COE538 Microprocessor Systems
Lab 5: Robot Roaming Program[1]

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson University
phiscock@ee.ryerson.ca

Contents

List of Figures

1 Overview

The objective of this programming exercise is to design, install and test a program to guide the *eebot* robot in a simple roaming pattern.

[1] This lab was adapted to be used with the HCS12 microcontroller by V. Geurkov.

Robot *roaming* behavior can be obtained with a very simple set of rules. Initially, the robot drives in a straight line. If it doesn't encounter any obstacles, after a certain interval it stops, executes a turn, and then runs again in a straight line.

If the robot encounters an obstacle, it executes a *back-and-turn* manoeuvre. It drives straight backward for a fixed interval and then briefly disables one motor to cause the vehicle heading to change. Then it resumes driving straight forward again.

This behavior must be translated into a working control program.

2 A Robot Control Program

A frontal attack on the robot control program would be to

- translate the description into pseudocode, using structured decision statements such as **If...Then...Else**, **While...Do** or **Repeat...Until**.
- further convert the pseudocode into assembly language.

However, the robot control program is complicated, with lots of decision branches. As a result, the code has the potential to become difficult to understand, and therefore difficult to debug. It is especially difficult to add features to this type of program. For example, what does the robot do if it encounters a disturbance while backing away from a previous obstacle? Adding code to take care of this situation or others like it can introduce bugs that are difficult to eradicate.

There is a better approach: the *state machine*.

2.1 The State Machine

The basic idea is this:

- The behavior of the robot is treated as a series of states – `DRIVING_FORWARD`, `DRIVING_BACKWARD`, `BACKING_TURN`, `FORWARD_TURN` and `ALL_STOP`. At any given time, the robot is in one of these states.
- The conditions that cause the robot to change from one state to another must be defined. These are known as the *transition conditions*.
- The machine starts in a particular state. When the appropriate transition condition occurs, the machine moves to a new state.

There are a number of advantages in structuring the program as a state machine:

- Some variable `MACHINE_STATE` can be made to contain the current state of the machine. This can be displayed to monitor what the machine is actually doing (as opposed to what we think it should be doing!) Being able to determine the current state of the machine is an invaluable debugging tool, especially in complicated machines.
- When adding new features to the program, new states and transition conditions can be added without introducing bugs in other parts of the program.
- It is possible to encode the state diagram into a data structure known as a *state table*. When the state machine is encoded into a state table, it is straightforward to check for improper behavior. The table is a 2 dimensional matrix of possible states in one dimension versus possible input conditions in the other dimension. The action of the state machine is entered at each row-column intersection. If there is a blank entry, then the behavior of the machine in that state and transition condition, is undefined. For mission critical applications, this would identify a possible fault situation that must be rectified.

  We will not use a table driven approach, but for the curious the table driven approach is described in the References, section 5 on page 18.

The state machine can be used in various ways. At its most basic, it can be used as design tool to identify in a very systematic way what should be the behavior of the device. Then the state diagram could be translated into structured pseudocode.

State machines are useful for handling a wide variety of programming applications, not just the operation of robots. They are particularly useful in handling input signals from a human-computer interface. They are also the key component in *parsing* commands strings, as in a software interpreter.

2.2 The Robot State Diagram

A state diagram for the robot behavior is shown in figure 1 on page 3. Each circle represents a possible state of the

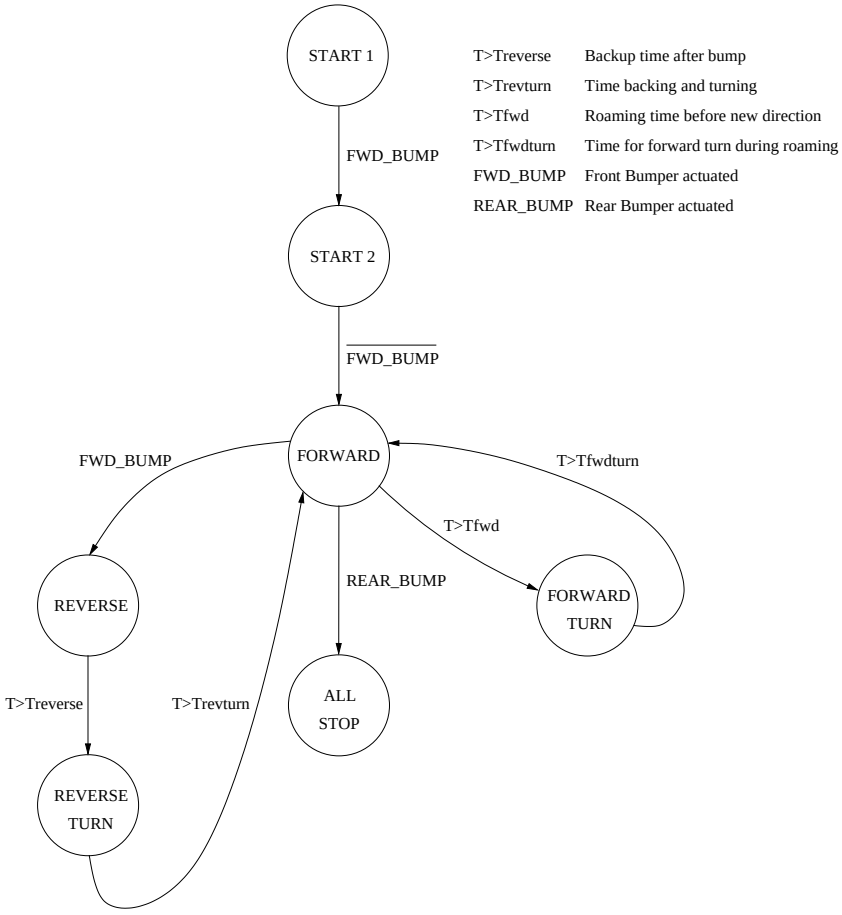

Figure 1: Robot State Machine

robot. The labels next to the directed arcs represent transition conditions that cause the robot to move to another state. For example, the transition condition $T > T_{end}$ means *when the current time T exceeds the ending time T_{end}, make the transition to the next state*.

The default condition is to not change states. If the transition condition does *not* occur, the state machine stays in its current state. Some people like to show a transition looping back into the state, but that's only necessary if there is another transition condition to consider.

Roaming Behavior

When *roaming*, the the robot proceeds in a straight line for a few seconds, then turns to a new vehicle direction, and then resumes moving in a straight line.

- The machine begins in the `START 1` state. When the operator actuates the `FRONT BUMPER`, the state machine unconditionally moves into the state `START 2`. When the operate releases the `FRONT BUMPER`, the machine moves into the `FORWARD` state. This activates both the motors and the robot begins to move forward in an approximately straight line.

  Why do we need the state `START 2`? Without it, the state machine will move directly into the `FORWARD` state, detect the `FRONT BUMPER` again, and immediately go into the `REVERSE` state. This is not what is wanted, so the state machine must wait for the `FRONT BUMPER` switch to be released before it transitions into the `FORWARD` state.

- When the time interval T_{fwd} elapses, the robot moves into state `FORWARD_TURN`. The starboard motor stops briefly, which causes the robot to pivot to starboard (clockwise rotation, viewed from the top)[2].

- When the $T_{fwdturn}$ interval is complete, the robot moves back into the `FORWARD` state, both motors are running, and the robot again proceeds in a straight line.

Obstacle Behavior

When the robot is moving forward and it encounters an obstacle, it should back up and then make a backing turn. Then it should resume moving forward. This occurs in two steps.

- Obstacles are only detected when the robot is in the `FORWARD` state. When a `FWD_BUMP` condition occurs, the machine transitions to the `REVERSE` state. It stays in this state until time $T_{reverse}$ elapses.

- The robot enters state `REVERSE_TURN`, which causes the starboard motor to stop briefly and a backing turn to occur.

- When the T_{return} interval is complete, the machine moves back into the `FORWARD` state and proceeds again in a straight line.

Finishing Up

When the machine is in the `FORWARD` state and the `REAR BUMPER` is actuated by the operator, the machine stops both motors and comes to rest.

2.3 State Machine Structure: First Attempt

The obvious way to write the state machine is to use *goto* statements to transfer control among the state routines. As a block diagram, this program looks exactly like the state diagram of figure 1. As pseudocode, the program might look like figure 2.

Each state is a block of code with a starting address named after the state and `GOTO` exit instructions. (In assembly language, the `GOTO`'s would be coded as `JMP` or `BRA` instructions.) The program loops in each state until some transition condition becomes true and then it transfers to the next state.

(For simplicity here, we show only how the flow of the program transfers control between the various states. We have left out the commands to actuate the motors. There would need to be commands to turn on the motors appropriately when entering each of these states. We'll deal with this later.)

This program will work and it has the virtue of simplicity. However, it has a severe limitation: while it's looping in a state, it's not doing anything useful. For example, suppose we want the program to read and display the battery voltage on a continuous basis. How could we do this? Because the program could be tied up in

[2]For a sharper turn, you may wish to reverse the direction of the starboard motor, instead of turning it off completely

```
Start          GOTO Forward

Forward        IF       {Fwd Bumper Detected}       THEN GOTO Reverse
               ELSEIF   {Rear Bumper Detected}      THEN GOTO All_Stop
               ELSEIF   {Time > Forward Time}       THEN GOTO Forward_Turn
               ELSE     GOTO Forward

Reverse        IF       {Time > Reverse Time}       THEN GOTO Reverse_Turn
               ELSE     GOTO Reverse

Reverse_Turn   IF       {Time > Reverse Turn Time}  THEN GOTO Forward
               ELSE     GOTO Reverse_Turn

Forward_Turn   IF       {Time > Forward Turn Time}  THEN GOTO Forward
               ELSE     GOTO Forward_Turn

All_Stop       GOTO     All_Stop
```

Figure 2: Robot State Controller: First Attempt

some of these states for long periods of time, we'd have to call the `Battery_Display` subroutine from those subroutines. This is inconvenient and awkward[3].

There is a better way to do this, using a *state dispatcher*.

2.4 State Machine Structure: Using a *Dispatcher*

According to The Canadian Oxford Dictionary, a *dispatcher* is

> *a person who coordinates the departure of taxis, busses, trains, etc.*

In this case, the state dispatcher is a block of software that directs the flow of the program to the correct state.

The key concept of the state dispatcher is this: rather than have the program loop in any given state, *loop the program through the dispatcher and some state subroutine.* This nets us a big advantage: any routine that must be updated on a regular basis can be included in this loop by prepending it to the dispatcher. A diagrammatic representation of the program is shown in figure 3 on page 6.

The structure of the state machine for the `START`, `FORWARD`, `FORWARD_TURN`, `REVERSE`, `REVERSE_TURN` and `ALL_STOP` states is shown as the pseudo-code given in figure 4 on page 7. In this diagram, the states are given names. These states would be the names of various routines which would be called by branch or jump instructions from the dispatcher.

(Again, the motor commands are omitted for clarity.) The *State Dispatcher* is at the heart of the state machine. Its function is very simple: it calls the appropriate subroutine based on the `CURRENT_STATE`. In a high level language, this would be a `CASE` statement. We'll see in a moment how it translates into assembly language source code.

Each state is implemented as a subroutine. A state subroutine checks the various transition conditions and if none of them are met the routine simply returns. However, when a state subroutine detects that a particular transition condition has been met it changes the `CURRENT_STATE` to the appropriate new value and returns. This new value of `CURRENT_STATE` then causes the state dispatcher to transfer control to some other state subroutine [4].

Notice that the program never loops in a delay in any of the state subroutines or anywhere else. It loops through the state dispatcher and the current state subroutine.

[3] It is possible to use the *real-time interrupt* to solve this problem. The program is interrupted at regular intervals (typically 60 times per second) and the real-time interrupt handler routine then executes tasks that need to be serviced on a regular basis. The HCS12 has hardware specifically designed to provide this feature. However, interrupts have their own problems so we'll chose a non-interrupt solution for this particular application.

[4] Strictly speaking, the state machine represented in figure 3 moves to the START state immediately after the microcontroller has been RESET (without waiting for the FWD_BUMP signal). To make it more realistic, the initial current state assignment must look like this: CURRENT_STATE := ALL_STOP. The corresponding state diagram is given in figure 9 on page 14.

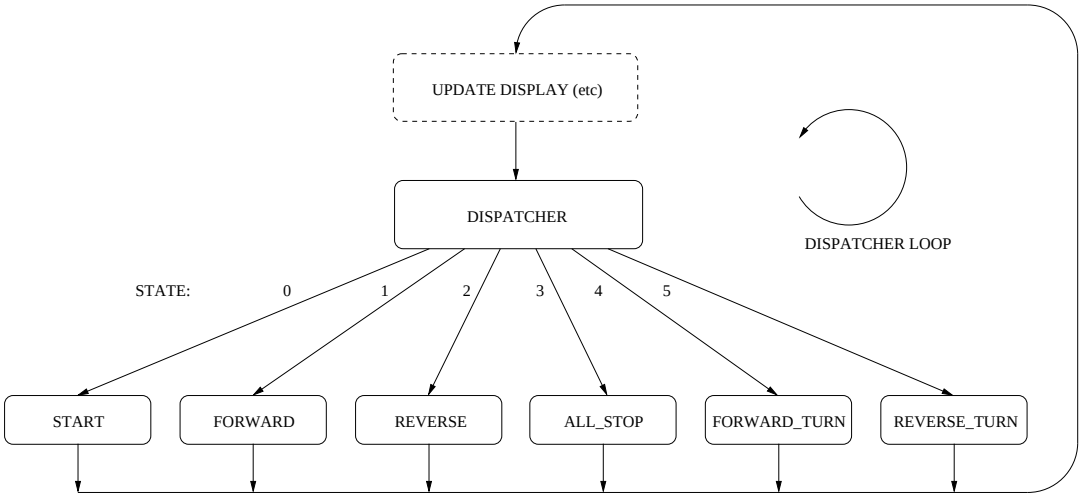

Figure 3: Dispatcher Concept

2.5 The State Dispatcher in Assembly Language

Now that we have some idea of the structure of the state machine controller, let us examine how the state dispatcher translates into assembly language.

The state dispatcher, represented in the pseudocode of figure 4 by the statement `CALL CURRENT_STATE`, can be implemented as a giant `IF` statement where each state is represented by a number and the dispatcher calls the appropriate subroutine based on the current state. This is shown in figure 5 on page 8.

Things to notice:

- The `CURRENT_STATE` is allocated one byte in RAM with an `RMB 1` assembler directive. For monitoring purposes, some other routine (not shown) could read this variable and put something on a display to indicate the current state of the robot.

- The various possible states are defined in assembler directive `EQU` statements. The microprocessor can only work with binary numbers so it is necessary that each state be represented by a number eventually. However, in the assembly language source code it facilitates reading the code if the states are given symbolic names. Then the reader doesn't have to remember that the state number **1** is the `FORWARD` state.

  This illustrates the general concept that we should write the assembly language in the form that is clearest to a human reader and let the assembler turn that information into a stream of bytes that the machine can interpret. More succinctly: *Let the assembler do the work.*

- The dispatcher is implemented as a subroutine, starting with the routine name `DISPATCHER` and ending with an `RTS` statement. So the dispatcher would be called with the invocation

  ```
  LDAA CURRENT_STATE
  JSR DISPATCHER
  ```

  Any routine that has to be called on a regular basis would be placed before these statements.

- The routine has one entrance and one exit point. Each successful test for a state comparison calls the appropriate subroutine and then exits via the `DISP_EXIT` address (the `RTS` instruction). We could have placed an `RTS` instruction at the conclusion of each test, instead of the `JMP DISP_EXIT` instruction. However, it simplifies debugging if the dispatcher is structured with only one entry and one exit point: it simplifies the placing of breakpoints during debugging. This is a general rule that should be followed in the construction of all subroutines.

```
                CURRENT_STATE := START

LOOP:           CALL CURRENT_STATE              ; Call the 'CURRENT STATE' subroutine
                GOTO LOOP                       ;  forever

START:          CURRENT_STATE := FORWARD        ; The START state subroutine
                Return

FORWARD:        If FWD_BUMP then                ; The FORWARD state subroutine
                  CURRENT_STATE := REVERSE
                  Mark the time
                  Return
                If REAR_BUMP then
                  CURRENT_STATE := ALL_STOP
                  Return
                If T>Tfwd then
                  CURRENT_STATE := FORWARD_TURN
                  Mark the time
                  Return
                Else
                  Return

FORWARD_TURN:   If T>Tfwdturn then              ; The FORWARD_TURN state subroutine
                  CURRENT_STATE := FORWARD
                  Mark the time
                  Return
                Else
                  Return

REVERSE:        If T>Treverse then              ; The REVERSE state subroutine
                  CURRENT_STATE := REVERSE_TURN
                  Mark the time
                  Return
                Else
                  Return

REVERSE_TURN:   If T>Trevturn then              ; The REVERSE_TURN state subroutine
                  CURRENT_STATE := FORWARD
                  Mark the time
                  Return
                Else
                  Return

ALL_STOP:       CURRENT_STATE := ALL_STOP       ; The ALL_STOP state subroutine
                Return
```

Figure 4: State Machine Pseudocode

- If the `CURRENT_STATE` is not successfully compared with any of the states (say it had become the value 37, for example), then the program will arrive at the address `NOT_FORWARD_TURN`, in which case we break to the monitor program with an `SWI` instruction. The chances of this happening are remote, especially once the program is debugged, but it is better to have this simple *error-checking* feature than nothing at all. An even better solution would be to call some routine to display an error message.
- The comments do not mimic the code operation. For example, the following comment is not useful:

  ```
  LDAA FOOBAR    ; Load the accumulator with FOOBAR
  ```

  Presumably, we can determine what the instruction does exactly by reading the code, so the comment is not helpful. What we really need to know is *what is intent or objective of this instruction in the context of the larger program?* So, much more helpful is a comment like

  ```
  LDAA FOOBAR    ; Set up for calling the state dispatcher
  ```

Further, you can see that some instructions are not commented. Sometimes, it is clearer not to comment at all. And a few clear comments are much more valuable than unclear comments on every line.

Good comments take careful thought and creativity.

```
; Current state
CURRENT_STATE RMB  1

; Definitions of the various states
START         EQU  0
FORWARD       EQU  1
REVERSE       EQU  2
ALL_STOP      EQU  3
FORWARD_TURN  EQU  4
REVERSE_TURN  EQU  5

*********************************************************************
*                     State Dispatcher
*
*  This routine calls the appropriate state handler based on the current
*   state.
*  Input:     Current state in ACCA
*  Returns:   None
*  Clobbers:  Everything

; State Dispatcher
DISPATCHER         CMPA #START              ; If it's the START state
                   BNE  NOT_START
                   JSR  START_STATE         ;  then call the START routine
                   JMP  DISP_EXIT           ;  and exit

NOT_START          CMPA #FORWARD            ; Else if it's the FORWARD state
                   BNE  NOT_FORWARD
                   JSR  FORWARD_STATE       ;  then call the FORWARD routine
                   JMP  DISP_EXIT           ;  and exit

NOT_FORWARD        CMPA #REVERSE            ; Else if it's the REVERSE state
                   BNE  NOT_REVERSE
                   JSR  REVERSE_STATE       ;  then call the REVERSE routine
                   JMP  DISP_EXIT           ;  and exit

NOT_REVERSE        CMPA #ALL_STOP           ; Else if it's the ALL_STOP state
                   BNE  NOT_ALL_STOP
                   JSR  ALL_STOP_STATE      ;  then call the ALL_STOP routine
                   JMP  DISP_EXIT           ;  and  exit

NOT_ALL_STOP       CMPA #FORWARD_TURN       ; Else if it's the FORWARD_TURN state
                   BNE  NOT_FORWARD_TURN
                   JSR  FORWARD_TURN_STATE  ;  then call the FORWARD_TURN routine
                   JMP  DISP_EXIT           ;  and  exit

NOT_FORWARD_TURN   CMPA #REVERSE_TURN       ; Else if it's the REVERSE_TURN state
                   BNE  NOT_REVERSE_TURN
                   JSR  REVERSE_TURN_STATE  ;  then call the REVERSE_TURN state
                   JMP  DISP_EXIT           ;  and exit

NOT_REVERSE_TURN   SWI                      ; Else the current state is not defined, so stop
DISP_EXIT          RTS                      ; Exit from the state dispatcher
```

Figure 5: State Dispatcher in Assembly Language

2.6 The State Routines in Assembly Language

Now we examine the internal details of some state routines.

The START Routine

This routine immediately transitions into the FORWARD routine, so it's very simple (see figure 6).

```
*************************************************************
*                START STATE HANDLER
*
*   Advances state to the FORWARD state if /FWD-BUMP.

* Passed:   Current state in ACCA
* Returns:  New state in ACCA
* Clobbers: None

START_STATE  BRCLR PORTAD0,$04,NO_FORWARD   ; If /FWD_BUMP
             JSR   INIT_FWD                 ; Initialize the FORWARD state
             MOVB  #FORWARD,CURRENT_STATE   ; Go into the FORWARD state
             BRA   START_EXIT

NO_FORWARD   NOP                            ; Else
START_EXIT   RTS                            ;  return to the MAIN routine
```

Figure 6: START State Routine

The routine is about to transition into the FORWARD state, so it should do the initializations required by that state. The routine to initialize the FORWARD state is shown in figure 7 on page 10.

The routine INIT_FWD turns on both the motors and then sets alarm time, T_{fwd}, in the way that was discussed in Lab 4. We assume that the subroutine BOTH_MOTORS_ON exists somewhere else in the program. The routine calls this to enable the motors.

Notice that the initialization of the FORWARD state occurs *in the states that call it*. It can't occur in the FORWARD state itself because then it would be initialized repeatedly, every trip around the dispatcher-state loop, which is not what we want.

Back now to START_STATE: having initialized the FORWARD state it advances the state to FORWARD and exits. The state dispatcher that we discussed earlier will send the program to the FORWARD_STATE routine.

The FORWARD Routine

This state routine is shown in figure 8 on page 11. It's complicated by the number of possible branches out of the FORWARD state to other states.

Notes:

- The bumper is checked with the instruction

  ```
  BRSET PORTAD0,$04,NO_FWD_BUMP    ; If FWD_BUMP then
  ```

  You used such an instruction in Lab 3.

- The last conditional branch is BNE NO_FWD_TURN. We could have written this as BNE FWD_EXIT. However, as a minor matter of style, the NO_FWD_TURN label is more informative and we can associate it with a NOP instruction so that the effect is equivalent. This gives us a place to put the ELSE comment, which improves readability. (Your opinion may differ.)

- Notice that the order in which these various conditions are checked implicitly sets a *priority* to the events. For example, the first condition checked is whether the bumper is actuated. If it is set, that determines the next state, regardless of the state of the various timers.

```
*****************************************************************
*                INITIALIZE 'FORWARD' STATE
*
*   This routine is called whenever the 'FORWARD' routine is entered.
*   It turns both the motors ON
*   It initializes the alarm used in by the FORWARD state.

* Passed:   Nothing
* Returns:  Nothing
* Clobbers: None

INIT_FWD    JSR BOTH_MOTORS_ON     ; Turn on the drive motors (if theyre off!)
            LDAA TOFCOUNTER        ; Mark the fwd time Tfwd
            ADDA #FWD_INTERVAL
            STAA T_FWD
            RTS
```

Figure 7: Initializing the FORWARD State

2.7 Organization of the State Machine Program

From top to bottom, the Robot Roaming program will consist of:

- Equates and Main Loop
- Dispatcher and State Subroutines
- Utility Subroutines

The program is organized as it would be decomposed: from top to bottom, starting with the main routine through to the utility subroutines. The contents of each of those three sections are now listed in greater detail.

Equates and Main Loop

- equates of the symbolic addresses, using `EQU` directives. For example, the various machine registers such as `ATDxCTLy` would be defined here[5].
- definitions of constants such as the time delays, also using `EQU` directives
- start of the `DATA` area at $3850 using an `ORG` directive.
- the working registers in RAM, using `ds.b` directives
- pre-initialized message strings (for the LCD) using `dc.b"...",0` directives.
- start of the `CODE` or `TEXT` area at $4000 using an `ORG` directive. A good name for the entry point of this code would be `STARTUP`: this is where the program starts.
- instructions that initialize the system. For example, this is where the `TOF` counter interrupt would be initialized and global interrupts enabled.
- the *main loop* routine, named `MAIN`. This calls any subroutines that must operate on a repetitive basis (such as updating the LCD), followed by the state dispatcher. The state dispatcher is called as a subroutine with the instruction sequence:

```
LDAA CURRENT_STATE
JSR DISPATCHER
```

[5]Many of the required equates are done in the .inc file, therefore you don't need to re-define these definitions in your program

```
*****************************************************************
*                FORWARD STATE HANDLER
*
* Algorithm:
*          If FWD_BMP then
*              Initialize the REVERSE state
*              Change the state to REVERSE
*              Return
*          If REAR_BUMP then
*              Initialize the ALL_STOP state.
*              Change the state to ALL_STOP
*              Return
*          If Tc>Tfwd then
*              Initialize the FORWARD_TURN state
*              Change the state to FORWARD_TURN
*              Return
*          Else
*              Return
*
* Passed:   Current state in ACCA
* Returns:  New state in ACCA
* Clobbers: Everything, probably. Make no assumptions.

FORWARD_ST    BRSET PORTAD0,$04,NO_FWD_BUMP       ; If FWD_BUMP then
              JSR  INIT_REVERSE                   ; initialize the REVERSE routine
              MOVB  #REVERSE,CURRENT_STATE        ; set the state to REVERSE
              JMP  FWD_EXIT                       ; and return

NO_FWD_BUMP   BRSET PORTAD0,$08,NO_REAR_BUMP      ; If REAR_BUMP, then we should stop
              JSR  INIT_ALL_STOP                  ; so initialize the ALL_STOP state
              MOVB  #ALL_STOP,CURRENT_STATE       ; and change state to ALL_STOP
              JMP  FWD_EXIT                       ; and return

NO_REAR_BUMP  LDAA TOFCOUNTER                     ; If Tc>Tfwd then
              CMP  T_FORWARD                      ; the robot should make a turn
              BNE  NO_FWD_TURN                    ; so
              JSR  INIT_FORWARD_TURN              ; initialize the FORWARD_TURN state
              MOVB  #FORWARD_TURN,CURRENT_STATE   ; and go to that state
              JMP  FWD_EXIT

NO_FWD_TURN   NOP                                 ; Else
FWD_EXIT      RTS                                 ; Return to the MAIN routine.
```

Figure 8: Forward State Handler

The `DISPATCHER` subroutine calls the appropriate state subroutine, which executes and returns to the dispatcher. The dispatcher subroutine then completes and returns control back to the main loop.

The last instruction in the main loop is a `JMP MAIN` instruction to return control back to the start of the main loop.

Dispatcher and State Subroutines

These subroutines make up the state machine.

- The *dispatcher* directs control to the appropriate subroutine based on the variable `CURRENT_STATE`. It's the traffic cop of the state machine.
- There is one *state routine*, one for each state in the diagram. Each state routine examines various transition conditions, selects what should happen next (the *next state*), and initializes the necessary conditions when state transition is to occur.
- Various *state initialization* subroutines contain the instructions that set up for entry into a new state.

Utility Subroutines

This area of the program is a collection of all the other subroutines necessary to make the program a working robot. For example, the following subroutines would be found here:

- Initialize and service the interrupt driven Timer Overflow counter.
- Read the A/D converter (used optionally to read the battery voltage).
- Actuate the robot motors.
- Check the various alarm timers.

2.8 A Strategy for Building the Robot Roaming Program

1. The first step involves no programming as such. *Ensure that you understand very clearly how the program is supposed to operate.* There is no way to debug a program when you do not understand every detail of it's operation. If something is obscure, get help from your lab instructor.
2. Collect and organize the *Utility Subroutines* section of the program. These are subroutines that were developed in previous lab exercises, so they should be functional as is. If there is any doubt that they are working properly, retest them. Make sure it is clear what each routine is passed and returns. Problems often occur at the interfaces to routines.
3. Build a subsection of the state machine. Implementing the `START`, `FORWARD` and `ALL_STOP` states might be a good choice. This will require writing the section of the dispatcher that deals with these states and the three state routines themselves. Debug this by breakpointing the dispatcher and checking that the state advances as it is supposed to do.
4. Add in the remaining states and test the completed state machine.

2.9 Bench Testing

You will not have a lot of time to debug this program on the actual robot. However, you can assemble the program and run it on the microprocessor bench systems.

Modify the bumper detect routine to use the potentiometer on channel AN05 of the bench system, and then you can test whether the bumper routines work by rotating the pot up and down.

If your main loop includes some sort of a display routine of the current state, then you can check that the entire program, except for the motor drive routines, is operating correctly.

You should have some indication on the LCD of what state the machine is in. This is a very helpful indication of what the machine is doing at any given time. A one-digit number or letter is sufficient.

It is also very helpful if your program triggers the `ALIVE` indicator to show that it is looping through the state machine dispatcher. Then, if the program crashes for some reason, you will know immediately.

At this point, you are ready to move the program onto a robot. The only things left to test should be that the bumper routine and the motor drive instructions work on the actual hardware. Since these were developed and tested earlier, they should be working correctly.

2.10 Operating the *eebot*

Here are the steps in operating the mobile *eebot*.

1. Ensure that your program is debugged to the extent possible on the bench system. It is much more difficult diagnosing software problems on the mobile robot.
2. Ensure that the robot battery is charged.
3. Disconnect the serial cable from the bench system and plug it into the mobile robot.

4. Turn on the robot LOGIC power.
5. Use CodeWarrior to download the working program into the mobile robot.
6. Start the program.
7. Disconnect the serial cable.
8. Move the robot over to the demo area.
9. Turn on the robot MOTOR power. The motors should not run at this point.
10. Place the robot in the demo area and actuate the front bumper.
11. The robot should start running in the FORWARD state and go through its paces.
12. When you want to stop the robot, actuate the rear bumper. If the robot is out of control, switch off the MOTOR power.

3 The Assignment

Your assignment is to write the Robot Roaming program as described in the state diagram of figure 9 on page 14.

- A passing grade will be assigned if the machine can correctly move through two states on the microprocessor bench system.
- A full mark will be assigned if the machine executes all the states on the microprocessor bench system. In this case, you will be provided a battery powered robot and can turn it loose with the Robot Roaming program providing guidance.
- Bonus marks will be given if the machine can simultaneously show battery voltage and current state on the display while operating the guidance program.

Some hints on creating the program can be found in Appendix A.

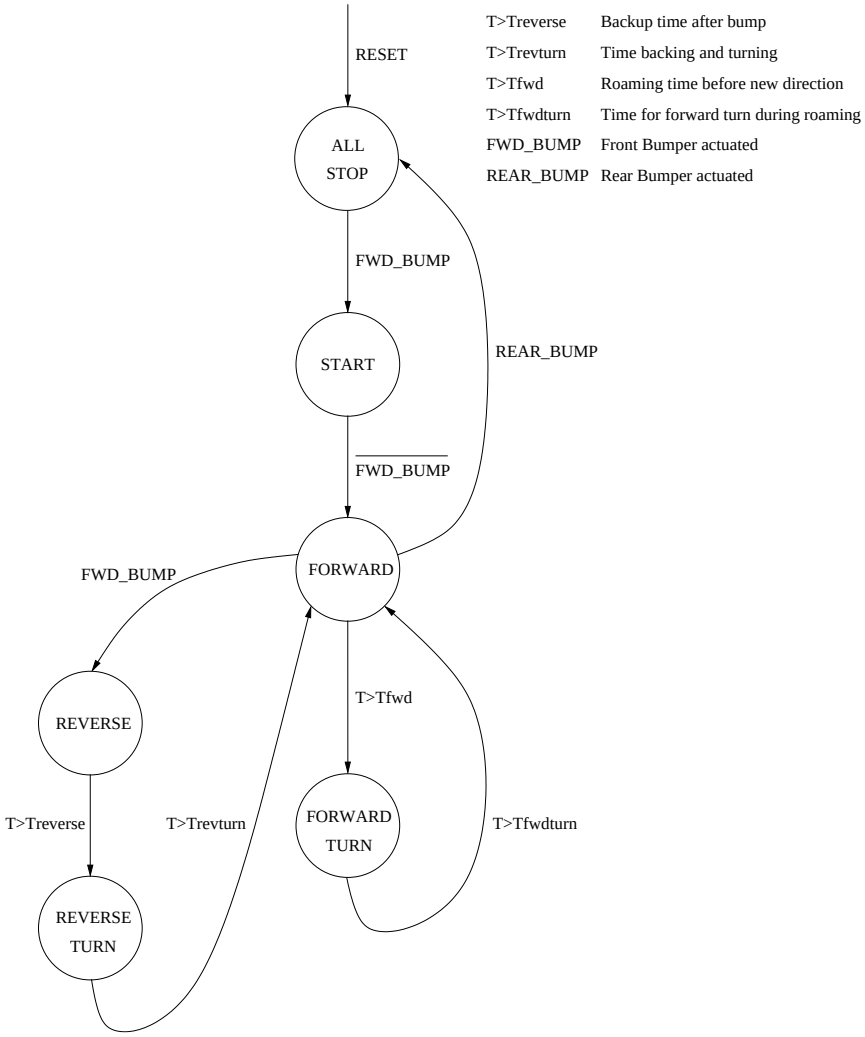

Figure 9: Realistic Robot State Machine

4 Appendix A

The general structure of the program is given below.

```
;*****************************************************************
; Lab 5: Robot Roaming Program (9S32C)                           *
;*****************************************************************

; equates section
;*****************************************************************
LCD_DAT      ...
LCD_CNTR     ...
LCD_E        ...
LCD_RS       ...
FWD_INT      EQU   69                    ; 3 second delay (at 23Hz)
REV_INT      EQU   69                    ; 3 second delay (at 23Hz)
FWD_TRN_INT  EQU   46                    ; 2 second delay (at 23Hz)
REV_TRN_INT  EQU   46                    ; 2 second delay (at 23Hz)
```

```
START        EQU   0
FWD          EQU   1
REV          EQU   2
ALL_STP      EQU   3
FWD_TRN      EQU   4
REV_TRN      EQU   5

; variable section
***********************************************************************
             ORG   $3850                    ; Where our TOF counter register lives
TOF_COUNTER  dc.b  0                        ; The timer, incremented at 23Hz
CRNT_STATE   dc.b  3                        ; Current state register
T_FWD        ds.b  1                        ; FWD time
T_REV        ds.b  1                        ; REV time
T_FWD_TRN    ds.b  1                        ; FWD_TURN time
T_REV_TRN    ds.b  1                        ; REV_TURN time
TEN_THOUS    ds.b  1                        ; 10,000 digit
THOUSANDS    ds.b  1                        ;  1,000 digit
HUNDREDS     ds.b  1                        ;    100 digit
TENS         ds.b  1                        ;     10 digit
UNITS        ds.b  1                        ;      1 digit
NO_BLANK     ds.b  1                        ; Used in 'leading zero' blanking by BCD2ASC

; code section
***********************************************************************
             ORG   $4000                    ; Where the code starts ---------------------
Entry:                                      ;                                           |
_Startup:                                   ;                                           |
             CLI                            ; Enable interrupts                         |
             LDS   #$4000                   ; Initialize the stack pointer
                                            ;                                           I
             BSET  DDRA,%00000011           ; STAR_DIR, PORT_DIR                        N
             BSET  DDRT,%00110000           ; STAR_SPEED, PORT_SPEED                    I
                                            ;                                           T
             JSR   initAD                   ; Initialize ATD converter                  I
                                            ;                                           A
             JSR   initLCD                  ; Initialize the LCD                        L
             JSR   clrLCD                   ; Clear LCD & home cursor                   I
                                            ;                                           Z
             LDX   #msg1                    ; Display msg1                              A
             JSR   putsLCD                  ;     "                                     T
                                            ;                                           I
             LDAA  #$C0                     ; Move LCD cursor to the 2nd row            O
             JSR   cmd2LCD                  ;                                           N

             LDX   #msg2                    ; Display msg2                              |
             JSR   putsLCD                  ;     "                                     |
                                            ;                                           |
             JSR   ENABLE_TOF               ; Jump to TOF initialization ----------------

MAIN         JSR   UPDT_DISPL               ; ------------------------------------------ M
             LDAA  CRNT_STATE               ;                                           A
             JSR   DISPATCHER               ;                                           I
             BRA   MAIN                     ; ------------------------------------------ N

; data section
***********************************************************************
msg1         dc.b  "Battery volt ",0
msg2         dc.b  "State ",0
tab          dc.b  "START  ",0
             dc.b  "FWD    ",0
             dc.b  "REV    ",0
             dc.b  "ALL_STP",0
             dc.b  "FWD_TRN",0
             dc.b  "REV_TRN",0
```

```
; subroutine section
*****************************************************************
DISPATCHER   CMPA  #START                  ; If it's the START state -----------------
             BNE   NOT_START               ;                                         |
             JSR   START_ST                ;  then call START_ST routine             D
             BRA   DISP_EXIT               ;  and exit                               I
                                           ;                                         S
NOT_START    ...                           ;                                         P
             ...                           ;                                         A
                                           ;                                         T
NOT_FWD_TRN  CMPA  #REV_TRN                ; Else if it's the REV_TRN state          C
             BNE   NOT_REV_TRN             ;                                         H
             JSR   REV_TRN_ST              ;  then call REV_TRN_ST routine           E
             BRA   DISP_EXIT               ;  and exit                               R
                                           ;                                         |
NOT_REV_TRN  SWI                           ; Else the CRNT_ST is not defined, so stop |
DISP_EXIT    RTS                           ; Exit from the state dispatcher ----------

*****************************************************************
START_ST     ...
             ...

NO_FWD       NOP                           ; Else
START_EXIT   RTS                           ;  return to the MAIN routine

*****************************************************************
FWD_ST       ...
             ...

NO_FWD_TRN   NOP                           ; Else
FWD_EXIT     RTS                           ;  return to the MAIN routine

*****************************************************************
REV_ST       LDAA  TOF_COUNTER             ; If Tc>Trev then
             CMPA  T_REV                   ;  the robot should make a FWD turn
             BNE   NO_REV_TRN              ;  so
             JSR   INIT_REV_TRN            ;  initialize the REV_TRN state
             MOVB  #REV_TRN,CRNT_STATE     ;  set state to REV_TRN
             BRA   REV_EXIT                ;  and return

NO_REV_TRN   NOP                           ; Else
REV_EXIT     RTS                           ;  return to the MAIN routine

*****************************************************************
ALL_STP_ST   BRSET PORTAD0,$04,NO_START    ; If FWD_BUMP
             BCLR  PTT,%00110000           ;  initialize the START state (both motors off)
             MOVB  #START,CRNT_STATE       ;  set the state to START
             BRA   ALL_STP_EXIT            ;  and return

NO_START     NOP                           ; Else
ALL_STP_EXIT RTS                           ;  return to the MAIN routine

*****************************************************************
FWD_TRN_ST   LDAA  TOF_COUNTER             ; If Tc>Tfwdturn then
             CMPA  T_FWD_TRN               ;  the robot should go FWD
             BNE   NO_FWD_FT               ;  so
             JSR   INIT_FWD                ;  initialize the FWD state
             MOVB  #FWD,CRNT_STATE         ;  set state to FWD
             BRA   FWD_TRN_EXIT            ;  and return

NO_FWD_FT    NOP                           ; Else
FWD_TRN_EXIT RTS                           ;  return to the MAIN routine

*****************************************************************
REV_TRN_ST   LDAA  TOF_COUNTER             ; If Tc>Trevturn then
             CMPA  T_REV_TRN               ;  the robot should go FWD
```

```
             BNE   NO_FWD_RT                 ;  so
             JSR   INIT_FWD                  ;  initialize the FWD state
             MOVB  #FWD,CRNT_STATE           ;  set state to FWD
             BRA   REV_TRN_EXIT              ;  and return

NO_FWD_RT    NOP                             ; Else
REV_TRN_EXIT RTS                             ;  return to the MAIN routine

*******************************************************************
INIT_FWD     BCLR  PORTA,%00000011           ; Set FWD direction for both motors
             BSET  PTT,%00110000             ; Turn on the drive motors
             LDAA  TOF_COUNTER               ; Mark the fwd time Tfwd
             ADDA  #FWD_INT
             STAA  T_FWD
             RTS

*******************************************************************
INIT_REV     BSET  PORTA,%00000011           ; Set REV direction for both motors
             BSET  PTT,%00110000             ; Turn on the drive motors
             LDAA  TOF_COUNTER               ; Mark the fwd time Tfwd
             ADDA  #REV_INT
             STAA  T_REV
             RTS

*******************************************************************
INIT_ALL_STP BCLR  PTT,%00110000             ; Turn off the drive motors
             RTS

*******************************************************************
INIT_FWD_TRN BSET  PORTA,%00000010           ; Set REV dir. for STARBOARD (right) motor
             LDAA  TOF_COUNTER               ; Mark the fwd_turn time Tfwdturn
             ADDA  #FWD_TRN_INT
             STAA  T_FWD_TRN
             RTS

*******************************************************************
INIT_REV_TRN BCLR  PORTA,%00000010           ; Set FWD dir. for STARBOARD (right) motor
             LDAA  TOF_COUNTER               ; Mark the fwd time Tfwd
             ADDA  #REV_TRN_INT
             STAA  T_REV_TRN
             RTS

; utility subroutines
*******************************************************************
initLCD      ...

*******************************************************************
clrLCD       ...

*******************************************************************
del_50us     PSHX                            ; (2 E-clk) Protect the X register
eloop        LDX   #300                      ; (2 E-clk) Initialize the inner loop counter
iloop        NOP                             ; (1 E-clk) No operation
             DBNE  X,iloop                   ; (3 E-clk) If the inner cntr not 0, loop again
             DBNE  Y,eloop                   ; (3 E-clk) If the outer cntr not 0, loop again
             PULX                            ; (3 E-clk) Restore the X register
             RTS                             ; (5 E-clk) Else return

*******************************************************************
cmd2LCD:     ...

*******************************************************************
putsLCD      ...

*******************************************************************
putcLCD      ...
```

```
*****************************************************************
dataMov      ...

*****************************************************************
initAD       ...

*****************************************************************
int2BCD      ...

*****************************************************************
BCD2ASC      ...

*****************************************************************
ENABLE_TOF   ...

*****************************************************************
TOF_ISR      ...

*****************************************************************
*         Update Display (Battery Voltage + Current State)      *
*****************************************************************
UPDT_DISPL   MOVB  #$90,ATDCTL5              ; R-just., uns., sing. conv., mult., ch=0, start
             BRCLR ATDSTAT0,$80,*            ; Wait until the conver. seq. is complete

             LDAA  ATDDR0L                   ; Load the ch0 result - battery volt - into A
             ...                             ; Display the battery voltage

;-------------------------
             LDAA  #$C6                      ; Move LCD cursor to the 2nd row, end of msg2
             JSR   cmd2LCD                   ;

             LDAB  CRNT_STATE                ; Display current state
             LSLB                            ;     "
             LSLB                            ;     "
             LSLB                            ;     "
             LDX   #tab                      ;     "
             ABX                             ;     "
             JSR   putsLCD                   ;     "

             RTS

*****************************************************************
*         Interrupt Vectors                                     *
*****************************************************************
             ORG   $FFFE
             DC.W  Entry                     ; Reset Vector
             ORG   $FFDE
             DC.W  TOF_ISR                   ; Timer Overflow Interrupt Vector
```

5 References

State Machines in Software
Peter Hiscocks
Circuit Cellar: The Computer Applications Journal
Issue 26, April/May 1992, pp 52-60

Microcomputer Technology: The 68HC11
Second Edition
Section 13.3, Sequential Machines
Peter Spasov
Prentice Hall, 1996