Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Microprocessor Systems
Course Number:	COE 538
Semester/Year (e.g.F2016)	F2023

Instructor:	Dr. Vadim Geurkov

Assignment/Lab Number:	Lab Project
Assignment/Lab Title:	Robot Guidance Challenge

Submission Date:	2023-12-06
Due Date:	2023-12-07

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Zhu	Janice	501040242	02	
Crowe	Dalton	501047978	15	D Crowe

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Content

Introduction

The COE538: Microprocessor Systems eeBot Project presented a challenging task given the time constraints imposed. This bot's requirements are to successfully traverse a line-maze without getting lost in three demonstration trials. This report outlines the successful implementation of this bot, as well as the challenges and sacrifices faced.

The generalized decision-making sequence implemented was to read the guider sensors to determine and act upon if left or right corrections were required to maintain adherence to the path. The outer most left and right sensors were then checked for alternate paths (intersections) before the bot would move forward incrementally. If an intersection was detected, the bot would move forward until it passed the intersection line to guarantee that all available paths have been found. At this point, the bot would choose, if available, to turn left, otherwise turn right. This was decided because it would result in the least dead ends encountered in the given maze. See figure 1.

While this project was successful in its task, the methods used to accomplish this were not what was originally planned. Initially the use of a PID function and adjusting the duty-cycle of the motors to maintain a consistent and uniform direction of travel was decided, but given the time constraints, additional testing required, and failed eeBot hardware supplied, this was scrapped to save time. In addition to this, the historic path-memory of the robot was reduced to only retain the previous intersection, but could be expanded if required. These sacrifices were made because it was decided that the completion of the challenge was more important than how gracefully it performed.

In the end, the successful demonstration of the eeBot's ability to autonomously navigate complex paths showcased the effectiveness of the implemented guidance and control algorithms.

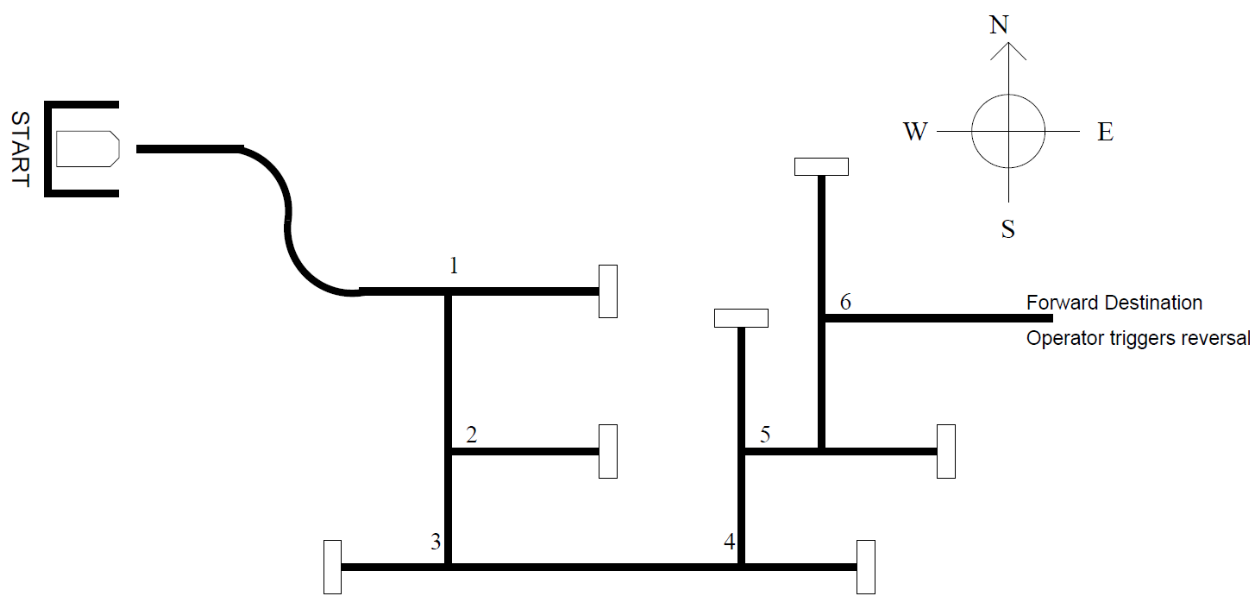

Figure 1: eeBot Guidance Challenge Maze Layout [1].

Case Analysis

Main Loop

The following code is the main execution loop that the eeBot cycles through. On startup, the applicable ports, ADC, and LCD are initialized before entering the runtime cycle. On each pass through the main section, the guider sensors are read first, followed by the bumpers. After this, an LCD refresh counter is compared against the desired refresh rate (every 20 cycles) and refreshes the display if conditions have been met; otherwise, continue to the dispatcher. Once the dispatcher is complete, the program starts a short delay (see table 2) before starting the loop again.

```
;*************************************************************
; MAIN CODE                                                  *
;*************************************************************
                    ORG         $4000                       ; Start of program text (FLASH memory)

Entry:
_Startup:
                    LDS         #$4000                      ; Initialize the stack pointer
                    CLI                                     ; Enable interrupts

                    JSR         INIT                        ; Initialize ports
                    JSR         openADC                     ; Initialize the ATD
                    JSR         openLCD                     ; Initialize the LCD
                    JSR         CLR_LCD_BUF                 ; Write space characters to LCD buffer

MAIN                LDAA        PTT
                    EORA        #$40
                    STAA        PTT
                    JSR         G_LEDS_ON                   ; Enable the guider LEDs
                    JSR         READ_SENSORS                ; Read the 5 guider sensors
                    JSR         G_LEDS_OFF                  ; Disable the guider LEDs
                    JSR         READ_BUMPERS
                    LDAB        DISP_REFRESH
                    CMPB        #LCD_REFRESH
                    BEQ         MAIN_CONT
                    JSR         DISPLAY_SENSORS             ; and write them to the LCD
                    LDAB        #0
                    STAB        DISP_REFRESH

MAIN_CONT           LDAA        STATE_CRNT
                    JSR         DISPATCHER
                    LDAB        DISP_REFRESH
                    INCB
                    STAB        DISP_REFRESH
                    LDY         #DLY_MAIN                   ; 150 ms delay to avoid 6000 = 300ms
                    JSR         del_50us                    ;  display artifacts
                    BRA         MAIN                        ; Loop forever

READ_BUMPERS        BRCLR       PORTAD0,$04,bowON
                    LDAA        #$31
                    BRA         bowOFF
bowON               LDAA        #$30
bowOFF              STAA        BUMPER_BOW

                    BRCLR       PORTAD0,$08,sternON
                    LDAA        #$31
                    BRA         sternOFF
sternON             LDAA        #$30
sternOFF            STAA        BUMPER_STERN
                    RTS
```

State Machine

The state machine is made of nine separate system states, and a dispatcher unit to direct program execution. See figures 2 and 3.

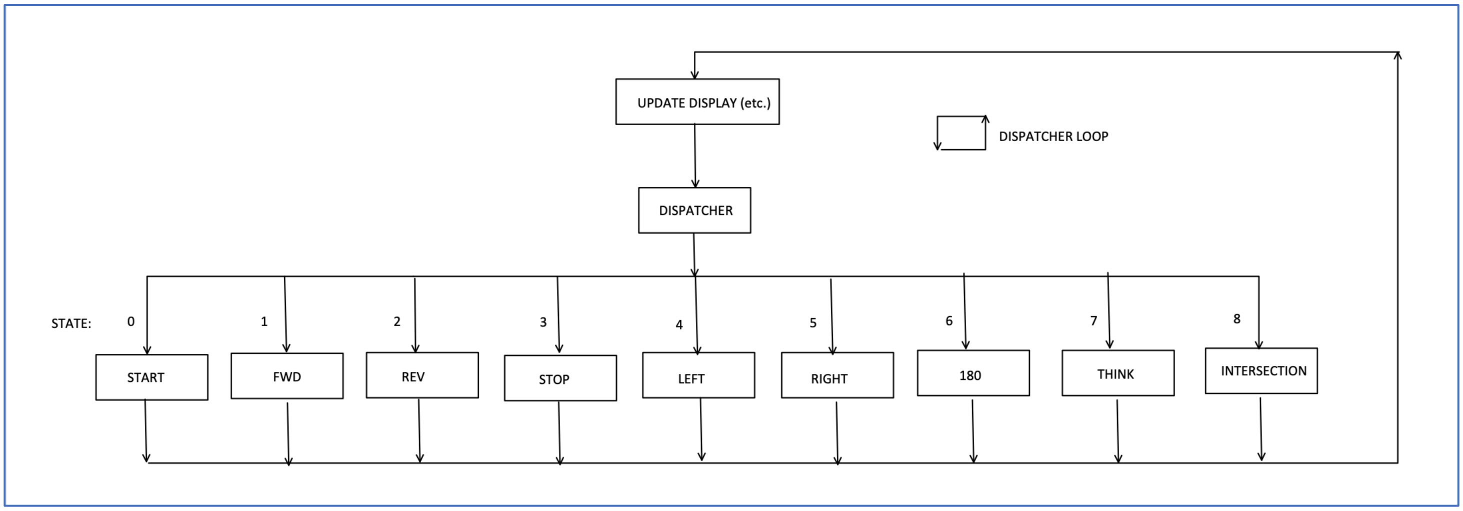

Figure 2: The dispatcher loop.

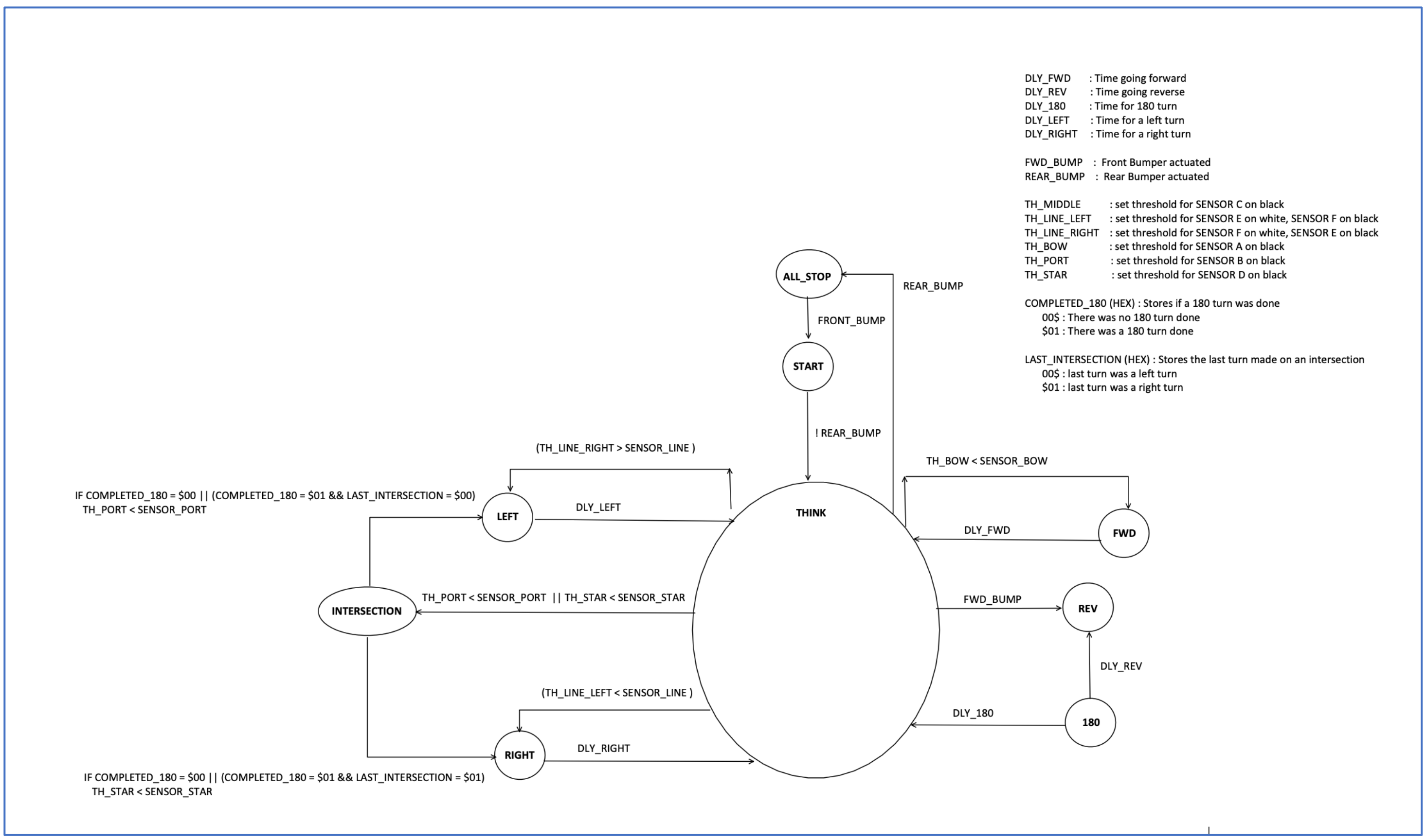

Figure 3: The robot state machine.

Dispatcher

The dispatcher directs control to corresponding subroutines based on STATE_CRNT, a variable that stores the current state, shown in **Figure 2**.

```
; Dispatcher
DISPATCHER          CMPA        #SS_START                   ; Start state
                    BNE         NOT_SS_START
                    JSR         STATE_START
                    BRA         DISPATCHER_EXIT

NOT_SS_START        CMPA        #SS_STOP                    ; Stop state
                    BNE         NOT_SS_STOP
                    JSR         STATE_STOP
                    BRA         DISPATCHER_EXIT

NOT_SS_STOP         CMPA        #SS_FWD                     ; Forward state
                    BNE         NOT_SS_FWD
                    JSR         STATE_FWD
                    BRA         DISPATCHER_EXIT

NOT_SS_FWD          CMPA        #SS_REV                     ; Reverse state
                    BNE         NOT_SS_REV
                    JSR         STATE_REV
                    BRA         DISPATCHER_EXIT

NOT_SS_REV          CMPA        #SS_LEFT                    ; Left turn state
                    BNE         NOT_SS_LEFT
                    JSR         STATE_LEFT
                    BRA         DISPATCHER_EXIT

NOT_SS_LEFT         CMPA        #SS_RIGHT                   ; Right turn state
                    BNE         NOT_SS_RIGHT
                    JSR         STATE_RIGHT
                    BRA         DISPATCHER_EXIT

NOT_SS_RIGHT        CMPA        #SS_180                     ; Right turn state
                    BNE         NOT_SS_180
                    JSR         STATE_180
                    BRA         DISPATCHER_EXIT

NOT_SS_180          CMPA        #SS_THINK                   ; Right turn state
                    BNE         NOT_SS_THINK
                    JSR         STATE_THINK
                    BRA         DISPATCHER_EXIT

NOT_SS_THINK        CMPA        #SS_INTER                   ; intersection state
                    BNE         NOT_SS_INTERSECTION
                    JSR         STATE_INTERSECTION
                    BRA         DISPATCHER_EXIT

NOT_SS_INTERSECTION SWI
DISPATCHER_EXIT     RTS
```

Stop State

The stop state disables both motors and checks the front bumper for activation, signifying the start of the maze activity.

```
; Stop State
STATE_STOP          BCLR        PTT,%00110000
                    BRSET       PORTAD0,$04,NOT_START        ; Check if front bumper is pushed
                    JSR         INIT_SS_STOP                 ; if so, initialize stop
                    MOVB        #SS_START,STATE_CRNT         ; set current tate to start
                    BRA         STOP_EXIT                    ; exit

NOT_START           NOP
STOP_EXIT           RTS

INIT_SS_STOP        BCLR        PTT,%00110000
                    RTS
```

Think State

The think state is where the bot sets the current system state to intersection if a flag is set; otherwise, it sets the system state to forward.

```
; Think State
STATE_THINK         BCLR        PTT,%00110000
                    LDAA        PATH_PORT                    ; no left path found
                    CMPA        #$01
                    BNE         CHK_STAR_INT
                    MOVB        #SS_INTER,STATE_CRNT
                    RTS

CHK_STAR_INT        LDAA        PATH_STAR                    ; no right path found
                    CMPA        #$01
                    BNE         GO_FWD
                    MOVB        #SS_INTER,STATE_CRNT
                    RTS

GO_FWD              MOVB        #SS_FWD,STATE_CRNT
                    RTS

INIT_SS_THINK       BCLR        PTT,%00110000
                    RTS
```

Start State

The start state waits until the forward bumper is released before initializing the forward state.

```
; Start State
STATE_START         BCLR        PTT,%00110000
                    BRCLR       PORTAD0,$08,NO_FWD           ; Check if rear bumper is released
                    MOVB        #SS_FWD,STATE_CRNT           ; Set current state to forward
                    BRA         START_EXIT

NO_FWD              NOP
START_EXIT          RTS
```

Forward Sate

The forward state is where the front and rear bumpers are polled, intersections are detected, and line tracking is monitored. This works by first detecting if the rear bumper is active, stopping the bot. Next the front bumper is checked signifying a dead-end. After that, the left and right sensors are checked for intersections. If one is found, it marks it in the intersection bytes. At this point, the front sensor is checked and moves the bot forward if it is on a line. If it's not on the line, then the line sensor is checked to see if the bot is still centered on the maze path-line. If it's not, then initiate a left or right turn to correct. See table 1 for sensor threshold values used.

```
; Forward State
STATE_FWD       BCLR    PTT,%00110000
                BRSET   PORTAD0,$08,NO_STOP        ; Check if the rear bumper is triggered
                JSR     INIT_SS_STOP               ; Initialize the all stop state
                MOVB    #SS_STOP,STATE_CRNT        ; Set the current state to all stop
                BRA     FWD_EXIT                   ; Return

NO_STOP         BRSET   PORTAD0,$04,NO_REV         ; Check if the front bumper is active
                MOVB    #SS_REV,STATE_CRNT         ; Set the current state to reverse
                BRA     FWD_EXIT                   ; Return

NO_REV          LDAA    #TH_PORT                   ; Check if left sensor has found a line
                CMPA    SENSOR_PORT
                BHI     PORT_NOT_BLK               ; If not, then branch
                BSET    PATH_PORT,#$01             ; Otherwise mark it on the map
                MOVB    #$31,DEBUG_1
                MOVB    #SS_THINK,STATE_CRNT
                BRA     FWD_EXIT

PORT_NOT_BLK    LDAA    #TH_STAR                   ; Check if right sensor found a line
                CMPA    SENSOR_STAR
                BHI     STAR_NOT_BLK               ; If not, then branch
                BSET    PATH_STAR,#$01             ; Otherwise mark it on the map
                MOVB    #$31,DEBUG_2
                MOVB    #SS_THINK,STATE_CRNT
                BRA     FWD_EXIT

STAR_NOT_BLK    LDAA    #TH_BOW                    ; Check if the front is on a line
                CMPA    SENSOR_BOW
                BLO     BOW_IS_BLK                 ; If it is, branch

                LDAA    #TH_LINE_LEFT              ; Check if line follow is left of line
                CMPA    SENSOR_LINE
                BLO     INIT_SS_RIGHT              ; if th > sensor, start a right turn

                LDAA    #TH_LINE_RIGHT             ; Check if line follow is right of line
                CMPA    SENSOR_LINE
                BHI     INIT_SS_LEFT               ; if th < sensor, start a left turn

BOW_IS_BLK      MOVB    #SS_THINK,STATE_CRNT
                BRA     INIT_SS_FWD                ; On the line

FWD_EXIT        RTS

INIT_SS_FWD     BCLR    PORTA,%00000011            ; Set both motor directions to forward
                BSET    PTT,%00110000              ; Turn on the drive motors
                LDY     #DLY_FWD
                JSR     del_50us
                BCLR    PTT,%00110000              ; Turn off drive motors
                RTS
```

Left State

The left state turns on the right wheel and disables the left allowing for a slow and smooth left turn.

```
; Left State
STATE_LEFT          BCLR        PORTA,%00000011
                    BSET        PTT,%00100000
                    BCLR        PTT,%00010000
                    LDY         #DLY_LEFT
                    JSR         del_50us
                    BCLR        PTT,%00110000
                    MOVB        #SS_THINK,STATE_CRNT
                    RTS

INIT_SS_LEFT        MOVB        #SS_LEFT,STATE_CRNT
                    RTS
```

Right State

The right state is identical to the left state, except the active and inactive motors are swapped.

```
; Right State
STATE_RIGHT         BCLR        PORTA,%00000011
                    BSET        PTT,%00010000
                    BCLR        PTT,%00100000
                    LDY         #DLY_RIGHT
                    JSR         del_50us
                    BCLR        PTT,%00110000
                    MOVB        #SS_THINK,STATE_CRNT
                    RTS

INIT_SS_RIGHT       MOVB        #SS_RIGHT,STATE_CRNT
                    RTS
```

Intersection State

The intersection state works by first checking if a 180° turn has been performed since the most recent intersection. If it has, then it clears the intersection flag that contains the turn that the bot had previously made before it made the incorrect turn. If a 180° turn has not been performed recently, then it does not modify the flags. At this point, the current state checks which paths have been found and starts a turn in order of priority; left then right. Once a turn is initiated, the outermost sensor in the direction of travel is checked until it is no longer on a line. At this point, the front sensor is then checked until it finds the next line where it will discontinue the intersection turn and clear the 180° flag.

```
; Intersection State
STATE_INTERSECTION  BCLR        PTT,%00110000
                    LDAA        COMPLETED_180
                    CMPA        #$00
                    BEQ         CHK_PORT
                    LDAA        INTERSEC_LAST
                    CMPA        #$00
                    BEQ         RMV_STAR

RMV_PORT            BCLR        PATH_PORT,#$01
                    BRA         CHK_PORT
RMV_STAR            BCLR        PATH_STAR,#$01

CHK_PORT            LDAA        PATH_PORT
                    CMPA        #$01
                    BNE         CHK_IF_STAR
                    LDAA        #TH_PORT   ;a0
                    CMPA        SENSOR_PORT
                    BHI         CHK_BOW                      ; if not on line
                    MOVB        #SS_LEFT,STATE_CRNT
                    BCLR        INTERSEC_LAST,#$01
                    MOVB        #$30,DEBUG_2
                    RTS
CHK_BOW             LDAA        #TH_BOW ;a0
                    CMPA        SENSOR_BOW
                    BLO         INTERSECT_DONE               ; if not on line
                    MOVB        #SS_LEFT,STATE_CRNT
                    BCLR        INTERSEC_LAST,#$01
                    RTS
CHK_IF_STAR         LDAA        PATH_STAR
                    CMPA        #$01
                    BNE         INTER_EXIT
                    LDAA        #TH_STAR
                    CMPA        SENSOR_STAR
                    BHI         CHK_BOW2
                    MOVB        #SS_RIGHT,STATE_CRNT
                    BSET        INTERSEC_LAST,#$01
                    MOVB        #$31,DEBUG_2
                    RTS

CHK_BOW2            LDAA        #TH_BOW
                    CMPA        SENSOR_BOW
                    BLO         INTERSECT_DONE               ; if not on line
                    MOVB        #SS_RIGHT,STATE_CRNT
                    BSET        INTERSEC_LAST,#$01
                    RTS
INTERSECT_DONE      BCLR        PATH_PORT,#$01
                    BCLR        PATH_STAR,#$01
                    MOVB        #$30,DEBUG_1
                    MOVB        #$30,DEBUG_2
                    MOVB        #SS_THINK,STATE_CRNT
                    BCLR        COMPLETED_180,#$01
                    RTS
INTER_EXIT          MOVB        #SS_THINK,STATE_CRNT
                    RTS
```

Reverse State

The reverse state sets both motors to reverse and drives them for a short period of time before stopping them.

```
; Reverse State
STATE_REV         BSET      PORTA,%00000011          ; Set both motor directions to reverse
                  BSET      PTT,%00110000            ; Turn on the drive motors
                  LDY       #DLY_REV
                  JSR       del_50us
                  BCLR      PTT,%00110000            ; Turn off the drive motors
                  MOVB      #SS_180,STATE_CRNT
                  BRA       REV_EXIT                 ; Return

REV_EXIT          RTS

INIT_SS_REV       BSET      PORTA,%00000011          ; Set both motor directions to reverse
                  BSET      PTT,%00110000            ; Turn on the drive motors
                  RTS
```

180° Turn State

In the 180° turn state, both motors are activated with opposite rotation direction for a short period of time before they are deactivated again.

```
; 180 Degree Turn State
STATE_180         BCLR      PORTA,%00000001          ; Set both motor directions to reverse
                  BSET      PTT,%00110000            ; Turn on the drive motors
                  LDY       #DLY_180
                  JSR       del_50us
                  BCLR      PTT,%00110000            ; Turn off the drive motors
                  MOVB      #SS_THINK,STATE_CRNT
                  BRA       SPIN_EXIT                ; Return

SPIN_EXIT         BSET      COMPLETED_180,#$01
                  MOVB      #$31,DEBUG_1
                  RTS

INIT_SS_180       BCLR      PORTA,%00000010          ; Set right motor direction to forward
                  RTS
```

System Initialization

This function initializes the ports that will be used in the program.

```
; Initialization
INIT              BCLR      DDRAD,$FF                ; Make PORTAD an input (DDRAD @ $0272)
                  BSET      DDRA,$FF                 ; Make PORTA an output (DDRA @ $0002)
                  BSET      DDRB,%11110000           ; Make PORTB an output (DDRB @ $0003)
                  BSET      DDRJ,%11000000           ; Make pins 7,6 of PTJ outputs
                  BSET      DDRT,%01110000
                  BSET      ATDDIEN,$0C
                  RTS
```

Software Delay

This routine creates a software delay of $50\mu s$.

```
; Software Delay
del_50us        PSHX
eloop           LDX         #300
iloop           NOP
                DBNE        X,iloop
                DBNE        Y,eloop
                PULX
                RTS
```

ADC Initialization

This routine initializes the ADC, reused from *the eebot Guider* [2].

```
; Open ADC
openADC         MOVB        #$80,ATDCTL2            ; Turn on ADC (ATDCTL2 @ $0082)
                LDY         #1                      ; Waitfor50usforADCtobeready
                JSR         del_50us                ; -"-
                MOVB        #$20,ATDCTL3            ; 4 conversions on channel AN1
                MOVB        #$97,ATDCTL4            ; 8-bit resolution, prescaler=48
                RTS
```

LCD

The following routines are used to control the Liquid Crystal Display.

Clear Buffer

This routine writes space characters into the LCD display buffer to prepare it for the building of a new display buffer at the start of the program, reused from *the eebot Guider* [2].

```
; Clear Buffer
CLR_LCD_BUF     LDX         #CLEAR_LINE
                LDY         #TOP_LINE
                JSR         STRCPY

CLB_SECOND      LDX         #CLEAR_LINE
                LDY         #BOT_LINE
                JSR         STRCPY

CLB_EXIT        RTS
```

Copy String

This subroutine copies a null-terminated string from one location to another, reused from *the eebot Guider* [2].

```
; Copy String
STRCPY          PSHX                            ; Protect the registers used
                PSHY
                PSHA

STRCPY_LOOP     LDAA        0,X                 ; Get a source character
                STAA        0,Y                 ; Copy it to the destination
                BEQ         STRCPY_EXIT         ; If it was the null, then exit
                INX                             ; Else increment the pointers
                INY
                BRA         STRCPY_LOOP         ; and do it again

STRCPY_EXIT     PULA                            ; Restore the registers
                PULY
                PULX
                RTS
```

Display Sensor

This routine writes the sensor values in hexadecimal to the LCD and uses the 'shadow buffer' approach, taken from *the eebot Guider* [2]. The physical layout of the data displayed on the LCD is as follows:

FF_MM_LL_____CS_____

PP_SS_____FR__DD_____

Where FF is the front, MM is middle, LL is the line, PP is port, and SS is starboard sensor. CS is the current state, FR is the front/rear bumper, and DD is for debugging.

Definitions

The corresponding addresses in the LCD buffer are defined in the following equates. The display position is the MSDigit.

```
; LCD Position Definitions
DP_FRONT_SENSOR     EQU         TOP_LINE+0
DP_MID_SENSOR       EQU         TOP_LINE+3
DP_LINE_SENSOR      EQU         TOP_LINE+6
DP_STATE            EQU         TOP_LINE+13

DP_PORT_SENSOR      EQU         BOT_LINE+0
DP_STBD_SENSOR      EQU         BOT_LINE+3
DP_BUMPERS          EQU         BOT_LINE+9
DP_DEBUG            EQU         BOT_LINE+13
```

Display

In this subroutine, each dataset is converted to ASCII (if applicable) and rendered onto the LCD it's defined location.

```
; Display Sensors
DISPLAY_SENSORS     LDAA        SENSOR_BOW              ; Get the FRONT sensor value
                    JSR         BIN2ASC
                    LDX         #DP_FRONT_SENSOR        ; Point to the LCD buffer position
                    STD         0,X

                    LDAA        SENSOR_PORT
                    JSR         BIN2ASC
                    LDX         #DP_PORT_SENSOR
                    STD         0,X

                    LDAA        SENSOR_MID
                    JSR         BIN2ASC
                    LDX         #DP_MID_SENSOR
                    STD         0,X

                    LDAA        SENSOR_STAR
                    JSR         BIN2ASC
                    LDX         #DP_STBD_SENSOR
                    STD         0,X

                    LDAA        SENSOR_LINE
                    JSR         BIN2ASC
                    LDX         #DP_LINE_SENSOR
                    STD         0,X

                    LDAA        BUMPER_BOW
                    LDAB        BUMPER_STERN
                    LDX         #DP_BUMPERS
                    STD         0,X

                    LDAA        DEBUG_1
                    LDAB        DEBUG_2
                    LDX         #DP_DEBUG
                    STD         0,X

                    LDAA        STATE_CRNT
                    JSR         BIN2ASC
                    LDX         #DP_STATE
                    STD         0,X

                    LDAA        #CLEAR_HOME
                    JSR         cmd2LCD

                    LDY         #40
                    JSR         del_50us

                    LDX         #TOP_LINE
                    JSR         putsLCD

                    LDAA        #LCD_SEC_LINE
                    JSR         LCD_POS_CRSR

                    LDX         #BOT_LINE
                    JSR         putsLCD
                    RTS
```

Initialization

This routine initializes the LCD of 4-bit data width, 2-line display, reused from *Lab 2: Programming the I/O Devices* [7]. It turns on the display, cursor off, blinking off, and shifting cursor right.

```
; Initialize
openLCD         BSET    DDRB,%11110000          ; set PS pins 7,6,5,4 to output
                BSET    DDRJ,%11000000          ; configure pins PJ7,PJ6 for output
                LDY     #2000                   ; wait for LCD to be ready
                JSR     del_50us                ; -"-
                LDAA    #INTERFACE              ; set 4-bit data, 2-line display
                JSR     cmd2LCD                 ; -"-
                LDAA    #CURSOR_OFF             ; display on, cursor off, blinking off
                JSR     cmd2LCD                 ; -"-
                LDAA    #SHIFT_OFF              ; move cursor right after character
                JSR     cmd2LCD                 ; -"-
                RTS
```

Clear

This routine clears the display and home cursor, reused from *Lab 2: Programming the I/O Devices* [7].

```
; Clear LCD
clrLCD          LDAA    #$01                    ; clear cursor and return to home
                JSR     cmd2LCD                 ; -"-
                LDY     #40                     ; wait for "clear cursor" command
                JSR     del_50us                ; -"-
                RTS
```

Send Command

This function sends a command in accumulator A to the LCD, reused from *Lab 2: Programming the I/O Devices* [7].

```
; Send a command
cmd2LCD         BCLR    LCD_CNTR,LCD_RS         ; Select the LCD Instruction register
                JSR     dataMov                 ; Send data to IR or DR of the LCD
                RTS
```

Print Character

This function outputs the character in accumulator A to LCD, reused from *Lab 2: Programming the I/O Devices* [7].

```
; Print a character
putcLCD         BSET    LCD_CNTR,LCD_RS         ; select the LCD Data register
                JSR     dataMov                 ; send data to IR or DR of the LCD
                RTS
```

Print String

This function outputs a NULL-terminated string pointed to by X, reused from *Lab 2: Programming the I/O Devices* [7].

```
; Print a string
putsLCD         LDAA    1,X+
                BEQ     donePS
                JSR     putcLCD
                BRA     putsLCD
donePS          RTS
```

Send Data

This function sends data to the LCD IR or DR depending on RS, reused from *Lab 2: Programming the I/O Devices* [7].

```
; Send Data
dataMov         BSET        LCD_CNTR,LCD_E          ; pull the LCD E-sigal high
                STAA        LCD_DAT                 ; send the upper 4 bits of data to LCD
                BCLR        LCD_CNTR,LCD_E          ; pull the LCD E-signal low to finish.
                LSLA                                ; match the lower 4 bits with LCD pins
                LSLA                                ; -"-
                LSLA                                ; -"-
                LSLA                                ; -"-
                BSET        LCD_CNTR,LCD_E          ; pull the LCD E signal high
                STAA        LCD_DAT                 ; send the lower 4 bits of data to LCD
                BCLR        LCD_CNTR,LCD_E          ; pull the LCD E-signal low to finish.
                LDY         #1                      ; adding this delay will finish ops
                JSR         del_50us                ; operation for most instructions
                RTS
```

Position Cursor

This routine positions the display cursor to prepare for the display of a character or string for a 20x2 display, reused from *the eebot Guider* [2]. The first line runs from 0 to 19, and the second line runs from 64 to 83.

```
; Set Cursor Position
LCD_POS_CRSR    ORAA        #%10000000              ; Set the high bit of the control word
                JSR         cmd2LCD                 ;  and set the cursor address
                RTS
```

Guider

The following routines read the eebot guider sensors and displays the values on the Liquid Crystal Display. The guider uses four brightness sensors and one differential pair of sensors of photo resistive cells. The voltage across the cells is measured through the HCS12 A/D converter channel AN1. Therefore, the sensor reading increases as the sensor becomes darker like when it’s over a black line.

LED’s on/off

This routine enables/disables the guider LEDs by setting/clearing Port A5, reused from *the eebot Guider* [2]. The readings of the sensors correspond to the ‘ambient lighting’ situation.

```
; LED's On
G_LEDS_ON       BSET        PORTA,%00100000         ; Set bit 5
                RTS

; LED's Off
G_LEDS_OFF      BCLR        PORTA,%00100000         ; Clear bit 5
                RTS
```

Select Sensor

This routine selects the sensor number passed in ACCA, taken from *the eebot Guider* [2]. Bits PA2, PA3, PA4 are connected to a 74HC4051 analog mux on the guider board, which selects the guider sensor to be connected to AN1. Motor direction bits 0, 1, guider sensor select bit 5 and unused bits 6,7 in the same register PORTA are not affected.

```
; Select Sensor
SELECT_SENSOR       PSHA                            ; Save the sensor number for the moment

                    LDAA        PORTA               ; Clear the sensor selection bits
                    ANDA        #%11100011
                    STAA        TEMP                ; and save it into TEMP

                    PULA                            ; Get the sensor number
                    ASLA                            ; Shift the selection number left 2x
                    ASLA
                    ANDA        #%00011100          ; Clear irrelevant bit positions

                    ORAA        TEMP                ; OR it into the sensor bit positions
                    STAA        PORTA               ; Update the hardware
                    RTS
```

Read Sensors

This routine reads the eebot guider sensors and puts the results in RAM registers, reused from *the eebot Guider* [2]. The A/D conversion mode used in this routine is to read the A/D channel AN1 four times into HCS12 data registers ATDDR0, 1, 2, 3.

```
; Read Sensors
READ_SENSORS        CLR         SENSOR_NUM          ; Select sensor number 0
                    LDX         #SENSOR_LINE        ; Point to start of the sensor array

RS_MAIN_LOOP        LDAA        SENSOR_NUM          ; Select the correct sensor input
                    JSR         SELECT_SENSOR       ;  on the hardware
                    LDY         #400                ; 20 ms delay to allow the
                    JSR         del_50us            ;  sensor to stabilize

                    LDAA        #%10000001          ; Start A/D conversion on AN1
                    STAA        ATDCTL5
                    BRCLR        ATDSTAT0,$80,*     ; Repeat until A/D signals done

                    LDAA        ATDDR0L             ; A/D conversion is complete in ATDDR0L
                    STAA        0,X                 ;  so copy it to the sensor register
                    CPX         #SENSOR_STAR        ; If this is the last reading
                    BEQ         RS_EXIT             ; Then exit

                    INC         SENSOR_NUM          ; Else, increment the sensor number
                    INX                             ; and the pointer into the sensor array
                    BRA         RS_MAIN_LOOP        ; and do it again

RS_EXIT             RTS
```

Binary to ASCII

Converts an 8-bit binary values in ACCA to the equivalent ASCII character to character string in accumulator D using a table-driven method, reused from *the eebot Guider* [2].

```
; Binary to ASCII
HEX_TABLE           FCC         '0123456789ABCDEF'

BIN2ASC             PSHA
                    TAB
                    ANDB        #%00001111
                    CLRA
                    ADDD        #HEX_TABLE
                    XGDX
                    LDAA        0,X

                    PULB
                    PSHA
                    RORB
                    RORB
                    RORB
                    RORB
                    ANDB        #%00001111
                    CLRA
                    ADDD        #HEX_TABLE
                    XGDX
                    LDAA        0,X
                    PULB
                    RTS
```

System Interrupts

This code section defines the entry interrupt vector for initial execution.

```
; Reset Vector
                    ORG         $FFFE
                    DC.W        Entry
```

References

[1] P. Hiscocks and V. Geurkov, "Robot Guidance Challenge", COE538 Microprocessor Systems. Available: https://www.ecb.torontomu.ca/~courses/coe538/Project/project.pdf [Accessed: November 23, 2023]

[2] P. Hiscocks and V. Geurkov, "The eebot Guider", COE538 Microprocessor Systems. Available: https://www.ecb.torontomu.ca/~courses/coe538/Project/Guider.pdf [Accessed: November 23, 2023]

[3] P. Hiscocks, "State Machines in Software", *Circuit Cellar: The Computer Applications Journal*, no. 26, Apr., pp. 52-60, 1992.

[4] P. Spasov, "Section 13.3, Sequential Machines", in *Microcomputer Technology: The 68HC11*, 2nd ed., Prentice Hall, 1996.

[5] Motorola, *S12CPUV2 Reference Manual Rev. 0*, https://motorola.com/semiconducters, Jul. 2003.

[6] H.-W. Huang, *HCS12/9S12: An Introduction to Software and Hardware Interfacing*, 2nd ed., Delmar Cengage Learning, 2010.

[7] P. Hiscocks and V. Geurkov, "Lab 2: Programming the I/O Devices", COE538 Microprocessor Systems. Available: https://www.ecb.torontomu.ca/~courses/coe538/Labs/lab2.pdf [Accessed: November 23, 2023]

Appendix

Table 1: Threshold values used.

Sensor Threshold Values	
Variable Name	**Value**
TH_LINE_LEFT	$CE
TH_LINE_RIGHT	$A8
TH_MIDDLE	$A0
TH_BOW	$A0
TH_PORT	$B0
TH_STAR	$60

Table 2: Delay values used.

Delay Values		
Variable Name	**Value**	**Effective Time (ms)**
DLY_FWD	2000	100
DLY_LEFT	2000	100
DLY_RIGHT	3250	163
DLY_REV	3500	175
DLY_180	18500	925
DLY_MAIN	250	13