ECE 375 LAB 6

External Interrupts

Lab Time: Thursday 1000-1200

Eric Prather

PRELAB

1. In computing, there are traditionally two ways for a microprocessor to listen to other devices and communicate: polling and interrupts. Give a concise overview/description of each method, and give a few examples of situations where you would want to choose one method over the other.

Polling is a deliberate *check* by a device to check a data value, which can then be interpreted by the device and result in a branching path of behaviors. Thus, for a specific behavioral branch to occur, the device must poll something like a peripheral. Hypothetically, if the polling interval was wider than the frame in which the data changes, some data fluctuations may then be missed. If timing is of essence and the polling interval is wide, this is also probable. One solution to this problem would be for the peripheral to store a set of data, rather than the instananious value, which the poller can review upon data retrieval so as not to miss any queues. Polling is useful in situations where immediate reations are important and data does not need to be missed. For example, an automatic door could poll its motion sensor every second, as people approaching the door who need to be let in will not likely move in instantaneous chunks and freeze during the polling times.

Interrupts are an action done *to* the microprocessor by an interrupt signal sent by another device. This results in a literal, physical modification to the interpretation of the program controller operates, causing it to disregard its regular interpretation of instructions in its program to execute a certain hardwired behavior corresponding to the interrupt vector's design. This is instantaneous. Interrupts are useful when the priority of the interrupting activity is high. For example, this may be useful to prevent a failure of a high-precision sub-millimeter CNC machine which must respond instantly and appropriately on a scale of microseconds- or even seconds- due to the sensitive nature of the microscopic physical world.

2. Describe the function of each bit in the following ATmega128 I/O registers: EICRA, EICRB, and EIMSK. Do not just give a brief summary of these registers; give specific details for each bit of each register, such as its possible values and what function or setting results from each of those values. Also, do not just directly paste your answer from the datasheet, but instead try to describe these details in your own words.

EICRA & EICRB: External interrupt control register A and B. Setting the pertinent bits, as defined by the datasheet, will allow for various interrupt handling behaviors to occur. There are four settings supported:

Falling edge interrupts- interrupts occur on the falling edge of the pertinent signal (after it was high)

rising edge interrupts- interrupts occur on the rising edge of the pertinent signal (after it was low)

low level- interrupts occur perpetually while vector is low

all interrupts- any changes in logic (i.e. falling or rising edges) will trigger an interrupt.

Each of four bits of these registers will control one setting to be on or off. The bit being actively considered is determined by EIMSK.

EIMSK: External interrupt mask. Allows the processor to selectively override interrupt controllers from the four cached settings, described above. Thus, only one of the four bit values defined above is pertinent to the board at a time, as EIMSK only has two bits with which to define which behavior Boolean to target.

3. The ATmega128 microcontroller uses interrupt vectors to execute particular instructions when an interrupt occurs. What is an interrupt vector? List the interrupt vector (address) for each of the following ATmega128 interrupts: Timer/Counter0 Overflow, External Interrupt 5, and Analog Comparator.

An **interrupt vector** is a means by which a signal generated by an external device can cause a change in behavior of the processor. The ATMEGA supports a variety of different input vectors, such as those built into the buttons on the board or those which can be defined manually.

Here are the requested addresses:

Timer/Counter0 Overflow: PG4 (port G pin 4) and PG3 (port g pin 3) both are used for this. The latter is simply the inverted of the former.

External Interrupt 5: PE5 (port e pin 5)

and Analog Comparator: ACI (Analog comparator interrupt, part of the analog comparator's dedicated register)

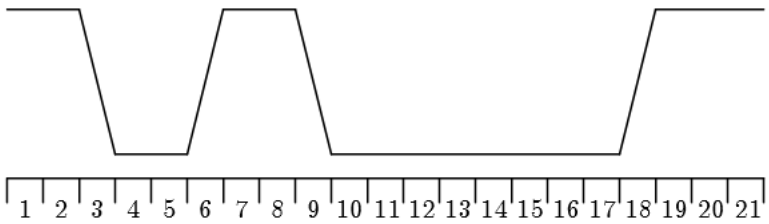

Figure 1: Sample Input to External Interrupt

4. Microcontrollers often provide several different ways of configuring interrupt triggering, such as level detection and edge detection. Suppose the signal shown in Figure 1 was connected to a microcontroller pin that was configured as an input and had the ability to trigger an interrupt based on certain signal conditions. List the cycles (or range of cycles) for which an external interrupt would be triggered if that pin's sense control was configured for: (a) rising edge detection, (b) falling edge detection, (c) low level detection, and (d) high level detection. Note: There should be no overlap in your answers, i.e., only one type of interrupt condition can be detected during a given cycle.

a. Rising edges: 6 and 18

b. Falling edges: 3 and 9

c. Low level detection: 4, 5, and 10-17

d. High level detection: 1, 2, 7, 8, 19, 20, and 21

INTRODUCTION

The behavior executed by this lab is extraordinarily simple- simply to ~~count the number of presses of the left or right "bumper" of the tekbot~~ replicate the bump-bot behavior from Lab1. The important element of it is **how this is achieved**- instead of using the *polling* strategy of responding to peripherals, we are instead using the **interrupt**

strategy. Specifically, we will be using the built-in capacitive button interrupts to trigger the count, rather than a cycle-based polling approach.

PROGRAM OVERVIEW

This program is very simple. Its initialization routine performs some I/O setup and the initialization of the stack pointer. The main new function introduced is the setup of the interrupt control, mask, and flag registers so that interrupts can be correctly received.

Because of the instantaneous nature of interrupts, there is no need to worry that both bumper interrupts will occur at the same time; one will always take precedence over the other.

INITIALIZATION ROUTINE

This performs some I/O setup and the initialization of the stack pointer. The main new function introduced is the setup of the interrupt control, mask, and flag registers so that interrupts can be correctly received.

MAIN ROUTINE

Loops infinitely, moving forward, otherwise doing nothing.

CLEAR WHISKER HIT HISTORY

Clears the history recorded by the bumpbot of the turns taken. This is called when a 180 degree turn happens, because after this nothing is known about the bot's new environment

LEFT SHIFT WHISKER HISTORY

Used when the next turn would ordinarily overflow the memory space allocated for recalling the direction the bump bot was going. Other than that, does what it sends on the tin.

LEFTBUMPERINTERRUPT

If the last item in the bumper memory array is the same as this item's flag ('L'):

Sets bumper memory to ['L', ' ', ' ', ' '] SuperTurnRight.

Otherwise, if the bumper memory array is [L,R,L,R]:

Clears the bumper memory array and performs Turn180.

Otherwise:

Adds "L" to the bumper memory array. Then performs HitRight

Note: While this routine is running, external interrupts are temporarily disabled.

RIGHTBUMPERINTERRUPT

Same as left bumper interrupt routine, just switch "left and right". Because of the instantaneous nature of interrupts, there is no need to worry that both bumper interrupts will occur at the same time; one will always take precedence over the other.

Note: While this routine is running, external interrupts are temporarily disabled.

GETUNSTUCK (TURN180)

Called when alternating whiskers are hit five times in a row. Turns 180 degrees- or in other words, turns for ____ ms.

SUPERTURNLEFT

Same as HitRight, but turns for twice as long before finishing.

SUPERTURNRIGHT

Same as HitLeft, but turns for twice as long before finishing.

LAB1 ROUTINES

HITRIGHT ROUTINE

The HitRight routine is copied directly from Lab 1. It executes as follows:

The HitRight routine first moves the TekBot backwards for roughly 1 second by first sending the Move Backwards command to PORTB followed by a call to the Wait routine. Upon returning from the Wait routine, the Turn Left command is sent to PORTB to get the TekBot to turn left and then another call to the Wait routine to have the TekBot turn left for roughly another second. Finally, the HitRight Routine sends a Move Forward command to PORTB to get the TekBot moving forward and then returns from the routine.

HITLEFT ROUTINE

The HitLeft routine is copied directly from Lab 1. It executes as follows:

The HitLeft routine is identical to the HitRight routine, except that a Turn Right command is sent to PORTB instead. This then fills the requirement for the basic BumpBot behavior.

WAIT ROUTINE

The wait routine is copied directly from Lab 1. It executes as follows:

The Wait routine requires a single argument provided in the *waitcnt* register. A triple-nested loop will provide busy cycles as such that $16 + 159975 \cdot waitcnt$ cycles will be executed, or roughly $waitcnt \cdot 10\text{ms}$. In order to use this routine, first the *waitcnt* register must be loaded with the number of 10ms intervals, i.e. for one second, the *waitcnt* must contain a value of 100. Then a call to the routine will perform the precision wait cycle.

ADDITIONAL QUESTIONS

As this lab, Lab 1, and Lab 2 have demonstrated, there are always multiple ways to accomplish the same task when programming (this is especially true for assembly programming). As an engineer, you will need to be able to justify your design choices. You have now seen the BumpBot behavior implemented using two different programming

languages (AVR assembly and C), and also using two different methods of receiving external input (polling and interrupts). Explain the benefits and costs of each of these approaches. Some important areas of interest include, but are not limited to: efficiency, speed, cost of context switching, programming time, understandability, etc.

The most appropriate approach to programming the ATMega128 board depends more on the emergent system properties the final product and development process must exhibit rather than on the functional requirements of the system itself. Developing in C, a higher-level programming language, incurs a considerably lower development cost and arguably results in more legible, more maintainable code. Critics of the C programming language will point out that undefined behaviors and poor safety and security standards make C a poor option, but in general the concept of using a higher level programming language to provide easier and more legible code remains. Furthermore, low-skill programmers may write *worse* assembly implementations than those the C compiler would generate. Thus, for novices, there is no reason not to use C.

However, experienced programmers may be able to implement algorithms that are more quick and efficient, taking full advantage of the target architecture, than C-generated assembly. As such, for the design of simple mid-level system functions such as firmware integration layers and drivers, assembly may be more appropriate. It is additionally useful when the compiler can not be trusted to create a maximally secure solution (see C's notorious history with terrible memory management vulnerabilities). Embedded systems developers may also benefit from programming directly in assembly because C has a tendency to be resource inefficient, not appropriately balancing between register, memory, and clock-time management based on context.

In either case, the use of interrupts as opposed to polling presents very simple advantages and drawbacks that require very little analysis. Interrupts are useful when rapid response times and high priorities are necessary. Polling, on the other hand, is useful when resources need to be conserved for other class or the data under consideration is only valuable to the system under some, but not all, circumstances (during which the polling should occur). From the prelab questions, the following still applies:

> **Polling** is a deliberate *check* by a device to check a data value, which can then be interpreted by the device and result in a branching path of behaviors. Thus, for a specific behavioral branch to occur, the device must poll something like a peripheral. Hypothetically, if the polling interval was wider than the frame in which the data changes, some data fluctuations may then be missed. If timing is of essence and the polling interval is wide, this is also probable. One solution to this problem would be for the peripheral to store a set of data, rather than the instananious value, which the poller can review upon data retrieval so as not to miss any queues. Polling is useful in situations where immediate reations are important and data does not need to be missed. For example, an automatic door could poll its motion sensor every second, as people approaching the door who need to be let in will not likely move in instantaneous chunks and freeze during the polling times.
>
> **Interrupts** are an action done *to* the microprocessor by an interrupt signal sent by another device. This results in a literal, physical modification to the interpretation of the program controller operates, causing it to disregard its regular interpretation of instructions in its program to execute a certain hardwired behavior corresponding to the interrupt vector's design. This is instantaneous. Interrupts are useful when the priority of the interrupting activity is high. For example, this may be useful to prevent a failure of a high-precision sub-millimeter CNC machine which must respond instantly and appropriately on a scale of microseconds- or even seconds- due to the sensitive nature of the microscopic physical world.

Instead of using the Wait function that was provided in BasicBumpBot.asm, is it possible to use a timer/counter interrupt to perform the one-second delays that are a part of the BumpBot behavior, while still using external

interrupts for the bumpers? Give a reasonable argument either way, and be sure to mention if interrupt priority had any effect on your answer.

This is possible. The ATMega128 board provides a plethora of time related signals and registers, many of which can be used as- or in conjunction with- interrupts.

It is better and more robust to practice designing using dedicated clocks rather than clock cycles. This helps make AVR design patterns more inter-device transferrable. In fact, it is foolish to assume a constant clock time at a fixed value unless it is specifically identified by the architecture for timing purposes, as there may be instructions which modify this in more advanced systems. The clock functionalities themselves, on the other hand, have actual guarantees given in exact human-pertinent temporal units.

Using functions like BumpBot's wait, on the other hand, is much simpler and may considerably reduce the solutions overall lines of code. This is a good idea if the maintainability or reliability of the system is a low priority, but extremely easy and rapid development is important, such as for ECE375 labs.

Interrupt priority did not have an effect on my answer. The choice to use interrupt priority should be compartmentalized away from the part responsible for timekeeping. This is a simple separation of responsibilities. Using one method over the other because of interrupt prioritization is a lazy practice and likely indicates a larger underlying design issue.

DIFFICULTIES

I didn't know how to do the initialization of the interrupt vectors in the AVR code, so I had to look it up.

https://www.electronicwings.com/avr-atmega/atmega1632-external-hardware-interrupts

I spent a lot of time in the debugger trying to figure out why some of my bumpbot movements were happening twice in a row per interrupt instead of just once per interrupt. It turns out that this was because of interrupt queuing. While this was mentioned in the slides, I totally forgot about it because this phenomenon wasn't mentioned in the lab report. This wasted a lot of my time, as I thought there was an issue in my logic (but there wasn't one)

CONCLUSION

This lab was an excellent exercise in the implementation of a robust embedded hardware driver. It was fascinating to see how much freedom I have when it comes to creating a relatively simple set of conditionals and operations, and it felt empowering to learn so much about different best practices that would help me write better assembly code. It also caused me to appreciate the C compi

Since I did the challenge code, I had to consider the structure of conditionals especially closely. The traditional paradigm of "if, then, else" wasn't the most easy thing to implement in AVR, so I ended up making a ton of nested "subroutines" controlled with rjmp instead of rcall.

SOURCE CODE

```
; ;***********************************************************
;*
;*      Prather_Eric_Lab6_sourcecode.asm
;*
```

```
;*      Program which responds to capacitive button inputs and
;*  counts the occurences via interrupts.
;***********************************************************
;*
;*       Author: Enter your name
;*         Date: Enter Date
;*
;***********************************************************

.include "m128def.inc"               ; Include definition file
;.include "interrupts"

;***********************************************************
;*      Internal Register Definitions and Constants
;***********************************************************
.def    mpr = r16                               ; Multipurpose register

;.equ   WskrR = 0                               ; Right Whisker Input Bit
;.equ   WskrL = 1                               ; Left Whisker Input Bit
;.equ INT0Addr = $0002               ; These are predefined in
                                                        ; m128def.inc so I can't
;.equ INT1Addr = $0004          ; redefine them here.


; The following is from lab 1
.def    waitcnt = r17                           ; Wait Loop Counter
.def    ilcnt = r18                             ; Inner Loop Counter
.def    olcnt = r19                             ; Outer Loop Counter

.equ    WTime = 100                             ; Time to wait in wait loop

.equ    WskrR = 0                               ; Right Whisker Input Bit
.equ    WskrL = 1                               ; Left Whisker Input Bit
.equ    EngEnR = 4                              ; Right Engine Enable Bit
.equ    EngEnL = 7                              ; Left Engine Enable Bit
.equ    EngDirR = 5                             ; Right Engine Direction Bit
.equ    EngDirL = 6                             ; Left Engine Direction Bit

;/////////////////////////////////////////////////////////////
;These macros are the values to make the TekBot Move.
;/////////////////////////////////////////////////////////////

.equ    MovFwd = (1<<EngDirR|1<<EngDirL)      ; Move Forward Command
.equ    MovBck = $00                            ; Move Backward Command
.equ    TurnR = (1<<EngDirL)                    ; Turn Right Command
.equ    TurnL = (1<<EngDirR)                    ; Turn Left Command
.equ    Halt = (1<<EngEnR|1<<EngEnL)            ; Halt Command

;***********************************************************
;*      Start of Code Segment
;***********************************************************
.cseg                                                   ; Beginning of code segment

;***********************************************************
;*      Interrupt Vectors
;***********************************************************
.org    $0000                                   ; Beginning of IVs
                rjmp    INIT                    ; Reset interrupt
                ; Set up interrupt vectors for any interrupts being used
                ; These are on page 59 of the datasheet.

                ; This is just an example:
;.org   $002E                                   ; Analog Comparator IV
;               rcall   HandleAC                ; Call function to handle interrupt
;               reti                                    ; Return from interrupt
.org INT0addr
                rjmp LeftBumperInterrupt
                reti
.org INT1addr
                rjmp RightBumperInterrupt
                reti
```

```
.org    $0046                               ; End of Interrupt Vectors

;***********************************************************
;*      Program Initialization
;***********************************************************
INIT:                                               ; The initialization routine
              ; Initialize Stack Pointer
              ldi           mpr, low(RAMEND)
              out           SPL, mpr
              ldi           mpr, high(RAMEND)
              out           SPH, mpr

              ;Port initalization copied from Lab 1 BasicBumpBot.asm
    ; Initialize Port B for output
              ldi           mpr, $FF              ; Set Port B Data Direction Register
              out           DDRB, mpr             ; for output
              ldi           mpr, $00              ; Initialize Port B Data Register
              out           PORTB, mpr            ; so all Port B outputs are low

        ; Initialize Port D for input
              ldi           mpr, $00              ; Set Port D Data Direction Register
              out           DDRD, mpr             ; for input
              ldi           mpr, $FF              ; Initialize Port D Data Register
              out           PORTD, mpr            ; so all Port D inputs are Tri-State

              ; Initalize array
              rcall ClearBumperMemory



              ; Initialize external interrupts
                     ; Set the Interrupt Sense Control to falling edge
              ; Instructions for how to do this provided by:
              ; https://www.avrfreaks.net/forum/external-interrupts-using-assembly
              ;ldi EICRA, (1<<ISC01) | (1<<ISC00)
              ldi mpr, 0b00001010 ;(1<<ISC01) | (1<<ISC10); (1<<0b00001010)
              sts EICRA, mpr ; Can't understand why we use sts when out is availible.


              ldi mpr, (1<<INTF0)
              out EIFR, mpr

              ; Configure the External Interrupt Mask
              ;ldi EIMSK, 1<<INT0
              ldi mpr, (1<<INT0) | (1<<INT1)
              out EIMSK, mpr


              ; Turn on interrupts
                     ; NOTE: This must be the last thing to do in the INIT function
              sei ; SHort for "set enable interrupt" (This is in the status register)
              ; Inverse of sei is cli

;***********************************************************
;*      Main Program
;***********************************************************
MAIN:                                               ; The Main program

              ldi mpr, MovFwd
              out PORTB, mpr

              rjmp    MAIN                  ; Create an infinite while loop to signify the
                                                    ; end of the program.

;***********************************************************
;*      Functions and Subroutines
;***********************************************************

;-----------------------------------------------------------
;       You will probably want several functions, one to handle the
```

```
;       left whisker interrupt, one to handle the right whisker
;       interrupt, and maybe a wait function
;-----------------------------------------------------------

;BumperMemoryState:
;       push ZL,
;       push ZR,
;       ldi ZL, low(BUMPER_MEMORY)
;       ldi ZH, high(BUMPER_MEMORY)
;
;       ret

; 0x20 = ASCII of ' '
; 0x?? = ASCII of 'L'
; 0x?? = ASCII of 'R'

ClearBumperMemory:
        push ZL
        push ZH
        push mpr
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        ldi mpr, 0x20
        st Z+, mpr
        st Z+, mpr
        st Z+, mpr
        st Z, mpr
        pop mpr
        pop ZH
        pop ZL
        ret

ShiftBumperMemoryLeft:
        push ZL
        push ZH
        push r16
        push r17
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        inc ZL
        inc ZL
        inc ZL
        ldi r16, 0x20
        ld r17, Z
        st Z, r16
        dec ZL
        ld r16, Z
        st Z, r17
        dec ZL
        ld r17, Z
        st Z, r16
        dec ZL
        st Z, r17
        pop r17
        pop r16
        pop ZH
        pop ZL
        ret

;-----------------------------------------------------------
; Func: LeftBumperInterrupt
; Desc: Performs memory management and branching conditional
;       when the left bumper is hit
;-----------------------------------------------------------
LeftBumperInterrupt:
        cli
        ;ldi r0, 0x00 ; // Conditional buffer
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        ld mpr, Z
```

```
        cpi mpr, 0x20 ; ' '
        breq LeftSimple
        inc ZL; Z -> BUMPER_MEMORY + 1
        cpi mpr, 0x4C ; 'L'
        breq LeftChain ;
        ; No challenge code detected
        ; Evaluate array item 2
LeftSimpleEval2:
        ld mpr, Z
        cpi mpr, 0x20
        breq LeftSimple
        inc ZL; Z -> BUMPER_MEMORY + 2
        cpi mpr, 0x4C
        breq LeftChainNo180A
        ; Evaluate array item 3
LeftSimpleEval3:
        ld mpr, Z
        cpi mpr, 0x20
        breq LeftSimple
        inc ZL ; Z -> BUMPER_MEMORY + 3
        cpi mpr, 0x4C
        breq LeftChainNo180B


        ; Evaluate array item 4 (different from 2,3)
LeftSimpleEval4:
        ld mpr, Z ; Z = BUMPER_MEMORY + 3
        cpi mpr, 0x20
        breq LeftSimple
        rcall ShiftBumperMemoryLeft ; Array is full, shift left
        cpi mpr, 0x4C
        breq LeftLong ; If L and last was L, big turn
        rjmp LeftSimple ; little turn



LeftChain:
        ; Case 1: ['L', 'L', '?', '?'] (Illegal state!)
        ld r17, Z
        cpi r17, 0x4C
        breq LeftSimpleEval2
        ; "Case 2: This is a regular 2x chain
        cpi r17, 0x20 ; [L, ' ', ' ', ' ']
        breq LeftLong
        ; Case 3: Array is [L,R,L,R]
        cpi r17, 0x52 ; 0x4C + num = 0x52 = 'R'
        brne LeftSimpleEval2

        inc ZL; Z <- BUMPER_MEMORY + 2
        ld r17, Z
        cpi r17, 0x4C
        brne LeftSimpleEval3
        inc ZL; Z <- BUMPER_MEMORY + 3
        ld r17, Z
        cpi r17, 0x52
        brne LeftSimpleEval4
        rjmp Left180

; we may have a special challenge code case:

; PRECONDITION: *BUMPER_MEMORY+1 == 'L'
LeftChainNo180A:
        ; Case 1: Next is L, in which case we are double
        ld mpr, Z
        cpi mpr, 0x20; *BUMPER_MEMORY+2 == ' '
        breq LeftLong
        ; Case 2: Next is R, in which case we are not a special case.
        rjmp LeftSimpleEval3

LeftChainNo180B:
        ; Case 1: Next is L, in which case we are double
```

```
        ld mpr, Z
        cpi r16, 0x20 ; *BUMPER_MEMORY+3 == 'L'
        breq LeftLong
        ; Case 2: Next is R, in which case we are not a special case.
        rjmp LeftSimpleEval4

;180:
Left180:
        ; Clear memory array
        rcall ClearBumperMemory
        rcall Turn180 ; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
        rjmp LeftDone


LeftLong:
        ; rewrite memory array => ['L',' ',' ',' ']
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        ldi mpr, 0x4C ; 'L'
        st Z+, mpr
        ldi mpr, 0x20 ; ' '
        st Z+, mpr
        st Z+, mpr
        st Z, mpr
        rcall SuperTurnRight; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
        rjmp LeftDone

LeftSimple: ; Does "hitLeft" then stores 'L' to the array at the current index.
        ldi mpr, 0x4C
        st Z, mpr
        rcall HitLeft ; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
LeftDone:
        ldi mpr, 1;(1<<INTF0)
        out EIFR, mpr
        sei
        ret

;-----------------------------------------------------------
; Func: RightBumperInterrupt
; Desc: Performs memory management and branching conditional
;       when the left bumper is hit
;-----------------------------------------------------------
RightBumperInterrupt:
        cli
        ;ldi r0, 0x00 ; // Conditional buffer
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        ld mpr, Z

        cpi mpr, 0x20 ; ' '
        breq RightSimple
        inc ZL; Z -> BUMPER_MEMORY + 1
        cpi mpr, 0x52 ; 'L'
        breq RightChain ;
        ; No challenge code detected
        ; Evaluate array item 2
RightSimpleEval2:
        ld mpr, Z
        cpi mpr, 0x20
        breq RightSimple
        inc ZL ; Z -> BUMPER_MEMORY + 2
        cpi mpr, 0x52
        breq RightChainNo180A
        ; Evaluate array item 3
RightSimpleEval3:
        ld mpr, Z
        cpi mpr, 0x20
        breq RightSimple
        inc ZL ; Z -> BUMPER_MEMORY + 3
        cpi mpr, 0x52
        breq RightChainNo180B
```

```
        ; Evaluate array item 4 (different from 2,3)
RightSimpleEval4:
        ld mpr, Z ; Z = BUMPER_MEMORY + 3
        cpi mpr, 0x20
        breq RightSimple
        rcall ShiftBumperMemoryLeft ; Array is full, shift Right
        cpi mpr, 0x52
        breq RightLong ; If L and last was L, big turn
        rjmp RightSimple ; little turn



RightChain:
        ; Case 1: ['L', 'L', '?', '?'] (Illegal state!)
        ld r17, Z
        cpi r17, 0x52
        breq RightSimpleEval2
        ; "Case 2: This is a regular 2x chain
        cpi r17, 0x20 ; [L, ' ', ' ', ' ']
        breq RightLong
        ; Case 3: Array is [L,R,L,R]
        cpi r17, 0x4C ; 0x52 + num = 0x4C = 'R'
        brne RightSimpleEval2

        inc ZL; Z <- BUMPER_MEMORY + 2
        ld r17, Z
        cpi r17, 0x52
        brne RightSimpleEval3
        inc ZL; Z <- BUMPER_MEMORY + 3
        ld r17, Z
        cpi r17, 0x4C
        brne RightSimpleEval4
        rjmp Right180

; we may have a special challenge code case:
RightChainNo180A:
        ; Case 1: Next is L, in which case we are double
        ld mpr, Z
        cpi mpr, 0x20; *BUMPER_MEMORY+2 == ' '
        breq RightLong
        ; Case 2: Next is R, in which case we are not a special case.
        rjmp RightSimpleEval3

RightChainNo180B:
        ; Case 1: Next is L, in which case we are double
        ld mpr, Z
        cpi r16, 0x20 ; *BUMPER_MEMORY+3 == ' '
        breq RightLong
        ; Case 2: Next is R, in which case we are not a special case.
        rjmp RightSimpleEval4

;180:
Right180:
        ; Clear memory array
        rcall ClearBumperMemory
        rcall Turn180 ; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
        rjmp RightDone


RightLong:
        ; rewrite memory array => ['L',' ',' ',' ']
        ldi ZL, low(BUMPER_MEMORY)
        ldi ZH, high(BUMPER_MEMORY)
        ldi mpr, 0x52 ; 'L'
        st Z+, mpr
        ldi mpr, 0x20 ; ' '
        st Z+, mpr
        st Z+, mpr
        st Z, mpr
```

```
        rcall SuperTurnLeft; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
        rjmp RightDone

RightSimple: ; Does "hitRight" then stores 'L' to the array at the current index.
        ldi mpr, 0x52
        st Z, mpr
        rcall HitRight ; COMMENT OUT DURING DEBUG MODE, UNCOMMENT DURING BUILD
RightDone:
        ldi mpr, 1;(1<<INTF0)
        out EIFR, mpr
        sei
        ret


;-----------------------------------------------------------
; Func: SuperTurnLeft
; Desc: Turns left for twice as long as HitRight
;       Mostly copies lab 1 code
;-----------------------------------------------------------
SuperTurnLeft:
                push    mpr                     ; Save mpr register
                push    waitcnt                 ; Save wait register
                in              mpr, SREG       ; Save program state
                push    mpr                     ;

                ; Move Backwards for a second
                ldi             mpr, MovBck     ; Load Move Backward command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Turn left for a second
                ldi             mpr, TurnL      ; Load Turn Left Command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; MY MODIFICATION HERE
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Move Forward again
                ldi             mpr, MovFwd     ; Load Move Forward command
                out             PORTB, mpr      ; Send command to port

                pop             mpr             ; Restore program state
                out             SREG, mpr       ;
                pop             waitcnt         ; Restore wait register
                pop             mpr             ; Restore mpr
                ret                             ; Return from subroutine

;-----------------------------------------------------------
; Func: SuperTurnRight
; Desc: Turns right for twice as long as HitLeft
;       Mostly copies the code
;-----------------------------------------------------------
SuperTurnRight:
                push    mpr                     ; Save mpr register
                push    waitcnt                 ; Save wait register
                in              mpr, SREG       ; Save program state
                push    mpr                     ;

                ; Move Backwards for a second
                ldi             mpr, MovBck     ; Load Move Backward command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Turn right for a second TWO TIMES
                ldi             mpr, TurnR      ; Load Turn Left Command
                out             PORTB, mpr      ; Send command to port
```

```
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function

                ; MY MODIFICATION HERE:
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function

                ; Move Forward again
                ldi             mpr, MovFwd    ; Load Move Forward command
                out             PORTB, mpr     ; Send command to port

                pop             mpr            ; Restore program state
                out             SREG, mpr      ;
                pop             waitcnt        ; Restore wait register
                pop             mpr            ; Restore mpr
                ret                            ; Return from subroutine

;-----------------------------------------------------------
; Func: Turn180
; Desc: Turns 180 degrees. Achieves by turning in one direction.
;       This lasts for four seconds.
;-----------------------------------------------------------
Turn180:
                push    mpr                    ; Save mpr register
                push    waitcnt                ; Save wait register
                in              mpr, SREG      ; Save program state
                push    mpr                    ;

                ; Move Backwards for a second
                ldi             mpr, MovBck    ; Load Move Backward command
                out             PORTB, mpr     ; Send command to port
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function

                ; Turn left for a second
                ldi             mpr, TurnL     ; Load Turn Left Command
                out             PORTB, mpr     ; Send command to port
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function

                ;MY MODIFICATION HERE
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function
                ldi             waitcnt, WTime ; Wait for 1 second
                rcall   Wait                   ; Call wait function

                ; Move Forward again
                ldi             mpr, MovFwd    ; Load Move Forward command
                out             PORTB, mpr     ; Send command to port

                pop             mpr            ; Restore program state
                out             SREG, mpr      ;
                pop             waitcnt        ; Restore wait register
                pop             mpr            ; Restore mpr
                ret                            ; Return from subroutine


;-----------------------------------------------------------
; Func: Template function header
; Desc: Cut and paste this and fill in the info at the
;               beginning of your functions
;-----------------------------------------------------------


;-----------------------------------------------------------
; Func: Template function header
; Desc: Cut and paste this and fill in the info at the
;               beginning of your functions
;-----------------------------------------------------------
FUNC:                                          ; Begin a function with a label
```

```
                ; Save variable by pushing them to the stack

                ; Execute the function here

                ; Restore variable by popping them from the stack in reverse order

                ret                                             ; End a function with RET

;-----------------------------------------------------------------
; LAB 1 ROUTINES
;-----------------------------------------------------------------

;-----------------------------------------------------------------
; Sub:  HitRight
; Desc: Handles functionality of the TekBot when the right whisker
;               is triggered.
;-----------------------------------------------------------------
HitRight:
                push    mpr                     ; Save mpr register
                push    waitcnt                 ; Save wait register
                in              mpr, SREG       ; Save program state
                push    mpr                     ;

                ; Move Backwards for a second
                ldi             mpr, MovBck     ; Load Move Backward command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Turn left for a second
                ldi             mpr, TurnL      ; Load Turn Left Command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Move Forward again
                ldi             mpr, MovFwd     ; Load Move Forward command
                out             PORTB, mpr      ; Send command to port

                pop             mpr             ; Restore program state
                out             SREG, mpr       ;
                pop             waitcnt         ; Restore wait register
                pop             mpr             ; Restore mpr
                ret                             ; Return from subroutine

;-----------------------------------------------------------------
; Sub:  HitLeft
; Desc: Handles functionality of the TekBot when the left whisker
;               is triggered.
;-----------------------------------------------------------------
HitLeft:
                push    mpr                     ; Save mpr register
                push    waitcnt                 ; Save wait register
                in              mpr, SREG       ; Save program state
                push    mpr                     ;

                ; Move Backwards for a second
                ldi             mpr, MovBck     ; Load Move Backward command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Turn right for a second
                ldi             mpr, TurnR      ; Load Turn Left Command
                out             PORTB, mpr      ; Send command to port
                ldi             waitcnt, WTime  ; Wait for 1 second
                rcall   Wait                    ; Call wait function

                ; Move Forward again
                ldi             mpr, MovFwd     ; Load Move Forward command
                out             PORTB, mpr      ; Send command to port
```

```
               pop             mpr             ; Restore program state
               out             SREG, mpr       ;
               pop             waitcnt         ; Restore wait register
               pop             mpr             ; Restore mpr
               ret                             ; Return from subroutine

;-----------------------------------------------------------------
; Sub:  Wait
; Desc: A wait loop that is 16 + 159975*waitcnt cycles or roughly
;               waitcnt*10ms.  Just initialize wait for the specific amount
;               of time in 10ms intervals. Here is the general eqaution
;               for the number of clock cycles in the wait loop:
;                       ((3 * ilcnt + 3) * olcnt + 3) * waitcnt + 13 + call
;-----------------------------------------------------------------
Wait:
               push    waitcnt                 ; Save wait register
               push    ilcnt                   ; Save ilcnt register
               push    olcnt                   ; Save olcnt register

Loop:   ldi             olcnt, 224              ; load olcnt register
OLoop:  ldi             ilcnt, 237              ; load ilcnt register
ILoop:  dec             ilcnt                   ; decrement ilcnt
               brne    ILoop                   ; Continue Inner Loop
               dec             olcnt           ; decrement olcnt
               brne    OLoop                   ; Continue Outer Loop
               dec             waitcnt         ; Decrement wait
               brne    Loop                    ; Continue Wait loop

               pop             olcnt           ; Restore olcnt register
               pop             ilcnt           ; Restore ilcnt register
               pop             waitcnt         ; Restore wait register
               ret                             ; Return from subroutine

;***********************************************************
;*      Stored Program Data
;***********************************************************
.dseg
.org $0100
BUMPER_MEMORY: ; Contiguous array of 'L', 'R', or ' '
       .byte 4

; Enter any stored data you might need here

;***********************************************************
;*      Additional Program Includes
;***********************************************************
; There are no additional file includes for this program
```