforms

2.1) Set the 'scope up for dual trace and examine the waveforms at the input and output of the DS3658. Draw a neat and readable representation of the two wave-forms. Include at least 2 full cycles. Label the parts of the waveform to show the active (driven) portion of the waveform, the decay period as the motor field collapses, the peak of the inductive kickback, and the back-EMF generated by the motor. Do this at 2 duty cycle levels (i.e. 20% and 80%). Make a separate scope face drawing for each. Note the different back-EMF values at the different motor speeds. To best see the back-EMF, you might want to draw a composite of two waveforms, one with the motor stalled (hold the shaft to keep it from turning) and one with the motor turning.

Using a Zener in lieu of the kick-back diode

2.2) Disconnect the clamp diode from the circuit and replace it with the 15 volt zener in a configuration like the one shown below. **Note**: the band on the zener diode body corresponds to the horizontal bar in the zener symbol. Repeat the labeled waveform drawing from Part 2.1. Contrast the decay times for the 2 techniques; be specific about durations. To do this you will probably need to use a different time scale on the 'scope. This portion will be most clear if you examine the alternatives at a very low duty cycle (i.e. active for 1 part in 256). This way the effects of back EMF do not obscure the decay time. From this exercise you can see how a zener protects the output transistor by limiting the peak voltage to the zener voltage, and at the same time minimizes the decay time of the collapsing field. It is important to minimize this decay time in order to make the duty cycle response of the motor as linear as possible. At very high duty cycles it would be possible for a long delay time to completely overlap the un-driven portion of the waveform.



The Limits of Software control

☐ 2.3) Using the oscilloscope and/or the counter-timer, determine the frequency of the duty cycle waveform that you are generating. What is the frequency?

☐ 2.4) Can you find a way to increase the operating frequency? (Look at how to shorten your cycle time to the minimum.)

☐ 2.5) What happens to the upper frequency limit if we only have 7 bits of resolution? How about 10 bits? We are making a trade-off between resolution and frequency of operation.

☐ 2.6) Given a resolution, what is the key parameter in determining the total period of the PWM signal?

☐ 2.7) Can your solution achieve a full range of operation? (i.e. What happens if you try to get 0% or 100% duty cycle). If it can't, don't try and fix it, just explain why it won't work.

☐ 2.8) For a given duty cycle, what are the sources of error in the approach you have chosen? (i.e. if you ask for 50% duty cycle do you get it? ...theoretically, that is, you probably will not be able to measure the error.)

In the report: Several scope face drawings are requested; please make your drawings legible. Using different colors for different waveforms is helpful. Include the discussion requested in Part 2.2. Be sure also to include in the lab report your answers to the all the questions in Parts 2.3-2.8. **To ease the grading task please quote the question you are answering before your answer. This way the grader can read the question you think you are answering followed by your answer.**

Part 3 Interfacing to a Stepper Motor

Reading: CKO Ch. 26, the CMPE118 A3949 H-Bridge Driver Module documentation, STEPS12 software module documentation.

Assignment: You are to design the necessary software to run on the 9S12C32 to drive the supplied stepper motor in Full Drive, Wave Drive, and Half-Step Drive modes. After demonstrating this to the TA or tutors, you will use a hardware stepper driver to drive the same motor.

Set-Up:

☐ 3.1) For this part, you should use the CMPE118 A3949 Dual H-Bridge Driver Module board.

☐ 3.2) You will need to connect power and ground to J3 of the CMPE118 H-Bridge Module, which will in turn supply current to the stepper motor. Note that you must supply it using the +12V source. J4 of the CMPE-118 H-Bridge Module will need to interfaced to the PortA on 9SC32 as follows to work properly with the STEPS12 module:

DA (J4.4) connected to PA1, EA (J4.3) to PA0, DB (J4.8) to PA3, and EB (J4.7) to PA2. Note: the motor drive board and I/O board need to have a common ground (back at the power supply).

☐ 3.3) You will have to probe the motor wires to determine which wires correspond to which coils and then connect those wires to the proper motor outputs on the Drive Module. Use the STEPS12 software module to control the Drive Module and ensure the motor rotates properly with the provided modules. You will need to experiment with the motor coil polarities to get the motor to rotate. If at first you don't succeed, try, try again (there are only a few possibilities

☐ 3.4) Now, edit the STEPS12 module to drive the motor using Wave Drive and Half-Step Drive. You will have to edit the state machine outputs and add additional states.

☐ 3.5) After verifying your motor can be driven all three ways, increase the frequency until it stops stepping. How fast can you go under each mode of operation? Note that having a load on the motor helps here (a finger should be fine).

☐ 3.6) Now run the motor forward a complete revolution then in reverse for a complete revolution—did you lose steps? How fast can you operate the motor without losing steps (for accurate positioning)?

☐ 3.7) After demonstrating the above code to the TAs/Instructors and getting signed off, get one of the CMPE-118 Stepper Motor Driver boards. These boards have only two control signals, step and direction, and a jumper to select Full or Half-Step Drive, as discussed in the attached description of the module. You will now have to write code to produce these signals.

☐ 3.8) Repeat parts 3.5 and 3.6 using the stepper driver under both Full and Half-Step operation. How did this compare to your software driver? Demonstrate the working stepper module to a tutor or TA for check off.

In the report: Include the wiring diagram of the circuit you used, a copy of the code that drives the stepper motor, discussion of the performance differences between hardware and software stepper drivers, and the sign-off sheet from the coach. Be sure to indicate on the diagram which bits of which ports of the 9S12C32 you used.

Part 4 Interfacing to a DC Motor and H-Bridge

Reading: The CMPE 118 A3949 Dual 2.8A H-Bridge Module documentation.

Assignment: You are to duplicate the functionality of Part 1, this time using the H-Bridge to drive the motor. You will also need to add a little complexity. You should include a provision to change the direction of rotation of the motor. When you have completed this task, get a tutor or TA to sign off on its function. For this section of the lab, you should use the supplied libraries providing PWM output.

Set-Up:

☐ 4.1) You will need to add a power connector to supply power to the H-bridge on the A3949 Dual 2.8A H-Bridge Module. Connect power and ground to J3 of the module.

☐ 4.2) You will need to supply enable and direction inputs to the module. Remember to bring ground back to the power supply for both the module and I/O board.

☐ 4.3) Experiment and determine the maximum and minimum frequencies you can use to properly drive your motor at a given duty cycle.

In the report: Include the wiring diagram of the circuit you used, a copy of the code that drives the DC motor and the sign-off sheet from the coach. Be sure to indicate on the diagram which bits of which ports of the 9SC32 you used.

Part 5 Driving an R/C Servo Motor

Reading: CKO Ch. 27 (especially section 27.3), RC Servo tutorial at: http://www.societyofrobots.com/actuators_servos.shtml

Assignment: You are going to drive an R/C servo with position feedback. By adjusting the duty cycle of a fairly slow PWM signal to the servo, you command a position for the head to go to and hold. Just as you

read an analog value of 0-5V and adjusted the speed of the DC motor, you will now read that same voltage and position the servo motor to correspond to the voltage. When reading 2.5V on the A/D, the servo should be in the neutral position, with 0V positioning it at one end of the range and 5V positioning it at the other.

Set-Up:

☐ 5.1) **Important note**: the R/C servos can only take +5V, so DO NOT connect them to a higher power supply, you will kill them. Power the servos off of +5V from the power supplies, again making sure NEVER TO EXCEED THIS—you will break the feedback circuitry around the motor. Eventually (like in your final project) you may need to use a linear regulator to get +5V from your batteries, but you will be wasting lots of power as heat—you will need a heatsink.

☐ 5.2) Connect the servo on the motor drive lab board to +5 on the RED wire, and GND on the BROWN wire. Connect a wire from your I/O board to the ORANGE signal wire (your I/O board and the servo need to have a common ground connection back to the power supply).

☐ 5.3) The servo is a position feedback device. That is, it will go to and hold a specific position that ranges from about +90 degrees to -90 degrees, with a 1.5ms pulse being CENTER. We will provide a routine to produce this signal, and it will be up to you modify it further to produce a range of duty cycles.

☐ 5.4) Experiment and determine the minimum change of the pulse high time that results in motion of the servo head.

☐ 5.5) Experiment and determine the minimum and maximum duty cycles that the servo is able to respond to. How do these limits and the neutral duty cycle correspond to head position?

☐ 5.6) Use the potentiometer from Part 1, after first checking that it stays within the range 0-5V, to command a position of the servo. When sweeping from 0-5V the servo head should follow from -90 degrees to +90 degrees. If the servo introduces noise on your power and this is affecting your A/D readings, bypass your potentiometer with a capacitor to smooth it out.

☐ 5.7) Show the completed servo control to the tutors or TA for your final checkoff.

In the report: Include the wiring diagram of the circuit you used, a copy of the code that drives the servo motor, a discussion of the behavior of the servo as relating to questions 5.4 and 5.5, and the sign-off sheet from the coach. Be sure to indicate on the diagram which bits of which ports of the 9SC32 you used.

Hints on working this assignment

When you go into the lab, follow a well thought-out and systematic approach to testing both the hardware and the software. It is a good idea to get into the habit of testing your software, as much as possible, separately from testing your hardware. This makes trapping failures much easier (and debugging is ALWAYS the longest task).

If you really did your preparation properly, you should come into the lab ready to build and test the hardware in Part 1. **If you have spent more than one hour on any single task, after coming in prepared as described above, something is wrong! STOP and ask a TA or your neighbor to take a look at what you are doing.** Often a new look will spot simple problems that you've missed.

And, needless to say, better preparation for this lab (reviewing the reading, examining the data sheets, planning your software) will greatly cut down on the amount of time in the lab. It is very inefficient to try to do the prep at the same time as doing the lab.

Lab #3

Time Summary

Be sure to turn this in with your lab report

This information is being gathered solely to produce statistical information to help improve the lab assignments.

Pre-Lab	Preparing Outside of the lab _____________	In the lab working this part___________
Part 1	Preparing Outside of the lab _____________	In the lab working this part___________
Part 2	Preparing Outside of the lab _____________	In the lab working this part___________
Part 3	Preparing Outside of the lab _____________	In the lab working this part___________
Part 4	Preparing Outside of the lab _____________	In the lab working this part___________
Part 5	Preparing Outside of the lab _____________	In the lab working this part___________
Report	Preparing the Lab Report _______________	

Lab #3

Partner Evaluation

Email the Professor (only) at mailto:elkaim@soe.ucsc.edu with a subject line of:
"CMPE118 Lab 3 Partner Evaluation" and include:

(1) Partner's name:

Rate them on a scale of 1 (bad) to 5 (excellent) on the following areas:

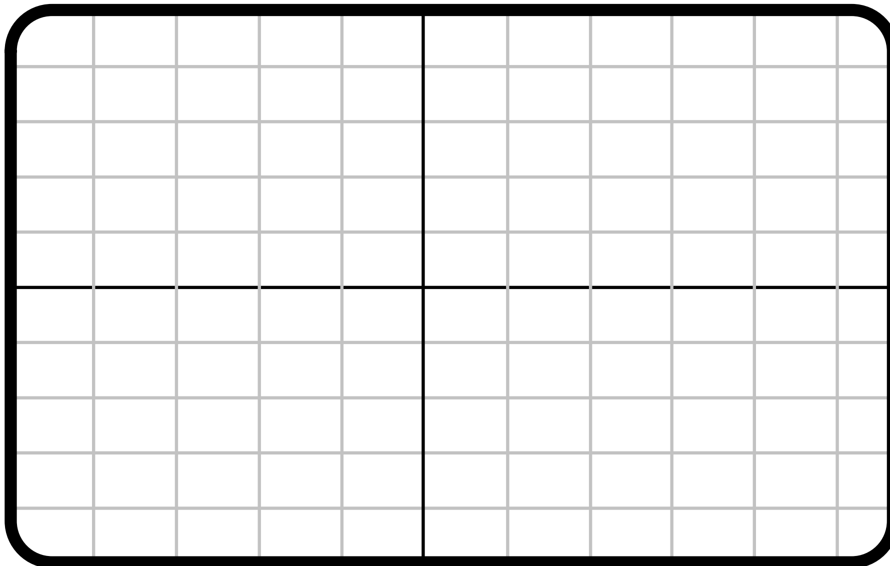
(2) Knowledge:
(3) Prep:
(4) Work:
(5) Neatness/Cleanup:

This will be used to help us match partners for the project, and is extremely important. Failing to send this in will result in a 20% penalty on your lab grade.

X Scale

Ch1 Scale

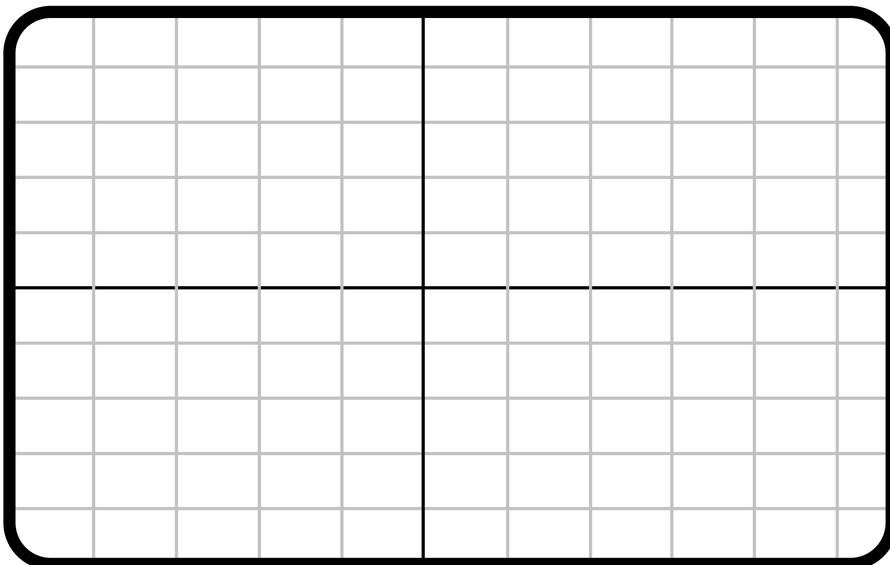
Ch. 2 Scale



X Scale

Ch1 Scale

Ch. 2 Scale



X Scale

Ch1 Scale

Ch. 2 Scale

The CMPE118 HC12 IO Board

Background:

The 9S12C32 Module is a single-board computer produced by Technological Arts, which integrates a Motorola 9S12C32 microcontroller, 2K of RAM scratchpad memory, 32K of EEPROM program memory, RS232 level-shifters, voltage regulation, and CPU voltage monitoring, all into a tidy little PC board.

CMPE118 uses the 9S12C32 as the single board computer platform throughout the lab exercises and final project. Since the 9S12C32 wasn't designed with just CMPE118 in mind, it's not *exactly* what we want, so we've designed and manufactured the CMPE118 HC12 IO Board to get us closer to the ideal.

The **HC12 IO Board** has 4 major functions:

1. Bring all connections from the 9S12C32's main header out so that they are easy to access and clearly labeled
2. Provide voltage regulation and power supply protection circuitry
3. Provide over-voltage and over-current protection circuitry on all port pins.

Using the CMPE118 HC12 IO Board:

Using the CMPE118 HC12 IO Board should be straightforward. All the connections that will typically be made to the board are located around the edge of the board, and are clearly labeled and logically grouped.

Digital Input/Output and A/D Converter Inputs:

Following is a description of the I/O available on the HC12 IO Board:

Port A (J7), Port B (J8), Port E (J9), and Port P (J10)
Port A, B, E, and P all act as general purpose input and output ports. In CMPE118, we use PTA0-PTA3 to provide digital I/O ports for the STEPS12 module.

Port AN/AD (J12):
Port AD on the newer 9S12C32 and acts as general purpose I/O pins and also analog inputs for the analog to digital converter.

Port T (J11):
Port T on the 9S12C32 acts as general purpose input and output ports and can be configured for a wide variety of uses. In CMPE118, we use it mostly to provide PWM and Pulse signals with our PWM and PLS software libraries. PWM outputs are limited to PT0-PT4

All port pins available for the user on the CMPE118 HC12 IO Board are protected from over-voltage and over-current conditions by the use of series resistors (to limit current) and clamping diodes (to limit voltage). To see how this is implemented, please refer to the attached schematic.

Power Regulation and Protection Circuitry:

The CMPE118 HC12 IO Board must be powered by connecting IN to a voltage supply where:

+6VDC < IN < +18VDC

This input voltage is regulated down to +5VDC for the 9S12C32 and the protection circuitry by an LM2940CT-5.0 low drop-out voltage regulator. Reverse-polarity protection is provided by a low-forward-voltage-drop diode (D1) on the positive Vin terminal. ESD protection and protection against supplying the Interface PCB with extremely high input voltage is provided by a 27V-trip transient voltage suppressor (TVS), at D3. In addition, the current flow for the entire board is limited to 500mA by a fuse.

The CMPE118 DS3658 Driver Module

Rev. 2

Dept. of Computer Engineering, UCSC

Background:

The DS3658 is a four-channel, high-current peripheral driver, manufactured by National Semiconductor. It integrates logic-level inputs, four open-collector output channels (each capable of sinking a maximum of 600mA), and internal kickback diodes.

The CMPE118 DS3658 Driver Module PCB provides a convenient and robust interface to the DS3658. Separate connectors provide access to the logic-level inputs, high-current switched outputs, kickback diodes, and the Module's power supply input.

Using the CMPE118 DS3658 Driver Module:

In order to make use of the CMPE118 DS3658 Driver Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J1):

Access to the logic-level inputs to each of the four channels of the DS3658 is provided through J1. In order to turn on a channel's high-current, open-collector output, a logic-level high must be supplied to that channel's input pin. The attached data sheet for the DS3658 provides complete specifications for the drive requirements of these inputs.

Logic ground connections must also be made through this connector. This will ensure that the ground of the (off-board) logic command circuitry and the ground of the CMPE118 DS3658 Driver Module agree.

The pinout of J1 is as follows:

J1	Connection
pin 1	logic ground
pin 2	Channel A input
pin 3	Channel B input
pin 4	Channel C input
pin 5	Channel D input
pin 6	logic ground

High-Current Switched Outputs (J2):

Access to the open-collector switched outputs is provided through the screw-terminal connector at J2. Each channel is capable of sinking a maximum of 600mA, and multiple channels may be used in parallel to provide additional current capability (see the attached data sheet). The pinout of J2 is as follows:

J2	Connection
pin 1	Channel A output
pin 2	Channel B output
pin 3	Channel C output
pin 4	Channel D output

Integrated Kickback Diodes (J3):

Access to the DS3658's integral kickback diodes is provided through the screw-terminal connector at J3. There are 2 kickback diodes built into the DS3658, each protecting two of the IC's four channels. If they are to be used to protect the circuitry of the DS3658 from being subjected to excessive voltages at the open-

collector outputs, external connections from the cathode of the internal diodes to the power supply of the connected switched load are required. If this protection is not needed, the diodes may be left unconnected.

J2 has the following pinout:

J2	Diode	Protects these outputs
pin 1	CLAMP 1	Channel A & Channel B
pin 2	CLAMP 2	Channel C & Channel D

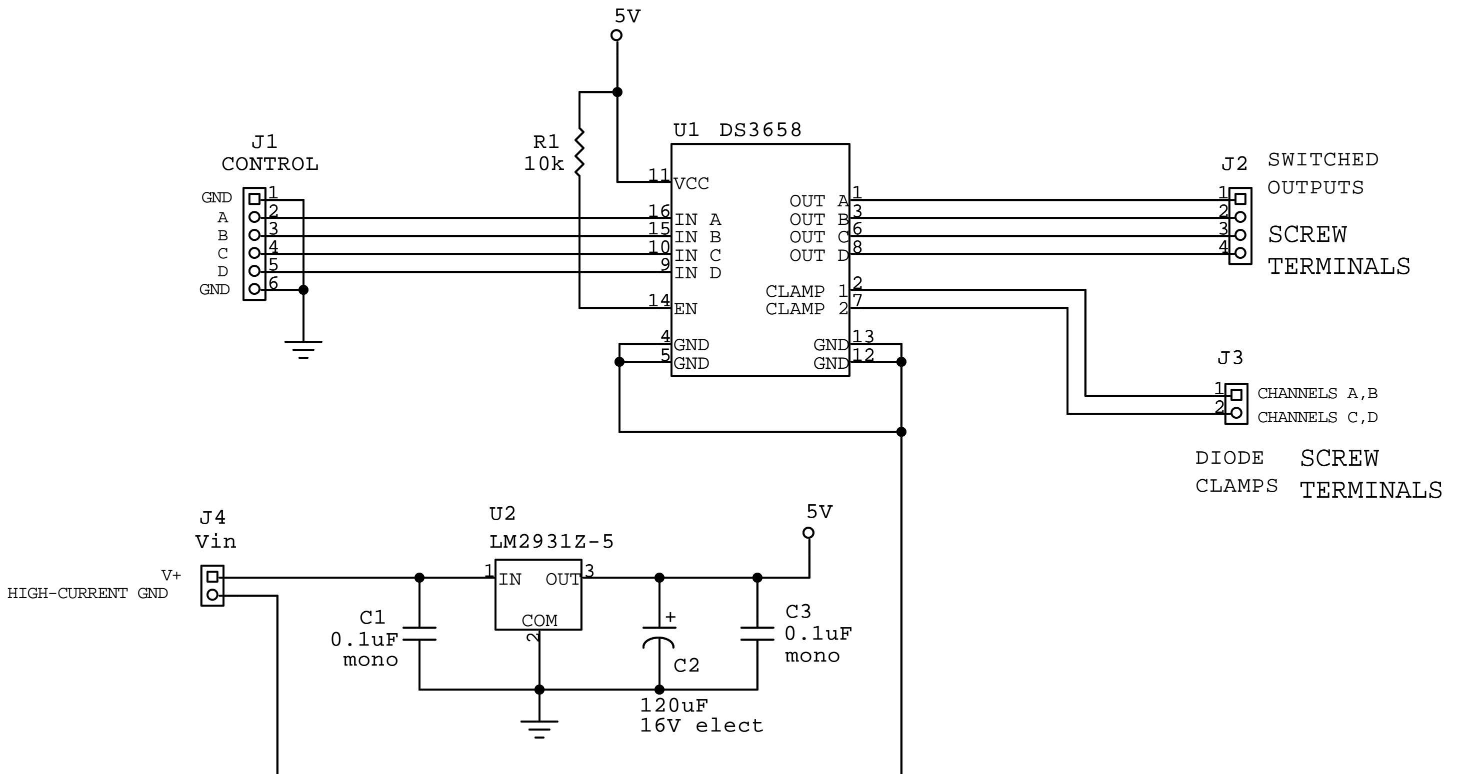
DS3658 Power Supply and High-Current Ground Connector (J4):

J4, pin 1: The DS3658 requires a power supply for its logic circuitry. Provisions for this are made through J4, pin 1. An LM2931Z-5.0 low drop-out, 100mA voltage regulator is provided on the DS3658 Driver Module PCB so that any voltage between 5.6V (minimum) and 18V (maximum) may be supplied to J4, pin 1. This makes the DS3658 module much easier to use, since it includes its own voltage regulation and does not require an externally regulated +5V supply.

DS3658 Power Supply Requirements: +5.6V < Vin (J4, pin 1) < +18V

J4, pin 2: Since the DS3658 is capable of switching four channels at a maximum of 600mA each, care must be taken in the methods employed returning this substantial current to ground. For this reason, a separate high-current ground connection is available at J4, pin 2. A separate connection should be made between J4, pin 2 and the power supply of the load(s) connected to the IC's outputs. This will ensure that the logic power supply maintains a clean ground.

CMPE118 DS3658 Driver Module Schematic:



The CMPE 118 A3949 Dual 2.8A H-Bridge Module

Dept. of Computer Engineering, UCSC

Background:

Allegro's A3949 is a high-current single H-bridge driver with integrated control logic. It comes in 16-pin SOIC and TSSOP packages. Internal circuit protection includes thermal shutdown with hysteresis, undervoltage monitoring, and crossover current protection.

The CMPE118 A3949 Dual 2.8A H-bridge Module provides a convenient and robust interface for two A3949s. Separate connectors provide access to the logic-level inputs, H-bridge output connections, and the IC's power supply. This module makes use of a 7805 regulator for pull-ups and a TL7726QP latch for protecting the digital inputs. External Clamping diodes are included to complement the A3949s internal diodes and protect against inductive kickback.

The data sheet for the A3949 motor driver 2.8A H-bridge chip can be found on the Allegro.com website at: http://www.allegromicro.com/en/Products/Part_Numbers/3949/3949.pdf

Using the CMPE118 A3949 Dual 2.8A H-Bridge Module:

In order to make use of the CMPE118A3949 Dual H-Bridge Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J4):

Access to the logic-level inputs of the A3949 is provided through J4. Directional and enable control for each of the two H-Bridges is specified through these connections.

There is no logic ground on this board to reduce noise. As such, care must be taken to provide a common ground.

The pinout of J4 is as follows (all other pins are NO CONNECT):

J4	Connection
Pin 3	Enable A
Pin 4	Direction A
Pin 7	Enable B
Pin 8	Direction B

Motor/Load Connections (J3):

Connections of one or two independent motors or similar loads should be made through the screw-terminal connector located at J3. The pinout of J3 is as follows:

J3	Connection
pin 1 & pin 2	motor/load A
pin 3 & pin 4	motor/load B

Power Supply and High-Current Ground Connector (J1):

J1, pin 1: The A3949 requires a power supply for both the motors as well as its logic circuitry. Provisions for this are made through J1, pin 1, which includes a Schottky diode to provide reverse bias protection. A 7805 low drop-out, 100mA voltage regulator is provided on the Module PCB to clamp the digital inputs between -0.2V and 5.2V. The chip will operate between 8V (minimum) and 36V (maximum). This makes the A3949 Module much easier to use, since it includes its own voltage regulation and does not require an

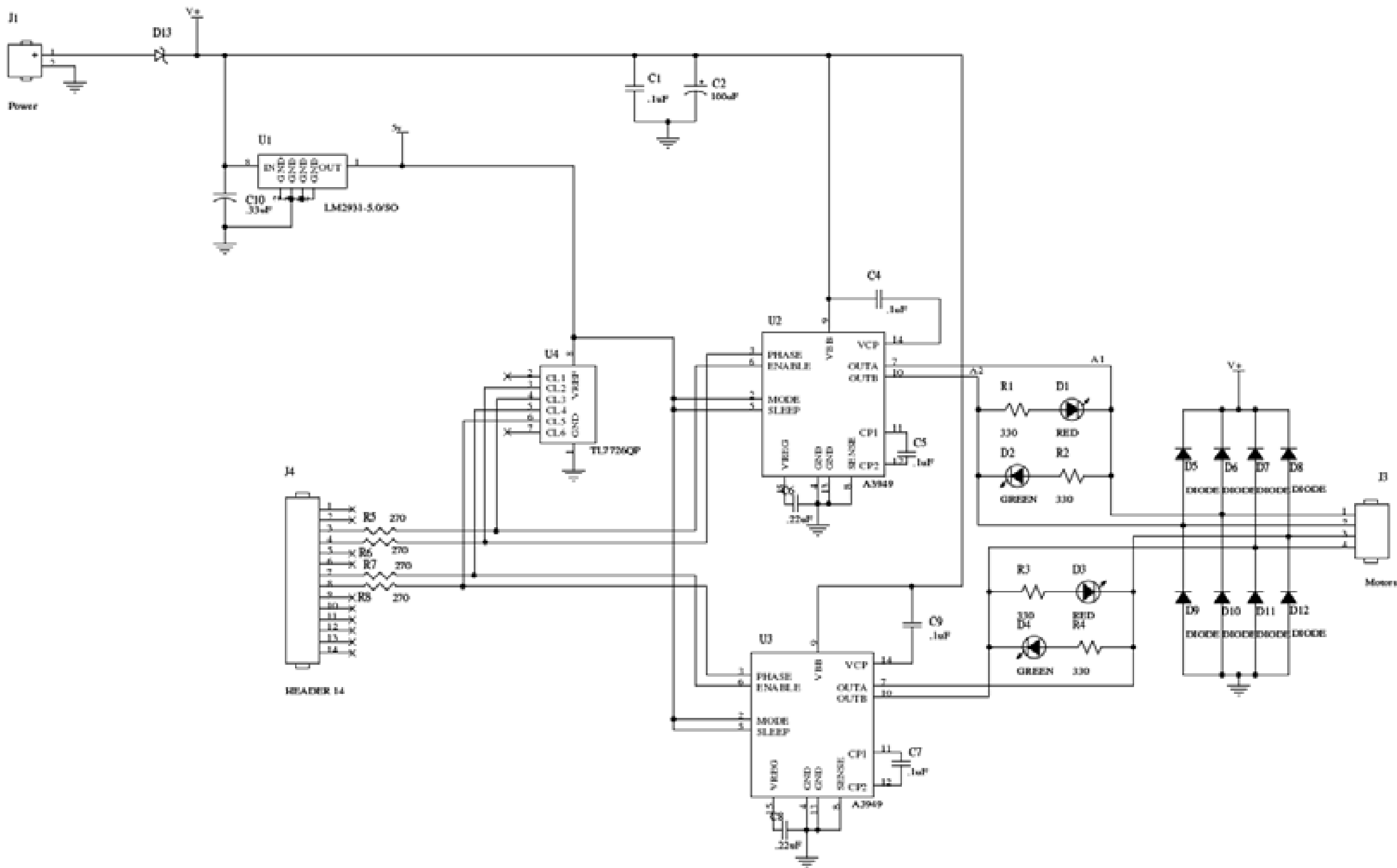
externally regulated +5V supply. Note that there is no ground connection between the input port (J4) and the power input (J1), for noise reasons. The source of the input signal should be connected back to ground at the power supply or batteries.

A3949 Power Supply Requirements: ***+8V < Vin (J1, pin 1) < +36V***

J1, pin 2: Since the module is capable of switching two H-Bridge channels at a maximum of 2.8A each, care must be taken in the methods employed returning this substantial current to ground. For this reason, a separate high-current ground connection is available at J1, pin 2. A separate connection should be made between J3, pin 2 and the power supply of the motors/loads connected to the IC's outputs. This will ensure that the logic power supply maintains a clean ground.

J1	Connection
Pin 1	Power (8V to 36V)
Pin 2	High Current Ground

CMPE118 A3949 Dual 2.8A H-Bridge Module Schematic:



J1
Power
D13
V+
C1 .1uF
C2 100uF
U1
IN GND GND GND GND OUT
LM2931-5.0/SO
C10 .33uF
5v
C4 .1uF
U2
PHASE ENABLE MODE SLEEP VBB VCP OUTA OUTB CP1 CP2 VREG GND GND SENSE
A3949
C5 .1uF
C6 .22uF
U4
CL1 CL2 CL3 CL4 CL5 CL6 VREF GND
TL7726QP
J4
HEADER 14
R5 270
R6 270
R7 270
R8 270
A1
A2
R1 330
D1 RED
D2 GREEN
R2 330
R3 330
D3 RED
D4 GREEN
R4 330
V+
D5 D6 D7 D8 DIODE DIODE DIODE DIODE
D9 D10 D11 D12 DIODE DIODE DIODE DIODE
J3
Motors
U3
PHASE ENABLE MODE SLEEP VBB VCP OUTA OUTB CP1 CP2 VREG GND GND SENSE
A3949
C9 .1uF
C7 .1uF
C8 .22uF

ASL, UCSC	Engr.: Max Dunne	
Title: Mechatronics DC Motor Board		
Size: A	Project: motor board.dsn	Rev: 1.2
Date: Monday, January 26, 2009	Sheet 1 of 1	

The CMPE 118 A3982 2.0A Stepper Motor Module

Dept. of Computer Engineering, UCSC

Background:

Allegro's A3982 is a complete microstepping motor driver with built-in translator for easy operation. It is designed to operate bipolar stepper motors in full-, half-, quarter-, and sixteenth-step modes, with an output drive capacity of up to 35 V and ±2 A. The A3982 includes a fixed off-time current regulator which has the ability to operate in Slow or Mixed decay modes. It comes in 24-pin SOIC and TSSOP packages. Internal circuit protection includes thermal shutdown with hysteresis, undervoltage monitoring, and crossover current protection.

The CMPE118 A3982 Stepper Motor Module provides a convenient and robust interface for the A3982. Separate connectors provide access to the logic-level inputs, Stepper motor coil output connections, and the IC's power supply. This module makes use of a 7805 regulator for pull-ups and a TL7726QP clamp for protecting the digital inputs.

The data sheet for the A3982 translator and driver chip can be found on the Allegro.com website at: http://www.allegromicro.com/en/Products/Part_Numbers/3982/3982.pdf

Using the CMPE118 A3982 2.0A Stepper Motor Module:

In order to make use of the CMPE118 A3982 Stepper Motor Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J5):

Access to the logic-level inputs of the A3982 is provided through J5. Enable, Direction, and STEP control for the stepper driver is specified through these connections.

There is no logic ground on this board to reduce noise. As such, care must be taken to provide a common ground back at the power supply or battery.

The pinout of J5 is as follows (all other pins are NO CONNECT):

J5	Connection
Pin 3	Step
Pin 4	Direction
Pin 5	Enable

Full/Half-Step Connections (J3):

The A3982 is capable of both full step (2 coils energized) or half-step (1-2 coils energized); this is set with the jumper J3. If it is OPEN, then the chip is set to FULL STEP mode. If a jumper is placed across the pins of J3, then the chip is set for HALF STEP mode.

Motor/Load Connections (J2):

Connections to the coils of the stepper motor should be made through the screw-terminal connector located at J2. The pinout of J2 is as follows:

J2	Connection
pin 1 & pin 2	Phase A
pin 3 & pin 4	Phase B

Power Supply and High-Current Ground Connector (J1):

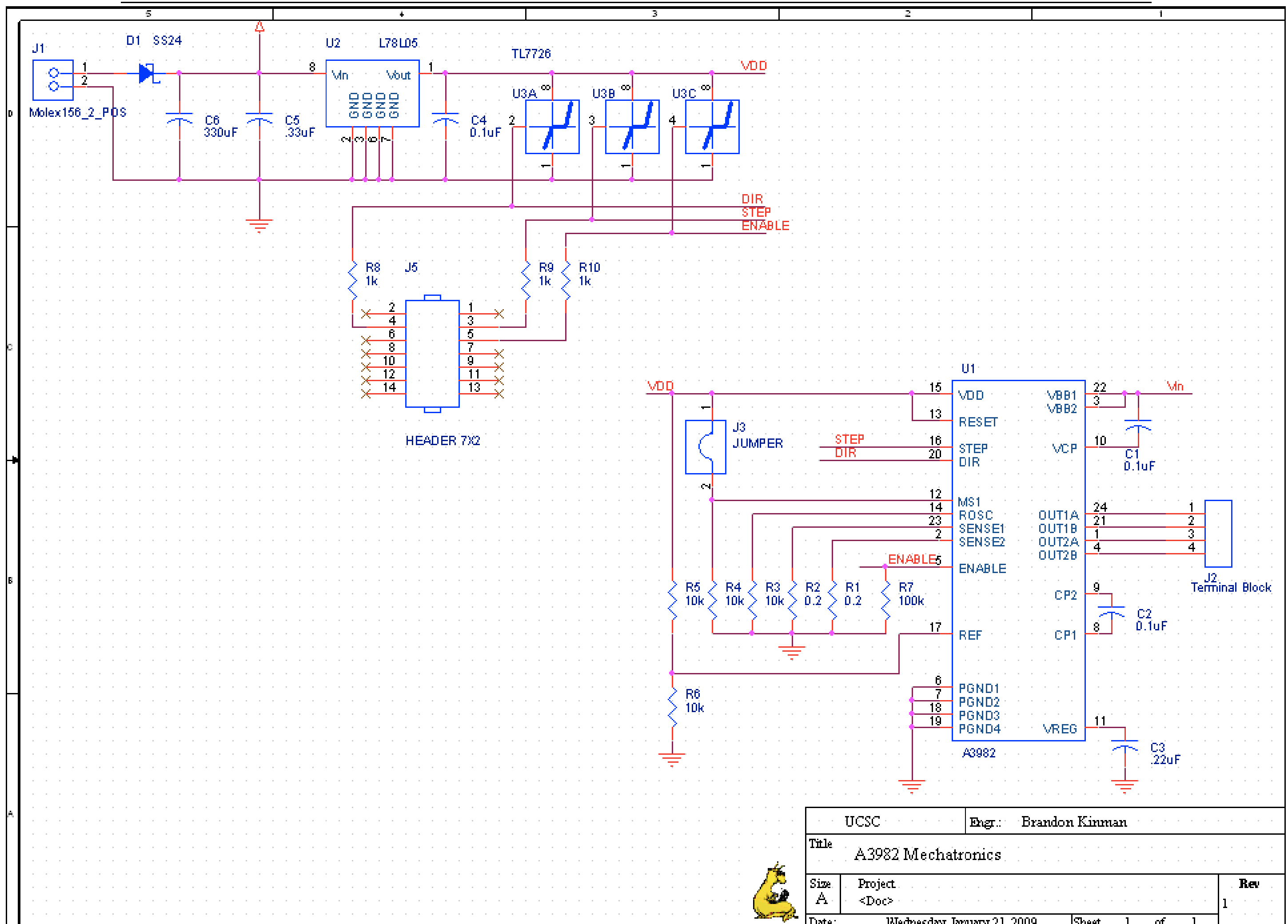
J1, pin 1: The A3982 requires a power supply for both the motors as well as its logic circuitry. Provisions for this are made through J1, pin 1, which includes a Schottky diode to provide reverse bias protection. A 7805 low drop-out, 100mA voltage regulator is provided on the Module PCB to clamp the digital inputs between -0.2V and 5.2V. The chip will operate between 8V (minimum) and 36V (maximum). This makes the A3982 Module much easier to use, since it includes its own voltage regulation and does not require an externally regulated +5V supply. Note that there is no ground connection between the input port (J5) and the power input (J1), for noise reasons. The source of the input signal should be connected back to ground at the power supply or batteries.

A3982 Power Supply Requirements: ***+8V < Vin (J1, pin 1) < +36V***

J1, pin 2: Since the module is capable of switching both coils on the stepper motor to 2.0A, care must be taken in the methods employed returning this substantial current to ground. For this reason, a separate high-current ground connection is available at J1, pin 2. A separate connection should be made between J3, pin 2 and the power supply or batteries. This will ensure that the logic power supply maintains a clean ground (the microcontroller must have a common ground to the power supply as well).

J1	Connection
Pin 1	Power (8V to 36V)
Pin 2	High Current Ground

CMPE118 A3982 2.0A Stepper Motor Module Schematic:



Purpose:

This module provides a set of interface functions to perform a simplified initialization of the A/D converter system on the Freescale MC9S12C32 microcontroller. The initialization function should be called before any use is made of Port AD (J12). After the initialization is complete, reads of the analog input pins should be made using the ADS12_ReadADPin() function. Reads of digital input pins on the AD port should be made by reading from PTAD. Writes to outputs defined on port AD should be performed by making assignments to PTAD.

Header:

The header file ADS12.h should be included in any module wishing to use the functions provided by this module.

Initialization

Function:

ADS12ReturnTyp ADS12_Init(char * modeString)

Parameters:

char [9] modeString

A null terminated string of 8 ASCII characters to describe the mode of each of the pins on Port AD. The legal values are: I for digital input, O for digital output, A for analog input. The string positions, reading left to right, correspond to the pins MSB to LSB (modeString[0]=MSB, modeString[7]=LSB)

Returns:

ADS12_Err if the input string is malformed

ADS12_OK if the mode string was OK

Description:

Initializes Port AD data direction register & ATDDIEN for digital I/O and the A/D converter to multi-channel, continuous conversion.

Notes:

Assumes a 24MHz bus clock, but simply sets the default values for now. Enforces a single block of A/D channels even though it is possible to make a contiguous sequence that occupies a non-contiguous block. For example, channels 6,7,0,& 1 are sequence contiguous but occupy 2 blocks so would be rejected by this code.

Usage:

if (ADS12_Init("OOIIAAAA") == ADS12_OK)

would initialize the A/D converter and port AD with bits 6 & 7 as outputs, 4 & 5 as inputs and bits 0-3 as analog inputs and test to see that the initialization was successful.

Analog Pin Read

Function:

short ADS12_ReadADPin(unsigned char pinNumber)

Parameters:

unsigned char PinNumber

the Port AD pin number to read the analog value from

Returns:

the A/D conversion result for the selected pin (10 bits right justified) if the pin number is legal, -1 otherwise

Description:

Reads and returns A/D conversion result from the appropriate A/D converter register.

Usage:

ADS12_ReadADPin(0)

Would read the analog value on bit 0 of port AD.

Purpose:

This module provides a set of interface functions to perform a simplified initialization of the Pulse Width Modulation system on the Freescale MC9S12C32 microcontroller. The initialization function should be called before any use is made of the other functions. After the initialization is complete bits 0, 1 & 2 of Port T on J11 are dedicated to the PWM output functions for PWM channels 0, 1 & 2.

Header:

The header file PWMS12.h should be included in any module wishing to use the functions provided by this module.

Initialization

Function:

void PWMS12_Init(void)

Parameters:

None

Returns:

Nothing

Description:

Initializes the PWM subsystem for all three channels. Gives the PWM subsystem control of Port T bits 0,1 & 2. Sets the initial PWM period to 2mS. Sets the Duty Cycle resolution to 1% and output duty cycle to 0%.

Notes:

The duty cycle resolution is fixed at 1% for this library.

Usage:

PWMS12_Init();

would initialize the PWM subsystem.

Set Duty Cycle

Function:

signed char PWMS12_SetDuty(unsigned char dutyCycle, unsigned char channel)

Parameters:

unsigned char dutyCycle
    a number between 0 & 100, the new duty cycle

unsigned char channel
    PWMS12_CHAN0, PWMS12_CHAN1 or PWMS12_CHAN2 to specify which channel's duty cycle to set.

Returns:

Signed char
    PWMS12_ERR if requested channel does not exist or the duty cycle is over 100%.
    PWMS12_OK otherwise

Description:

Sets the output duty cycle on the specified channel. Setting the duty cycle to 0% (full off) or 100% (full on) allows you to control the PWM channel as if it were a digital output.

Usage:

PWMS12_SetDuty(50, PWMS12_CHAN0)

Would set channel 0 (Port T bit 0) to output a 50% duty cycle signal at the previously programmed period.

Set PWM Period

Function:

signed char PWMS12_SetPeriod(unsigned short newPeriod, unsigned char group)

Parameters:

unsigned char newPeriod

a defined constant to specify the new period. Do not enter a number in mS or any other time units. This constant encodes the values needed to program the 9S12C32 timer system. Use only the pre-defined constants documented here or in PWMS12.h.

unsigned char group

PWMS12_GRP0 to specify group 0, which applies to channels 0 & 1 or
PWMS12_GRP1 to specify group 1, which applies to channel 2.

Returns:

Signed char

PWMS12_ERR if requested group is invalid.
PWMS12_OK otherwise

Description:

Sets the output period on the specified channel group.

Notes:

Takes newPeriod directly. Makes no sanity check on the period being programmed.

Usage:

PWMS12_SetPeriod(PWMS12_PERIOD_4MS, PWMS12_GRP0)

Would set channel group 0 (channels 0 & 1) to output a duty cycle with a 4mS period.

Disable PWM Subsystem

Function:

void PWMS12_End(void)

Parameters:

None

Returns:

Nothing

Description:

Disables the PWM subsystem for all three channels. Gives the control of Port T bits 0,1 & 2 back to the DDR and port registers.

Usage:

PWMS12_End();

would disbale the PWM subsystem.

Defined Constants For Period

```
#define PWMS12_4000US 20510
#define PWMS12_3300US 8390
#define PWMS12_2500US 8342
#define PWMS12_2000US 16414
#define PWMS12_1950US 4330
#define PWMS12_1650US 4294
#define PWMS12_1250US 4246
#define PWMS12_1000US 12318
#define PWMS12_825US 198
#define PWMS12_650US 4174
#define PWMS12_500US 8222
```

```c
/****************************************************************************
 Module
         steppers12.h

 Revision
         1.0.1

 Description
         Header File for the Stepper Motor Driver Module

 Notes


 History
 When           Who What/Why
 -------------- --- --------
 02/01/06 13:34 rjk Began Coding
****************************************************************************/

#ifndef STEPS12_H
#define STEPS12_H



#define STEPS12_RATE_10MS 0x49// set OSCCLK divider to 10*2^13 (100Hz)
#define STEPS12_RATE_1MS 0x17// set OSCCLK divider to 8*2^10 (1000Hz)

void STEPS12_Init(long Period, unsigned char Direction, long Steps);
void STEPS12_RTI_Resp(void);
void STEPS12_Stop(void);

#endif   /* STEPS12_H */
/*------------------------------ End of file ------------------------------*/
```

```c
#define TEST
/****************************************************************************
 Module
     steps12.c

 Description
     This is a module to drive a 2-phase stepper motor using the
     CMPE118 LM298 dual H-Bridge Module.  The module uses a
     periodic interrupt to act as a heartbeat.  The stepper is then advanced
     at a multiple of that frequency.
 Notes

 History
 When           Who     What/Why
 -------------- ---     --------
 02/01/06 13:34 rjk      Began Coding
****************************************************************************/

/*---------------------------- Include Files ----------------------------*/
#include <hidef.h>         /* common defines and macros */
#include <mc9s12c32.h>     /* derivative information */
#include <steps12.h>
#include <stdio.h>
#include <termio.h>

/******* GLOBALS  ****************************/

#define MAX_STEPS 4 // total number of step states for driving FULL STEP
static unsigned char StepDirection;
static long StepsLeft;
static int StepState;
static long StepPeriod;
static long Counter;
static unsigned char CurrentlyStepping;

/****************************************************************************
 Function
     STEPS12_Init

 Parameters
     Period: Number of milliseconds between each step (must be an integer);
     Direction: 0 for CW or non-zero for CCW
     Steps: Number of steps to advance motor

 Returns
     None.

 Description
     Initializes the RTI module for OSCCLK=8MHz and sets global variables.
 Notes

 ****************************************************************************/
void STEPS12_Init(long Period, unsigned char Direction, long Steps) {
    RTICTL = STEPS12_RATE_1MS; // set OSCCLK divider
    CRGFLG = CRGFLG_RTIF_MASK;              /* clear any pending int */
    CRGINT |= CRGINT_RTIE_MASK;             /* enable the RTI int */

    StepDirection=Direction;
    StepsLeft=Steps;
    StepState=1;
    StepPeriod=Period;
    Counter=1;
    CurrentlyStepping=1;

    DDRA|=0x0F; // First 4 pins on PORTA all set to be outputs.

    EnableInterrupts;

  return;
}


/****************************************************************************
 Function
     STEPS12_RTI_Resp

 Parameters
     None.

 Returns
     None.

 Description
     Interrupt routine that consists of the state machine for stepping the motor
     in the proper order.  Interrupt Routine to handle RTI.  Function will be
     executed by rate defined in STEPS12_Init.

 Notes

****************************************************************************/


void interrupt 7 STEPS12_RTI_Resp(void)
{
  CRGFLG = CRGFLG_RTIF_MASK;   /* clear the source of the int */


   if(Counter>=StepPeriod)
    Counter=1;
   else {
    Counter++;
    return;
   }


   if(StepsLeft<=0) {
    CurrentlyStepping=0;
    return;
   } else
    StepsLeft--;

   if(StepDirection)  {
    if(StepState>=MAX_STEPS)
      StepState=1;
    else
      StepState++;
   } else {
    if(StepState<=1)
      StepState=MAX_STEPS;
    else
      StepState--;
   }

   // Use PortA for motor direction logic
   // PortA0 = Motor 1 enable
   // PortA1 = Motor 1 direction
   // PortA2 = Motor 2 enable
   // PortA3 = Motor 2 direction

   // For Full Step driving both motor outputs always enabled
   // HINT: both motors outputs will not always be on for wave driving or
   //       half step driving!
```

```
   PORTA|=PORTA_BIT0_MASK;
   PORTA|=PORTA_BIT2_MASK;


   switch(StepState) {
    case 1:
      PORTA|=PORTA_BIT1_MASK;
      PORTA|=PORTA_BIT3_MASK;
      break;
    case 2:
        PORTA|=PORTA_BIT1_MASK;
      PORTA&=~PORTA_BIT3_MASK;
      break;
    case 3:
      PORTA&=~PORTA_BIT1_MASK;
      PORTA&=~PORTA_BIT3_MASK;
      break;
    case 4:
      PORTA&=~PORTA_BIT1_MASK;
      PORTA|=PORTA_BIT3_MASK;
      break;
   }

   return;
}

/****************************************************************************
 Function
     STEPS12_Stop

 Parameters
     None.

 Returns
     None.

 Description
     Stops the RTI from advancing the motor.

 Notes

****************************************************************************/


void STEPS12_Stop(void) {
   StepsLeft=0;

   return;
}

#ifdef TEST

void main(void) {

  while(1) {
   while(CurrentlyStepping); // wait until the active steps are done
   STEPS12_Init(100,1,10);
   while(CurrentlyStepping);
   STEPS12_Init(100,0,10);
  }

 return;
}
#endif // TEST
/*------------------------------ End of file ------------------------------*/
```

```
/****************************************************************************
 *
 *  CMPE-118 basic servo driving for Lab 3
 *
 *  History
 *  When           Who     What/Why
 *  -------------- ---     --------
 *  1/30/09        jbb     simplified for lab exercise
 *
 *  Original BotLib created by Scott Early as part of CMPE-118 in Winter 2008
 */

#ifndef SERVO_H
#define SERVO_H

#include <mc9s12c32.h>     /* derivative information */

#define uchar unsigned char

/****************************************************************************
 * General Functions
 */

void Bot_Init(void);                // call before any other library functions


/****************************************************************************
 * R/C Servo Control
 *
 * Actual positional accuracy is about ±5 degrees
 */

void RCServo_Set(char position);
// position: a value in degrees from -60 to +60

#endif
```

```
/****************************************************************************
 *
 *  CMPE-118 basic servo driving for Lab 3
 *
 *  History
 *  When           Who     What/Why
 *  -------------- ---     --------
 *  1/30/09        jbb     simplified for lab exercise
 *
 *  Original BotLib created by Scott Early as part of CMPE-118 in Winter 2008
 */

#include <hidef.h>         /* common defines and macros */
#include <mc9s12c32.h>     /* derivative information */
#include <S12C32bits.h>    /* bit definitions  */

#include <stdio.h>
#include <stdlib.h> /* needed for rand() */
#include <termio.h>
#include <ads12.h>



/****************************************************************************
 * Private General Functions
 */
void Bot_SetPortBit(uchar* port, uchar bit, uchar value);


/****************************************************************************
 * PWM Functions
 *
 * PWM Channels 2 and 3 are used by the R/C servo functions
 */

void Bot_PWMInit(void);
void Bot_PWMSetPins(uchar pins);
char Bot_PWMSetClock(uchar clock, uchar prescale, uchar scale);
char Bot_PWMSetChannel(uchar channel, uchar enable, uchar period, uchar duty,
                       uchar scaler, uchar polarity, uchar cae);
char Bot_PWMSetPeriod(uchar channel, uchar period);
char Bot_PWMSetDuty(uchar channel, uchar duty);

/****************************************************************************
 * R/C Servo Control
 */

#define RCSERVO_PORT_PWM_0  2                    // PWM channel for servo 0
#define RCSERVO_PORT_PWM_1  3                    // PWM channel for servo 1

void RCServo_Init(void);

/****************************************************************************
 ****************************************************************************/
```



```

/****************************************************************************
 * General Functions
 */

void Bot_Init(void) {
    Bot_PWMInit();
    RCServo_Init();

    DDRT = 0xC0;
}

void Bot_SetPortBit(uchar* port, uchar bit, uchar value) {
    if (bit > 7) return;
    bit &= 1;
    *port = (*port & ~(1 << bit)) | (value << bit);
}



/****************************************************************************
 * PWM Functions
 */

void Bot_PWMInit(void) {
    // PWMCTL?
    PWME = 0x00;    // disable all PWM channels

    // Bot_PWMSetClock(clock, prescale, scale)
    Bot_PWMSetClock(0, 4, 10);  // motor drive 295Hz clock (pins 0, 1, 4, 5)

    Bot_PWMSetClock(1, 6, 14);  // RC Servo 52Hz clock (pins 2, 3)
    Bot_PWMSetPins(0x0F);               // enable pins 0, 1, 2, 3
}

void Bot_PWMSetPins(uchar pins) {
    MODRR = pins & 0x3F;   //0011 1111
}

char Bot_PWMSetClock(uchar clock, uchar prescale, uchar scale) {
    if (clock > 1)
        return(!0);

    clock &= 1;
    prescale &= 7;
    if (clock == 0) {
        PWMPRCLK = (PWMPRCLK & ~0x07) | prescale;
        PWMSCLA = scale;
    } else {
        PWMPRCLK = (PWMPRCLK & ~0x70) | prescale << 4;
        PWMSCLB = scale;
    }
    return 0;
```

```c
}

char Bot_PWMSetChannel(uchar channel, uchar enable, uchar period, uchar duty,
                       uchar scaler, uchar polarity, uchar cae)
{
    if (channel > 5) return !0;
    enable &= 1;
    polarity &= 1;
    scaler &= 1;

    // disable channel
    PWME &= ~(1 << channel);

    // set registers
    Bot_PWMSetPeriod(channel, period);
    Bot_PWMSetDuty(channel, duty);
    PWMCLK = (PWMCLK & ~(1 << channel)) | scaler << channel;
    PWMPOL = (PWMPOL & ~(1 << channel)) | polarity << channel;
    PWMCAE = (PWMCAE & ~(1 << channel)) | cae << channel;

    // enable channel
    PWME |= enable << channel;

    return 0;
}

char Bot_PWMSetPeriod(uchar channel, uchar period) {
    if (channel > 5) return !0;
    *(&PWMPER0 + channel) = period;
    return 0;
}

char Bot_PWMSetDuty(uchar channel, uchar duty) {
    if (channel > 5) return !0;
    *(&PWMDTY0 + channel) = duty;
    return 0;
}



/*****************************************************************************
 * R/C Servo Control
 */

/* This function sets the period of a PWM signal to roughly 19ms = 53Hz

   You probably don't need to change this unless you want higher resolution-
   if you've already finished the lab!

 */
void RCServo_Init(void) {
    // Bot_PWMSetChannel(channel, enable, period, duty, clock, polarity, cae)
    Bot_PWMSetChannel(RCSERVO_PORT_PWM_0, 1, 254, 0, 1, 1, 0);
```



```c
    return;
}


/* This function should set the duty cycle of a PWM signal to position an
   R/C servo motor to the function's argument 'position'.

   After RCSerco_Init() has been called to set the period of the signal, a PWM
   signal can be made by varying the duty cycle of this signal.  As provided,
   the period is set to 19ms by setting the PWMPERx register to 254 and
   setting other prescalers.

   The desired duty cycle can be created by setting the PWMDTYx register
   through the Bot_PWMSetDuty() function.  When the PWM signal starts, a
   counter is set to zero and the output goes high; this counter counts
   up at a rate specified by various prescalers.  When this counter value
   matches PWMDTYx, the output goes low.  It remains low until the counter
   matches PWMPERx (as set in RCServo_Init()), when the counter is reset
   causing the output to go high.

   This function is currently limited by the usage of 8 bit counters.  The
   resolution of the duty cycle is limited by the value of PWMPERx: there
   are only 254 gradations to which the duty cycle can be set, even fewer of
   which are useful when driving a servo.

   It is possible to concatenate the 8 bit registers into a 16 bit register
   for increased resolution.  This is left as an excercise for the 118er.
   See the MC9S12C Reference Manual, section 12.4.2.5, for details. .
*/
void RCServo_Set(char position) {
/* Modify this function to set a duty cycle based on the desired position.
     1ms ~= -60 degrees,
   1.5ms ~=   0 degrees,
     2ms ~=  60 degrees
 */

    uchar duty;     // raw PWM duty counter

  // Please, don't push the servos past this.
    if (position < -60 || position > 60) return;

  // Modify this to create a duty cycle based on the desired position.
    duty = 20; // (20/254)*19ms = 1.5ms

  //printf("Position: %03d, Duty: %02d\r\n", position, duty);

    Bot_PWMSetDuty(RCSERVO_PORT_PWM_0, duty);
}

/*------------------------------ Footnotes ------------------------------*/
#ifdef SERVO_TEST

#include <stdio.h>
#include <termio.h>
```

```
#include "servobasic.h"


void main(void) {

  char position = 0;

  TERMIO_Init();

  printf("Starting up...\n\r");

  Bot_Init();
  printf("Botlib initialized...\n\r");

  RCServo_Set(position);

  printf("Servo set to 0 degrees...\n\r");

  /* Changes two degrees with every key hit...
                                            ...once you finish RCServo_Set()
   */
  while(TERMIO_GetChar()) {
    position += 2;

    if ( position > 60 ) {
      position = -60;
    }
    RCServo_Set(position);
    printf("now at %d degrees\n\r", (int)position);
  }

}
#endif
```