**EET 240 Microcontroller I**

**Instructor Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Lab: Stepper Motor - Part I**

**Introduction:**

The stepper motor lab is actually a two part lab (Lab 8 and Lab 9) which will allow you to learn how a stepper motor works, the specific code (Gray's Code) that it takes to energize/de-energize the coils to induce/collapse a magnetic field of current that creates the necessary polarity change for the motor to step. We will then revisit the delay loop (nested loop) that allows the stepper motor to operate for so many pulses or steps one way and then back another way before starting again.

There are two basic types of stepper motors: Unipolar and Bipolar. The unipolar stepper motor has two windings per phase, one for each direction of magnetic field. Since in this arrangement a magnetic pole can be reversed without switching the direction of current, the commutation circuit can be made very simple (e.g., a single transistor) for each winding. Typically, given a phase, one end of each winding is made common: giving three leads per phase and six leads for a typical two phase motor. Often, these two phase commons are internally joined, so the motor has only five leads. The one we will be using has six.





The unipolar stepper motor does not require the H-bridge or stepper motor driver IC (L293D which as 2 H-bridge built-in). Either motor requires more current then the MCU can provide so the Darlington array will be used to provide that in addition to isolating the microcontroller from the stepper motor in case of a short in the circuit happens. The Darling array is the NTE2018 IC which has built-in shunt diodes to protect for peak-inverse current changes when the magnetic field collapses on the coil. Just like in the DC motor lab we do not want to have current back feeding towards the MCU.

There three "pulse patterns" that can be used to make a stepper motor move. The motor can operate in full-step, half-step, and wave-step modes. The actual degree of step is configured physically by the stepper motor manufacturer. So for example the unipolar we will be using is a 7.5⁰/step motor. We can reduce that in half to 3.75⁰ by operating it in half-step mode -- if we need something finer than that in accuracy then a different motor must be chosen. The wave-step mode will turn on only one coil per pulse which can speed up how fast the motor rotates per revolution but you sacrifice torch. So if you have a heavy mechanical structure that you need the torch to move around when the motor steps the wave-step mode is not an option.

The four unique codes for full-step mode (Gray's code) is 0x0C, 0x09, 0x03, and 0x06 for the motor. When the code is sequenced in one direction the motor will step in the clockwise (CW) mode and when it is sequenced in the opposite direction the motor will step in the counterclockwise (CCW) direction. Because the coils take time to collapse and build the magnetic field in the coils a delay loop is required between each step. We will determine how fast we can pulse the motor without it twitching itself to death in this lab. In connecting the motor up if the wires on not in the correct order the motor will start to step one direction and then seem to go back in the other direction. Take time to build the interface circuit so that the motor will operate correctly. Have an instructor show you how to verify how to test a stepper motor out of circuit if you do not have documentation for one.

We will use the logic analyzer to verify the code going out to the stepper motor and the direction that the motor is operating based on the sequence of the Gray code observed. We will also verify the voltages and currents between the MCU and stepper motor circuit through the Darlington array motor driver circuit. **Do not disconnect the output circuit it will be used for the next part of the lab.**

After working with the stepper motor circuit we want to control the operation of it with a sensor input. The AD820 is a rail-to-rail op-amp is single supply (+5.0V and Ground) which makes it great for MCU projects -- no negative voltages here. The circuit is a comparator that will use a reference voltage of +2.5V. The other input on the op-amp will contain the light dependent resistor (LDR) which can swing above 2.5V and below 2.5V as the LDR is covered (darkness) or is left in light (brightness). The output of the op-amp will be connected to an input port and the program will check to see which logic level the output represents (i.e., +2.5V or above is a logic 1; 0.0V to 0.7V is a logic 0). Based on the logic level the stepper motor will turn clockwise or counterclockwise depending on how the program is written to "sense" the logic level.

The final procedure in this lab is to take a look at using the interrupt subsystem with the input sensor to control the direction of the stepper motor. Instead of polling the input pin to see if the sensor has changed states to change direction on the motor we will enable the interrupt subsystem to sense when the input changes and run the ISR or interrupt service routine.

Interrupts are just what you think about when you are doing something and someone or something interrupts you from the task at hand. An interrupt is an asynchronous signal (i.e., similar to the PRE’ or CLR’ lines on the flip-flops that forces Q output to either be a logic 0 or 1) that requires the program to change the sequence of execution of programming code. Plus, there is a propagation delay time for recognizing the interrupt which is called interrupt latency and that information is in the full datasheet of the microcontroller. The AVR supports a variety of interrupts:



The table includes information about the type of interrupt, the priority of the interrupt, and a brief description of each interrupt. It will be important to place your “jump to interrupt requests” in the order of the vector table address, otherwise you will get an overlap error on addressing.

Interrupts are used instead of a technique called polling which waste time waiting for an external event to happen (i.e., SBIC PinD, 0; is PD0 == 0?). This command constantly loops back to check to see if the PD0 line has gone low before completing the sequence of instructions or instruction set if it is within a multi-step process. Interrupts allow us to multitask, work in real-time, and deal with situations based on a condition. The act of interrupting is called an interrupt request or IRQ (based on the x 86 PC systems). In terms of the AVR it is called an interrupt service routine or ISR.

Hardware interrupts were introduced as a way to avoid wasting the processor's valuable time in a polling loop that is used with input/output operations continue to check and wait and check again until the condition is true and/or some cases false based on what the program is accomplishing. Hardware interrupts can be implemented in hardware as a distinct system with control lines, or they may be integrated into the memory subsystem of the architecture of the embedded device. You need to be careful about the code you place into an ISR because if you place too much other ISR routines will not be serviced timely and the system will falter or fail. It may be a decision such as placing the “counter for a particular amount of time” in the interrupt service routine but the rest of the main code about conversion to a number system and displaying the results out to the LCD for example.

An interrupt service routine does not pass a parameter nor does it return anything. Once the hardware interrupt line is driven to the active state it will complete the task and then return to where the PC last left off; no more or less.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt**. Such an interrupt has four properties:

1. The Program Counter (PC) is saved in a known place (the stack).
2. All instructions before the one pointed to by the PC have fully executed.
3. No instruction beyond the one pointed to by the PC has been executed (that is no prohibition on instruction beyond that in PC, it is just that any changes they make to registers or memory must be undone before the interrupt happens).
4. The execution state of the instruction pointed to by the PC is known.

An interrupt that does not meet these requirements is called an **imprecise interrupt**.

The phenomenon where the overall system performance is severely hindered by excessive amounts of processing time spent handling interrupts is called an **interrupt storm**. An interrupt storm will cause low perceived system responsiveness, or even appear to be a complete system freeze. This state is commonly known as live-lock and opening the task manager within the operating system to see which processes are conflicting or turning off tasks will “unfreeze’ the computer. In such a state, the system is spending so much time processing interrupts that it is not completing any other work. Therefore, it does not appear to be processing anything at all, because of a lack of output to the user, the network, or otherwise. An interrupt storm is sometimes mistaken for **thrashing.** This situation is where large amounts of computer resources are used to do a minimal amount of work. The system is in resource contention (bus contention) and using the task manager to shut off unneeded tasks may resolve the issue; thrashing is typically self-sustaining until something occurs to remove the original situation that led to the initial thrashing behavior.

Interrupt storms can have many different causes, including misconfigured or faulty hardware devices, faulty device drivers, or flaws in the operating system. Most modern hardware implement methods for reducing or eliminating the possibility of an interrupt storm. For example, many Ethernet controllers implement interrupt "rate limiting", which causes the controller to wait a programmable minimum amount of time between each interrupt it generates.

Methods of “Sensing” an Interrupt:

A **level-triggered interrupt** is a class of interrupts where the presence of an un-serviced interrupt is indicated by a high level (1), or low level (0), of the interrupt service routine and associated interrupt request line. A device wishing to signal an interrupt drives line to its active level, and then holds it at that level until serviced. It ceases asserting the line when the CPU commands it to or otherwise handles the condition that caused it to signal the interrupt.

Multiple devices may share a level-triggered interrupt line if they are designed to. The interrupt line must have a pull-down or pull-up resistor so that when not actively driven it settles to its inactive state. Devices actively assert the line to indicate an outstanding interrupt but let the line float (do not actively drive it) when not signaling an interrupt. The line is then in its asserted state when any (one or more than one) of the sharing devices is signaling an outstanding interrupt. Floating a line is not a good design practice, because with CMOS devices the “floated” input will toggle back and forth trying to sense the logic level on the line. In addition, in this “floated” condition the MOS device is more susceptible to ESD.

There are also serious problems with sharing level-triggered interrupts. If any device on the line has an outstanding request for service the line remains asserted, so it is not possible to detect a change in the status of any other device. Deferring servicing a low-priority device is not an option, because this would prevent detection of service requests from higher-priority devices. If there is a device on the line that the CPU does not know how to service, then any interrupt from that device permanently blocks all interrupts from the other devices.

An **edge-triggered interrupt** is a class of interrupts that are signaled by a level transition on the interrupt line, either a falling edge or negative-going transition edge (1 to 0) or a positive-edge going transition edge or rising edge (0 to 1). Basically, this signal is used to “pulse the line” through a transition state and release it back to its normally low or high state called the quiescent state. The “pulse” needs to be long enough for detection otherwise a software delay or hardware interface circuit such as a latch or one-shot circuit may be required to detect the change in transition for the interrupt to be serviced.

Multiple devices may share an edge-triggered interrupt line if they are designed to. The interrupt line must have a pull-down or pull-up resistor so that when not actively driven it settles to one particular state. Devices signal an interrupt by briefly driving the line to its non-default state and let the line float (do not actively drive it) when not signaling an interrupt. This type of connection is also referred to as open collector. The line then carries all the pulses generated by all the devices. (This is analogous to the pull cord on some buses and trolleys that any passenger can pull to signal the driver that they are requesting a stop.) However, interrupt pulses from different devices may merge if they occur close in time. To avoid losing interrupts the CPU must trigger on the trailing edge of the pulse (e.g. the rising edge if the line is pulled up and driven low). After detecting an interrupt, the CPU must check all the devices for service requirements.

Edge-triggered interrupts do not suffer the problems that level-triggered interrupts have with sharing. Service of a low-priority device can be postponed arbitrarily, and interrupts will continue to be received from the high-priority devices that are being serviced. If there is a device that the CPU does not know how to service, it may cause a spurious interrupt, or even periodic spurious interrupts, but it does not interfere with the interrupt signaling of the other devices. However, it is fairly easy for an edge triggered interrupt to be missed - for example if interrupts have to be masked for a period - and unless there is some type of hardware latch that records the event it is impossible to recover. Such problems caused many "lockups" in early computer hardware because the processor did not know it was expected to do something. More modern hardware often has one or more interrupt status registers that latch the interrupt requests; well written edge-driven interrupt software often checks such registers to ensure events are not missed.

Some systems use a hybrid of level-triggered and edge-triggered signaling. The hardware not only looks for an edge, but it also verifies that the interrupt signal stays active for a certain time. A common use of a hybrid interrupt is for the NMI (non-maskable interrupt) input. Because NMIs generally signal major – or even catastrophic – system events, a good implementation of this signal tries to ensure that the interrupt is valid by verifying that it remains active for a period of time. This 2-step approach helps to eliminate false interrupts from affecting the system. A **non-maskable interrupt** (**NMI**) is an interrupt on microprocessors and microcontrollers that cannot be ignored by standard interrupt masking techniques in the system. It is typically used to signal attention for non-recoverable hardware errors. (Some NMIs may be masked, but only by using proprietary methods specific to the particular NMI.)

The other typical uses for interrupts seen in the table include system timers, analog-to-digital (ADC), serial peripheral interface (SPI), universal asynchronous receive transmit (USART), etc.

The Configuration of External Interrupts:

1. Setup the program with upon reset the program jumps from location 0x0000 to the main part of the program.
2. Setup the vector table jump statements (i.e., .org 0x02 jmp exo\_isr)
3. In the main code setup the stack, the interrupt, the ports, and whatever the “default” main program is doing prior to the interrupt service routine being called… maybe the display is showing a number or the stepper motor is off until an interrupt is called.



The value for INT1 = $80; for INT0 = $40; and for INT2 = $20



|  |
| --- |
|  |
|  |

The lower nibble of the MCUCR (MCU control register) is used to configure the sense control for each external interrupt line.

The MCUCR register will be used to set how the interrupt will be triggered; level, edged, or any type of transition. Note: INT2 can only sense either a falling or rising edge and is configured in the MCUCSR.



Refer to the datasheet for propagation delay times for the external interrupt lines.

1. At the end of the main code section make sure to enable the global interrupt (SEI) bit in the SREG
2. Write the external interrupt service routine code.
3. At the end of the external interrupt service routine include the RETI command so that after the ISR is completed the “main” program will begin to execute.
4. If the “flag-bits” are important or any data in the registers needs to be saved push it onto the stack prior to starting the ISR. This is not automatically done with the AVR architecture. Only the “old” program counter is saved to the stack.

**Outcomes:**

1. Develop the IPO Chart, Pseudo-Code or Flowchart for the stepper motor lab.
2. Write instructions and thorough descriptive comments explaining the process for stepping a uni-polar stepper motor around in clockwise (CW) and counterclockwise (CCW) directions.
3. Write various delay loop instructions to change the amount of time per step to vary the speed of the motor.
4. Build an interface circuit (output) using the Darlington 8-channel array IC to drive the unipolar stepper circuit with LEDs for display effect.
5. Build an interface circuit (input) using either a rail-to-rail circuit (AD820N preferred) or a 741 op-amp circuit. Note: isolate the voltages from the op-amp circuit from the microcontroller and connect all grounds together.
6. Configure the Interrupt System to work with the AD820N interface circuit to change directions.
7. Test and troubleshoot the interface circuit using the DMM, oscilloscope, logic analyzer, and ISP Debugger Emulator to verify correct operation. Document the results in a report.

**Equipment and Parts:**

Microchip Studio (AVRStudio) software

1 - Development Board with Cables.

User Guide for STK200/STK500

1 - Unipolar Stepper Motor

Resistors: (1 - 1.0kΩ)

1 - NTE2018 (equiv. to ULN2003 IC) Darlington Array (8 channels)

1 – BAR LED for light effects (can use built-in BAR LED on STK200 board).

1 - AD820N rail-to-rail single supply op-amp

1 - Logic Analyzer 8 channel

1 – MSO Oscilloscope (4 Channel Digital Scope) with Digital POD and 10 x probe

1- mkll-ICE Debugger/Emulator

2 - DMMs (one is sufficient you will just need to move it back and forth between input and output legs of the LM7805 power supply circuit).

1 - 7805 +5.0V Linear Voltage Regulator

2 - Capacitors (0.1uF)

1 - LED with series limiting resistor (150.0Ω) to verify the power supply is turned on.

1 - 9.0V battery

**Procedures:**

**Part I - Let's make the interface circuit portable**

|  |  |
| --- | --- |
|  | The LMxx family is a set of voltage regulators for positive voltage outputs. The 7805 produces a  + 5.0V output with a maximum current rating of 1.5A. Conversely, the LM79xx family is a set of voltage regulators for negative voltage outputs. |

**Schematic 1: LM7805 Power Supply Circuit**

1. Build the circuit above and use a +9.0V source from a DC power supply for the next procedures before verifying operation with a battery when making your circuit portable for the stepper motor.

2. Adjust the input voltage from the DC power supply to 30.0V and measure the output voltage:

Vout = 5.23V @ Vin = 30.0V

What is the maximum Vin value from the datasheet? 35V

3. Studying the regulator dropout characteristics. Replace the LED and 150.0Ω resistor with a 1.0kΩ resistor and determine the approximate dropout voltage of the LM7805 voltage regulator.

|  |  |  |
| --- | --- | --- |
| **Vin** | **Vin (Measured)** | **Vout (Measured)** |
| **9.0V** | **9.06** | **5.14** |
| **8.0V** | **7.99** | **5.14** |
| **7.5V** | **7.46** | **5.14** |
| **7.0V** | **6.99** | **5.13** |
| **6.5V** | **6.48** | **5.10** |
| **6.0V** | **6.03** | **4.67** |
| **5.5V** | **5.55** | **4.21** |
| **5.0V** | **5.05** | **3.74** |
| **4.5V** | **4.53** | **3.24** |

What is the approximate dropout voltage? Vdrop = 6.5V

4. Replace the 1.0kΩ resistor with the LED and series limiting resistor on the output. Disconnect the DC power supply and use a 9.0V battery to connect to the stepper motor and the NTE2018 Darlington Array IC interface circuit. Optional is to place a switch in the positive leg of the DC battery leg to turn the "portable power supply" off when not being used.

**Procedures:**

**Part II - Getting the Stepper Motor to turn one way**

1. Develop the IPO Chart for the stepper motor lab.
2. Develop a flowchart or pseudo-code for the problem statement: Problem - move the stepper motor in single steps in one direction with a delay loop that you determine the amount of time.
3. Calculate the number of steps required for one full revolution: please show your math.

7.5x = 360º = 48

1. Write the assembly language program from your pseudo-code and IPO Chart that will be used with the unipolar stepper motor interface circuit.
2. Build the output interface circuit using the Darlington array 8-channnel IC to drive the unipolar stepper motor. Note: a separate power supply is required for this interface circuit. The stepper motor does not have a ground connection but a "Vcc" of +5 volts to the diode/common cathode pin must be completed. The diode/common cathode pin represents internal shunt diodes that will be placed across the coils of the stepper motor to protect the IC and MCU from reverse current flow. The IC does have a ground and all grounds need to be connected together. You may choose to use the BAR LEDs, the STK200/STK-500 LEDS (remember they are reverse logic), or build your own LEDs circuits.



**Schematic 2: Unipolar Stepper Motor Circuit**

Notes on schematic: Placing a rectangle graphic for the STK200/STK-500 and NTE2018 IC was used to draw schematic and then junction nodes (CTRL-J) to draw lines to make connections were done. EagleCAD software does have Atmega devices in the tool bin. It does take some time to learn the software. **Using the new unipolar motor has a different color sequence on the wires.**

1. Build and run your program verify operation before uploading to the target device.
2. Upload the executable code to the target device and observe if the LEDs are operating correctly. Connect the Logic Analyzer up to verify that you see the correct code in the sequence.
3. Connect up the stepper motor interface circuit. Remember to power down your board when making the connections [the STK200/STK500 is not hot-swappable ;)].
4. Using the logic analyzer determine which direction the motor is operating? CW

What is the pulse code sequence? 1100, 0110, 0011, 1001

1. Determine the current and voltage input and output coming from one channel of the Darlington array IC.

Input: 3.0V, 0.93mA Output: 6.5V

**Procedures:**

**Part III - Getting the Stepper Motor to Turn the Other Way**

1. Copy and paste the first code into a new project. Edit this code to run the motor in the opposite direction.
2. Repeat steps 1-9. What is the motor's direction? CCW

What is the pulse code sequence? 1001, 0011, 0110, 1100

**Procedures:**

**Part IV - Counting Loops with the Stepper Motor**

1. Create a program that will allow the motor to turn 180⁰ CW and 720⁰ CCW. Use the handouts in the online environment to assist you with this part.
2. Calculate your delay loop time between each step.

**Procedures:**

**Part V - Using a Sensor to Change Directions based on "sense". Think of a surveillance camera.**

1. Develop a flowchart or pseudo-code for the problem statement. Problem: The stepper motor will single step in one direction at one rate of speed and then change directions to a different rate of speed in the opposite direction based on the change of input signal coming from the input op-amp circuit output. Note: think about the check key-press command we used when sensing an input from the buttons on the STK200/STK-500. Hint! Consider the key-press program and how you would use this "poll" the input PD0 to check for the logic-level on the output of the op-amp.
2. Write the assembly language program from your flowchart or pseudo-code that will be used with the input interface circuit to operate the unipolar stepper motor interface circuit.
3. Leave the output interface circuit built with the Darlington array 8-channnel IC to drive the unipolar stepper motor.
4. Calculate the voltage divider required to provide a logic 1 and logic 0 when the light dependent resistor (LDR) is either fully covered or has full light on it from Schematic 1.

What is the resistance of the LDR when fully covered 26.4kΩ versus with full light 3.85kΩ

What is the TBD resistor value for an operational circuit? 5.6kΩ

1. Predict the output voltages for when the LDR is covered 6V versus when the LDR is left with full light on it 200mV.
2. Measure the output of the circuit when the LDR is covered 7.45V versus when the LDR is left with full light on it 195mV.
3. What happens to the output of the circuit when you drop the Vin to the Vdrop voltage for the LM7805 voltage regulator? Explain. You will need to use the DC variable power supply for this operation.
   1. The motor never ends up changing states because with so little supplied the output never spikes high enough.
4. Check with the logic analyzer and MSO Digital Scope to see that the input "sensor" interface circuit is working and controlling the direction of the stepper motor. Document your observations of how you can tell.



**Schematic 3: Input Sensor Circuit for Stepper Motor Interface**

1. Switch the positions of R4 (LDR) and R3 (TBD) resistors. What impact does this have on the operation of the interface circuit with the program? Explain. Change the configuration back to the original setting before answering the next question.
   1. It makes the device work backwards since it is now reading high when in full like and low when covered.
2. Switch the two voltage dividers with one another by changing VREF to the non-inverting input on the op-amp and place the "sensor" leg on the inverting input. What impact does this have on the operation of the interface circuit with the program?
   1. The circuit is working the opposite way.

**Procedures:**

**Part VI – Connecting the “Sensor” Input to the Interrupt Subsystem**

1. Type the code from the example in this lab into Microchip Studio (AVRStudio). Develop your own comments so that you understand what is going on with the program. Build and upload the executable file into the target device (Atmega32).
2. Connect the stepper motor interface circuit with the rail-to-rail op-amp circuit with LDR).
3. Calculate the two delay loops for time and determine based on a 1 MHz clock signal from the Atmega32 internal oscillator register being used to generate it what the frequency is for each delay loop.
4. Connect the oscilloscope to one of the PortB pins and measure the time and calculate the measured frequency for the shorter of the two delay loops. Based on this measurement is the interrupt service routine working as anticipated? Explain.
5. Connect the Atmel mkll-ICE Debugger/Emulator to the JTAGEN port on the STK200/STK500 connector and enable the simulator within Microchip Studio to verify the operation of the code as the interrupt service routine is working. Connect the oscilloscope to the external interrupt pin and verify that the PD2 input transitions between low and high voltages to be the correct logic levels and the digital pod is connected to the output pins the operate the stepper motor.
6. Document what happens with the interrupt happens and the resulting output changes.

**Procedures:**

**Part VII – Changing the Interrupt Vector Address**

1. In Microchip Studio (AVRStudio) swap the .org 0 and .org 0x2 statements with their jump statements so that the .org 0x2 statement is above the .org 0 statement. Rebuild the program. What happened? Explain why this happened?
   1. We get an error because we are trying to have an interrupt that comes before the start of the program.

Interrupt Program Code

;Programmer: A. Trausch

; Rogue Community College -- Electronics Department

; Date: 8March2021

; File Name: interrupt4.asm

; Version: 6

; Location: I:\Electronics\EET241ANGEL\exampleprograms241

;Abstract: Demonstration program for falling edge-triggered interrupt on INT0 (PD2).

; Runs the stepper motor clockwise until low-level logic on PD2 is sensed then motor

; runs counter clockwise at a different speed.

; INT0 and INT1 can be configured for level and edge-triggered interrupts

; In the GICR register (global interrupt control register) the three most

; significant bits are (INT1 INT0 INT2) using the value $40 will

; enable INT0 external interrupt pin. All others are disabled with logic 0

; INT2 external pin can only be edge-triggered. To configure level or edge triggered

; the MCRCR (MCU control register) is used. The bits are ISC01 and ISC00 with a value

; of $2 for falling edge reaction. See page 378 in textbook.

; .

.include "m32def.inc"

.org 0

jmp reset ; reset button and main program starts at address 0x00

.org 0x02

jmp ex0\_isr ; interrupt vector table address for INT0

reset:

.def step1 = r0 ; user defined labels for stepper motor code at full-step

.def step2 = r1

.def step3 = r2

.def step4 = r3

.def temp = r16 ; user defined labels for working registers r16 - r18

.def dly1= r17

.def dly2= r18

ldi temp, high(ramend) ; copies high byte address of ramend into temp

out sph, temp ; copies high byte address of ramend into stack pointer

; register high byte.

ldi temp, low(ramend) ; copies low byte address of ramend into temp

out spl, temp ; copies low byte address of ramend into stack pointer

; register low byte

;top of stack at address 0x085f

ldi temp, 0x00 ; load immediately $00 into r20

out mcucr, temp ; use $00 to configure mcucr for INT0 -- low level

; logic .

ser temp ; set all bits high in r16 register

out ddrb, temp ; set bits PB0 - PB7 as output

; using PB0 - PB3 for stepper motor

sbi portd, 2 ; pull-up resistor active on PD2 (INT0)

; by default on reset Ports are input.

ldi temp, 1<<int0 ; shift 1 into int0 place -->$40

out gicr, temp ; configure INT0 bit within gicr I/O reg.

; for using INT0 interrupt

ldi temp, 12 ; using r16 to copy full-step motor code into

mov step1,temp ; r0, r1, r2, and r3 successively

ldi temp, 6

mov step2,temp

ldi temp, 3

mov step3, temp

ldi temp, 9

mov step4, temp

sei ; set the global interrupt bit within SREG

cw: ; "main" program running prior to interrupt request

out portb, step1 ; sending out full-step code to portb and then calling

rcall delay1 ; delay1 (256 x 256) direction clockwise

out portb, step2

rcall delay1

out portb, step3

rcall delay1

out portb, step4

rcall delay1

rjmp cw

delay1:

dec dly1 ; dly1 initial value = 0 then decrements to 255

brne delay1 ; (is z bit == 0)? no back to decrement dly1

; (is z bit == 0)? yes fall through and decrement dly2

dec dly2

brne delay1 ; (both dly1 and dly2 == 0)? no back to delay1 and do it

; again. yes ---> go get next step sequence

ret

ex0\_isr: ; when INT0 (PD2) == 0 then execute

; ccw code at a faster rate.

; when INT0 (PD2) does not equal 0 return

; from interrupt and continue cw direction code.

out portb, step4 ; sending out full-step code to portb and then calling

rcall delay2 ; delay2 (255 x 5) direction counterclockwise

out portb, step3

rcall delay2

out portb, step2

rcall delay2

out portb, step1

rcall delay2

reti ; after service routine completed return to main code."cw"

delay2:

ldi dly1, 255

ldi dly2, 5

wait2a:

dec dly1

brne wait2a

wait2b:

dec dly2

brne wait2b

ret

;code uses concepts from textbook related to example text 10-5 pg377

**Extra Credit:**

1. Add another interrupt service routine for INT1 with a different interrupt service routine condition.
2. Create a different interrupt application such as an ISR to count for 1 second and then go back to the main code to convert the number to be displayed out to PortB for a timer or clock. Add a Binary to BCD for 7-segment display or Binary to ASCII for LCD display.

Note: Please make sure that the lab is completed with full documentation before working on the extra credit.

Be able to respond to these questions for the final exam.

1. If a motor takes 90 steps to make one complete revolution, what is the step angle for this motor?

a. 360/90 = 4 º

2. Calculate the number of steps per revolution for a step angle of 1.8⁰?

a. 360/1.8 = 200

3. Finish the normal 4-step sequence clockwise if the first step is 0011b?

a. 1001

b. 1100

c. 0110

4. Finish the normal 4-step sequence clockwise if the first step is 1100b?

a. 0110

b. 0011

c. 1001

5. Finish the normal 4-step sequence counterclockwise if the first step is 1001b?

a. 0011

b. 0110

c. 1100

6. What is the purpose of the ULN2003 or NTE2018 placed between the AVR and the stepper motor? What does the diode/common cathode pin when it is connected correctly do? Can this IC be used for 3.0A motors?

a. It allows interfacing between low level digital logic circuitry and high powered peripheral loads.

b. Keeps current from coming back through into the microcontroller.

c. Yes it can be used up to a 4A according to the datasheet

7. What is the effect of a time delay between issuing each step? How do you make the motor step faster?

a.

8. When an RCALL is executed, how many locations of the stack are used? How many locations with a CALL command?

9. Describe the action associated with the RET instruction. Elaborate, provide details, etc.

10. In the AVR, which address is pushed into the stack when a call instruction is executed. What is the order on the stack?

**Documentation to Turn-In with Formal Report:**

1. Individualized IPO chart and pseudo-code for programs.
2. Programs with individualized comments.
3. Typed up questions and calculated work for delay loops to this lab.
4. Zip up work in a folder called StepperMotor\_yourname and attach it to the submission box for this lab activity.
5. Demonstrate your project to an instructor for sign off for Stepper Motor1. If completed remotely submit a video or schedule a Zoom session with the instructor.