Modular Design of Assembly Parking Controller Implemented with Simple VHDL Computer for DE2Bot

Noah Roberts
Zachary Russell
Arjun Sabnis
Justin Vuong
Thomas Wyatt

Georgia Institute of Technology
School of Electrical and Computer Engineering

Submitted
April, 25, 2017

Abstract

This project was intended to automatically park a robot controlled by a simple computer (SCOMP) running on an Altera DE2 board — all driven by a compiled MIF assembly file. Using an infrared remote, a sonar system, and the robot's built-in odometry, a solution had to be designed to allow a user to manually drive the robot to a parking space or have the robot autonomously park when given a certain space number. Sonar functionality was judged to be too complex to implement in the allotted time, so the final solution relied heavily on user judgement and measured distances. In manual mode, the driver can rotate the robot in set increments, drive forward and back, and initiate parking sequences. Function calls are handled through a switch-like code block that interprets IR codes received from the remote. For autonomous mode, the driver specifies the desired space on the number pad, and the robot used the movement functions to navigate to the appropriate space. The project was successful: both modes worked ideally with three successful runs, though one of the attempts was restarted after a misalignment left the robot slightly askew.

Modular Design of Assembly Parking Controller Implemented with Simple VHDL Computer for DE2Bot

Introduction

The problem to be solved is multifaceted: First, the DE2Bot must be manually controlled using the given IR remote. Second, it must be able to park automatically in an arbitrary space in one of several known positions. Finally, both modes must be precise enough to park in a 35 cm by 45 cm space. This setup can be observed in Figure 1.

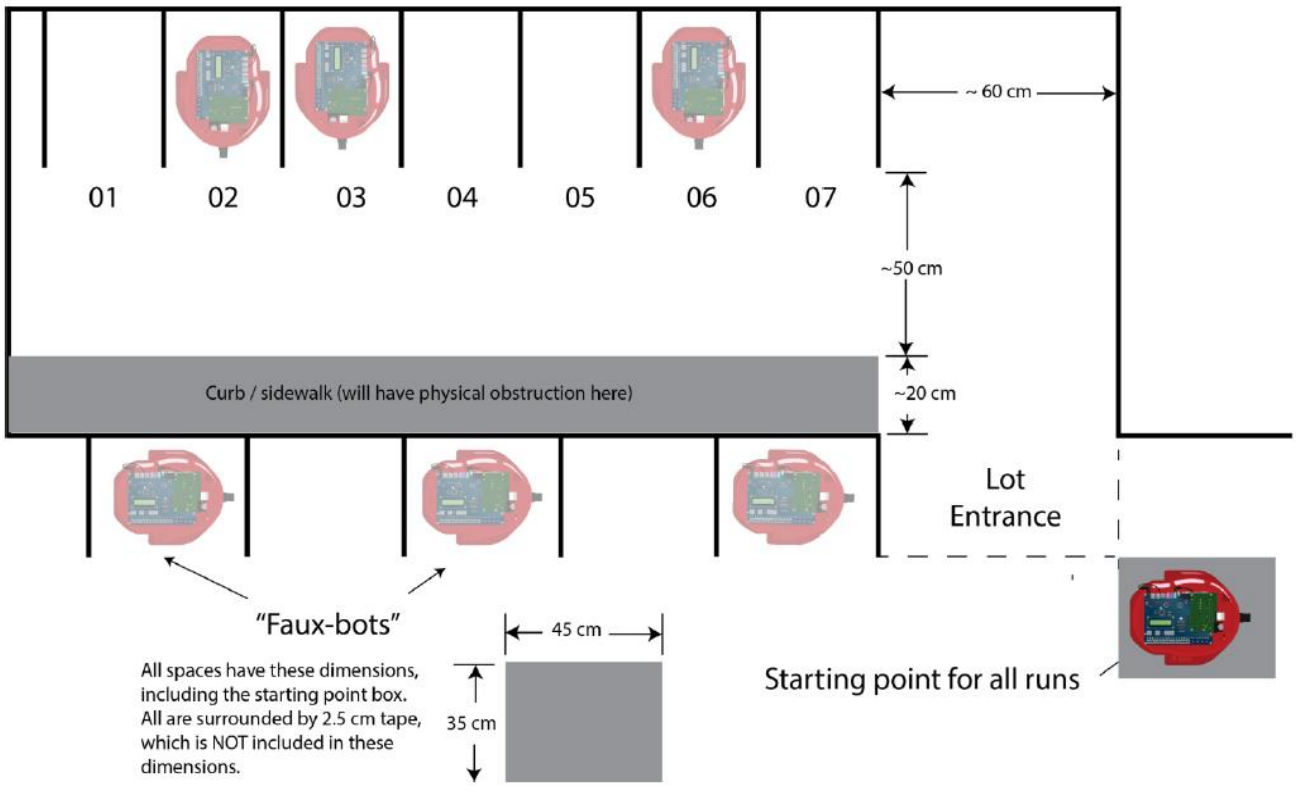

Figure 1. Position of parallel and perpendicular spots within the demo area for the project.

The remote only transmits information at the beginning of each button press and the robot offers minimal feedback (sonar and odometer), both of which proved challenging to accommodate for in the final design. The solution our group decided on was straightforward: start by writing basic control modules - turning

left and right and moving forward and backward set distances - then use those to write the parking subroutines and automatic and manual control modes. This modular approach allowed for testing of each motion function before combining them all into the more complex routines, making it easier to test, debug, and improve our final program. Although we initially wanted to rely on sonar for some of the control, the final solution relied purely on direct driver-control and the robot's built-in odometer. Despite not managing to include all the intended functionality, we still met the requirements for the project.

Technical Approach

Code Structure

Most of the code in the project is structured hierarchically; the driver chooses which functions are executed through a single overarching controller: the Stop state. This section of code serves as a hub for all the actions that can be performed by the driver; they are all called from and all return to this subroutine, keeping the code straightforward and easy to manage. The function, when jumped to, freezes the robot's motors and waits for a button press from the remote. Once it receives one, it compares it to all the assigned buttons on the remote and jumps to the appropriate subroutine when it finds a match. This includes forward, reverse, two different angles of turns, both parking sequences, and buttons to initiate parking in each of the 7 automatic spaces, as seen in Figure 2 on the next page.

In automatic mode, there is one main subroutine that relies on a switch statement block in the Stop state to initialize the values for each parking space. Since most of the automatic parking sequence is identical for each space (excluding the horizontal distance travelled), we wrote it as a single sequence of forward and turn operations and use a preloaded value for the variable-length movement.

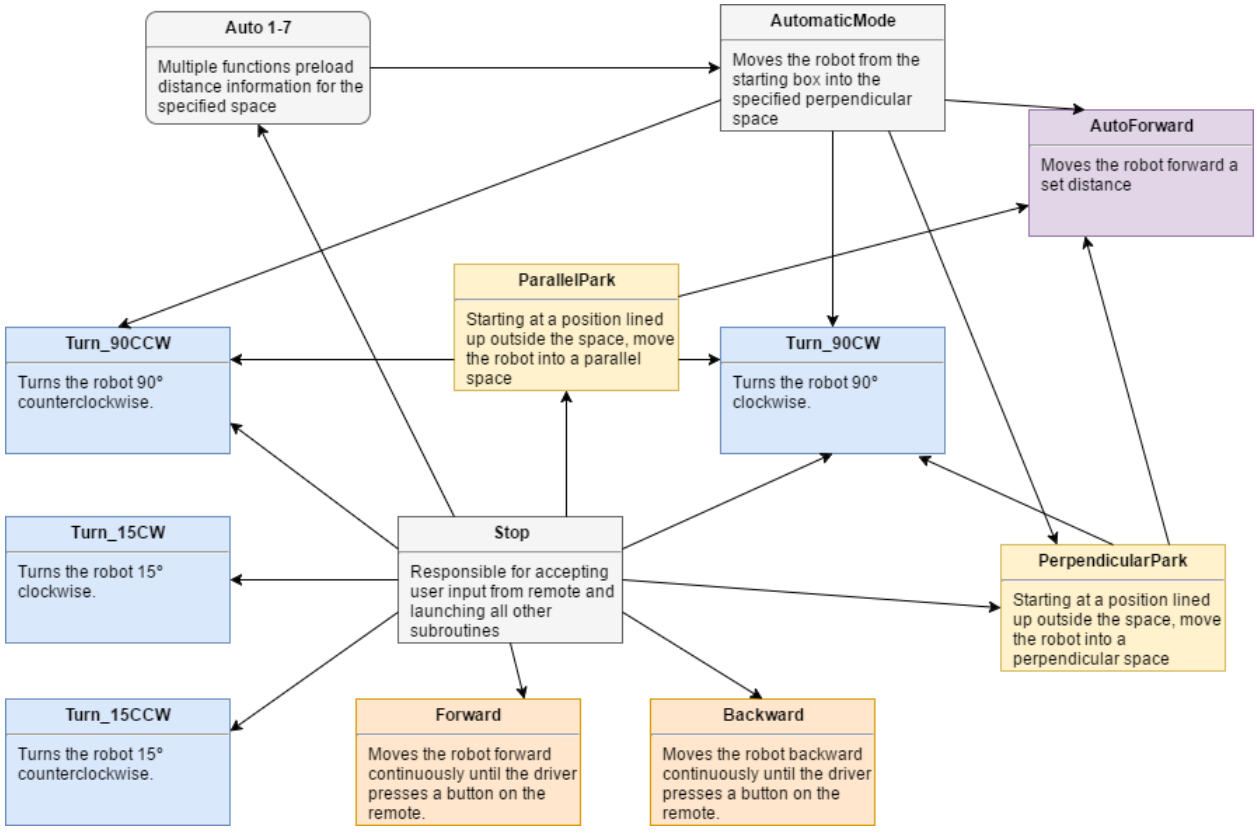

Figure 2. Chart of subroutines implemented in assembly code for both manual and automatic parking sequences. The Stop state is the base state, from where all other states can be called upon.

Code Functionality and Implementation

Discrete-Angle Turns

These functions are used to turn the robot in increments of 15° or 90° in either direction. Since they are free-standing, we can use them in both manual and automatic mode, though we only need the smaller angle turns for realignment during manual control. All four functions follow the same structure, and are differentiated only in the actual angle of the turn. As seen in Figure 3 on the next page, upon entering the subroutine, the odometer values are immediately reset to avoid accidentally turning to an unexpected heading. The desired angle is then stored to DTheta, and the program loops until GetThetaErr reaches a

value less than 2°. At this point, the robot should be within 2° of the desired heading and the function can return control to the caller.

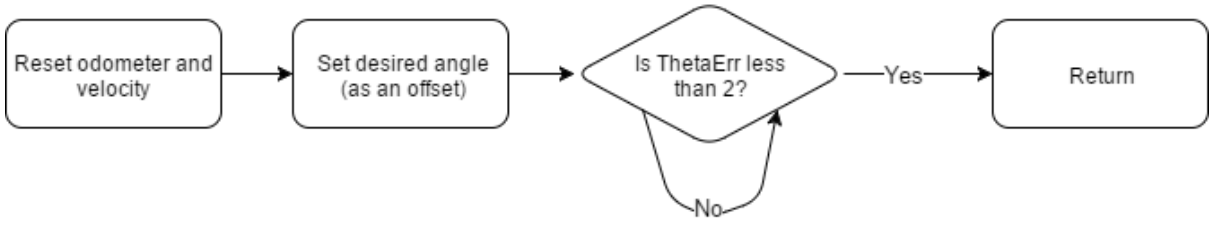

Figure 3. Flowchart demonstrating control and functionality of subroutine used to turn robot.

Discrete-Distance Linear Movement

To limit static instruction count and implement cohesive code, a key objective of the project was to split the code into two sections: drivers and functions (subroutines). As observed in Figure 4, the AutoForward subroutine exists as a simple movement mechanism to be used numerous times in the robot's automatic drivers (AutomaticMode, ParallelPark, PerpendicularPark). Similar to the provided movement controller, values are "passed in" using a set address in memory. The caller specifies AutoFDist (desired discrete travel distance) and AutoFVel (desired velocity in ticks/s). AutoForward uses the DE2Bot odometer (particularly XPOS register value) to check the distance traveled. This compact piece of code clarifies the caller's functionality — compacting code and making it more reader-friendly, which is invaluable during the testing and debugging stage.

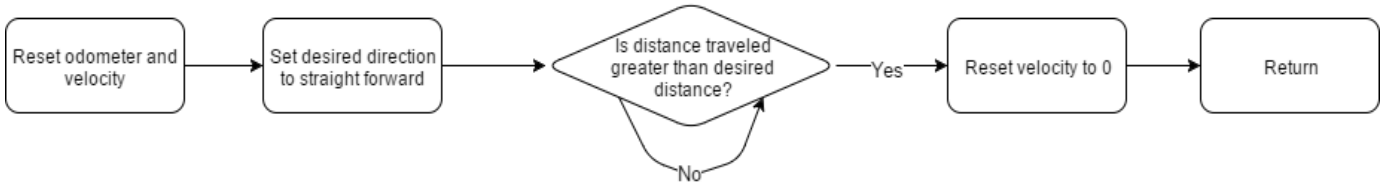

Figure 4. Flowchart demonstrating functionality of subroutine used for discrete distance movement using on-board odometer values and recorded distances.

Continuous Linear Movement

There are two subroutines that fall into this category (Forward and Backward), both used exclusively in manual mode. To make manual control more intuitive, continuous linear movement was chosen instead of movement across a set distance. This enables the robot to travel an unlimited distance before being stopped. However, this meant code could not be recycled from other functions.

These functions have two components to them, as seen in Figure 5. First, upon entering the function, the desired velocity is set, causing the robot to start moving. Second, to control when the robot stops, a loop to waits until the IR subsystem receives an appropriate button-press. In each iteration, the latest IR value is compared to the stop button code. When the two values match, control is returned to the caller.

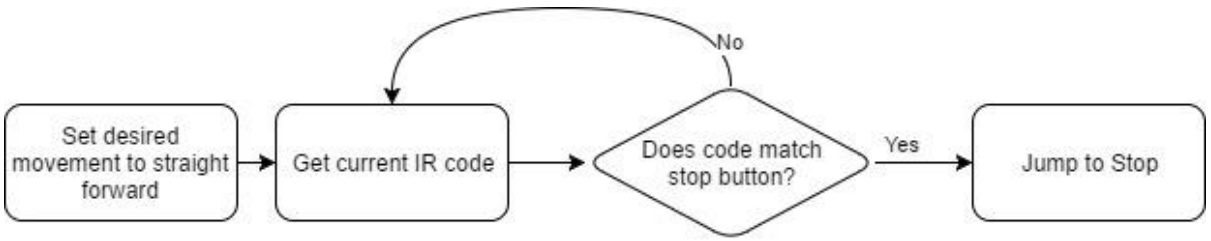

Figure 5. Flowchart demonstrating functionality of subroutine used for continuous forward and backward movement using the IR remote.

Parking sequences

The parking subroutines are both very simple. Using a series of hard-coded movements, they move the robot into a space from a position next to it, making them useful for the final stage of both manual and automatic mode. They both start with a 90° clockwise turn, then pull into the space using AutoForward (290 ticks for parallel, 400 for perpendicular). For the parallel subroutine, the robot must also rotate back to the initial heading, a simple 90° turn counterclockwise. A flowchart demonstrating these subroutines can be seen in Figure 6 on the next page.

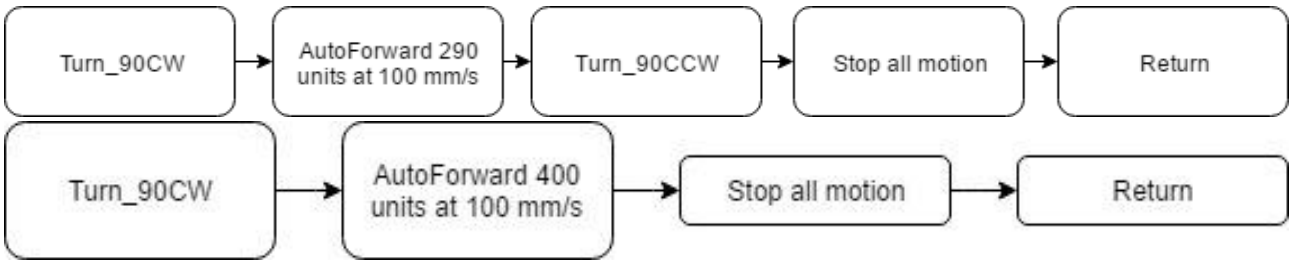

Figure 6. Flowcharts demonstrating functionality of subroutines used for parallel parking (top) and perpendicular parking (bottom) using the AutoForward and Turn_90 subroutines.

Automatic Parking Mechanism

The automatic functionality of the self-parking robot consists of a spot-selection stage and the AutomaticMode subroutine. Figure 7 details the basic operation of the robot's automatic movement.

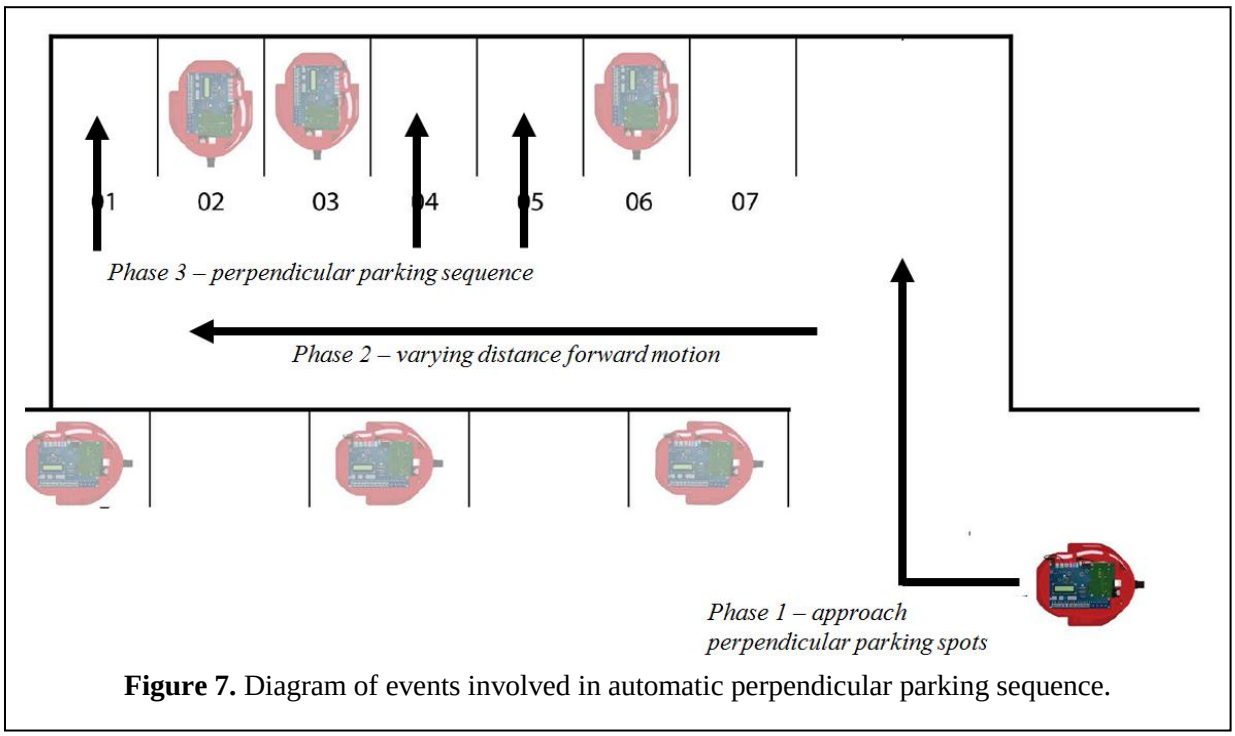

Figure 7. Diagram of events involved in automatic perpendicular parking sequence.

Instead of writing seven separate blocks of code for each perpendicular parking space, AutomaticMode can be used for each when passed the measured spot distance (phase 2 in Figure 7). The provided VHDL simple computer (SCOMP) has 11-bit operands, allowing support up to $\pm 2^{10} = \pm 1024$(base 10). Spots

1-5 require travel distances greater than 1024 ticks, so the desired distances were split into equal segments. This number was stored in memory at SpaceDist, and the factor to multiply by SpaceDist to achieve the total distance was stored at FIterations. When the robot is in the Stop state at the starting position, it waits for IR input from user (details discussed in Code Structure). If the IR register holds the value for remote buttons 1-7, the corresponding values for SpaceDist and FIterations are stored and subroutine AutomaticMode is called (see Figure 8).

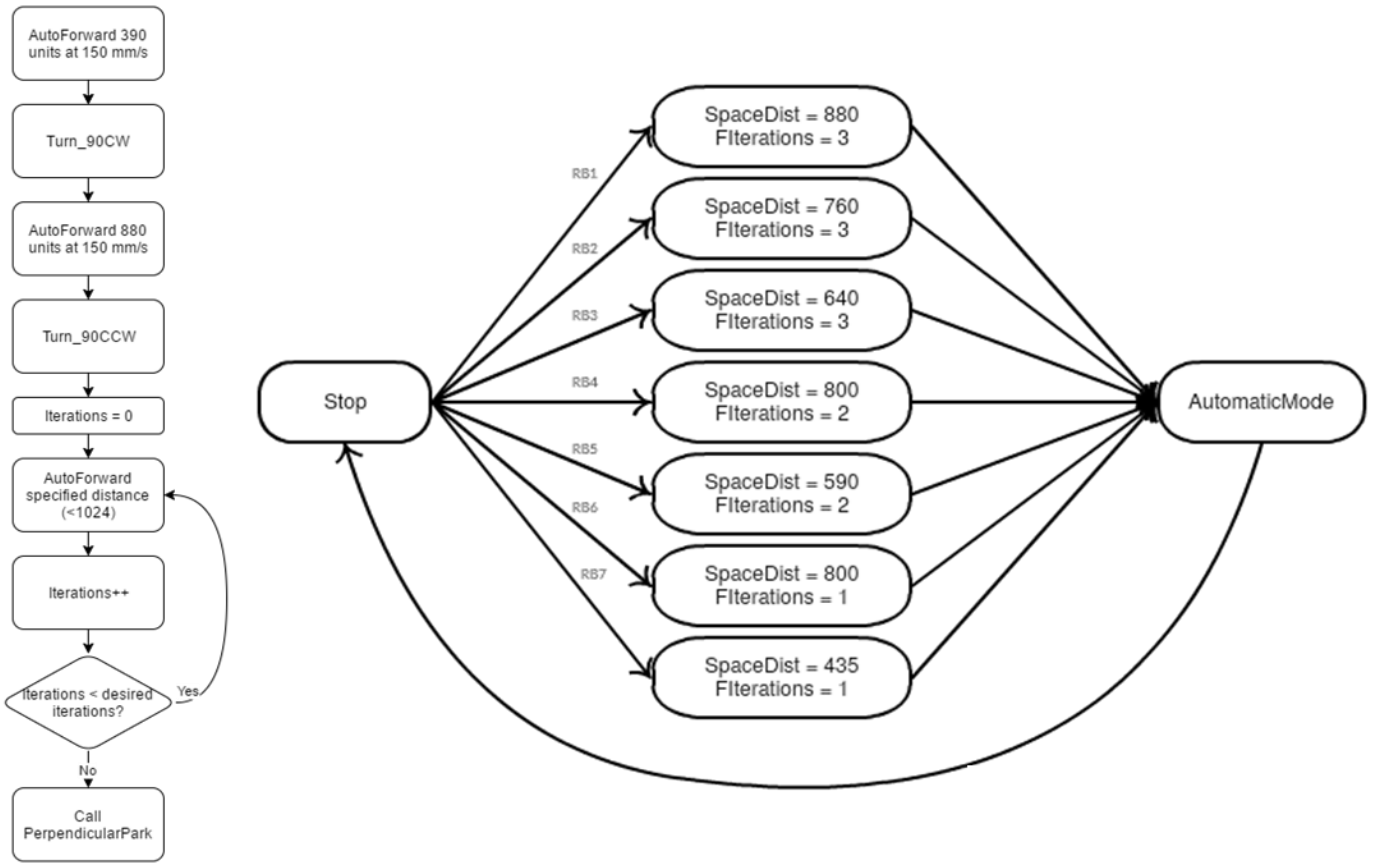

Figure 8. Flowcharts demonstrating functionality of the automatic parking sequence, with the use of variable SpaceDist and FIterations in the assembly code to achieve a modular design.

AutomaticMode itself is relatively simple: it calls existing subroutines for a series of turns and set-distance forward motions using measured distances (tweaked in testing phase). For phase 2, the AutoForward subroutine with parameter SpaceDist stored as AutoFDist (desired forward distance) is

called the number of times specified by FIterations, followed by a call to the same automatic perpendicular park subroutine used in manual control (PerpendicularPark).

Overall Control

The Stop state handles all user input except for canceling the continuous forward/backward operations, in addition to serving as a safe waiting state between motions. The overarching control can be seen in Figure 9. Upon entering the subroutine, the robot's velocity is set to zero and it begins a loop waiting for an IR code. On receiving a code, the program stores it at a memory address to keep the value from being lost from the AC. For each possible action, the button code is subtracted from the received code, and a JZERO operation jumps to the appropriate subroutine if the codes match. By chaining a series of LOAD, SUB and JZERO operations in this fashion, we created a rudimentary switch. If the received code matches none of the assigned buttons, the program loops and waits for another button press.

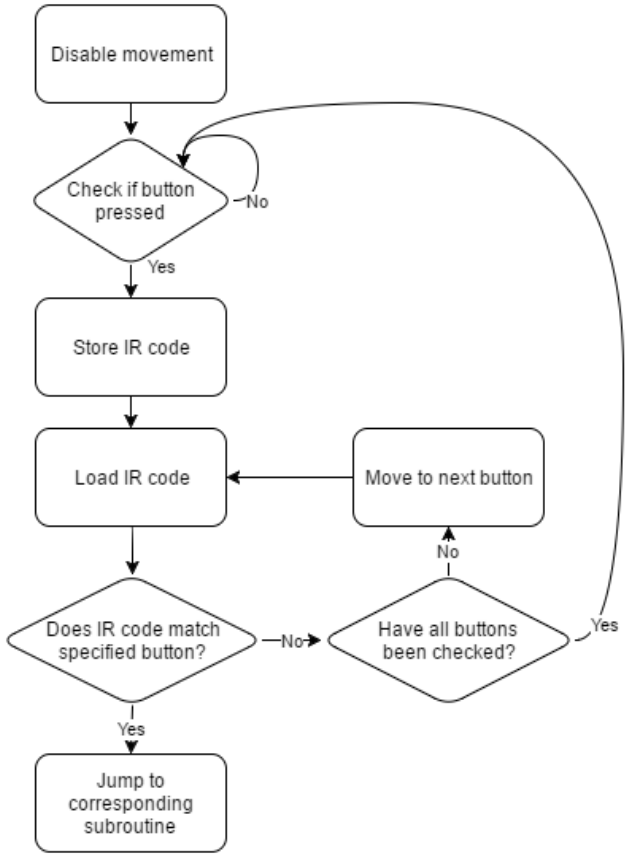

Figure 9. Flowchart demonstrating overall function of the use of IR codes from a remote to control movement functions in the DE2Bot.

Testing/Design Cycle

The testing and design cycle took a results-driven approach. Initially, all project members brainstormed ways to control the robot. Several ideas were combined to form the final product. Designed to be modular, efficient, and simple to write, the final program would be a group of interacting functions to control the robot's movement. As the necessary functions were identified, the best possible implementation of each was discussed and logged.

After determining what functions were necessary to complete the required parking maneuvers, several members were assigned the task of programming individual functions. Dividing development in this fashion ensured that multiple group members understood the project's code and eased the burden of programming the entire system at once. The members assigned to programming each function would meet deadlines set at group meetings so that the code could be tested during lab meetings.

Most code was written outside of lab meetings. Once a function was completed, it was loaded into a temporary MIF for testing. Each function was tested individually, isolated from other components of the project. Testing the functions individually allowed for the isolation of any problems in those specific routines without affecting other parts of the code. Errors in each function were fixed by testing in this manner. For example, an error in the AutoForward function preventing forward motion was uncovered during isolated testing and was quickly fixed after the bug was identified. During this phase, if problems were identified, several members reviewed the code to isolate the issue. Once the cause was determined, it was promptly fixed before being reassembling the program and loading it onto the robot.

Figure XYZ shows the approach used in the project. Code was initially written outside of lab. If it did not assemble in SCASM, any errors preventing assembly were fixed before the resulting MIF was loaded onto the robot. At this stage, the robot's behavior was tested and intently observed. If the robot initially performed as expected, it was tested more thoroughly to ensure the worked properly. If it did not behave as expected, several group members reviewed the code to identify errors. Having several members analyze the same block of code allowed errors to be discovered more quickly. Once the identified error

was fixed, the code was reloaded onto the robot for testing. This process was repeated until the robot behaved as expected, as seen in Figure 10.

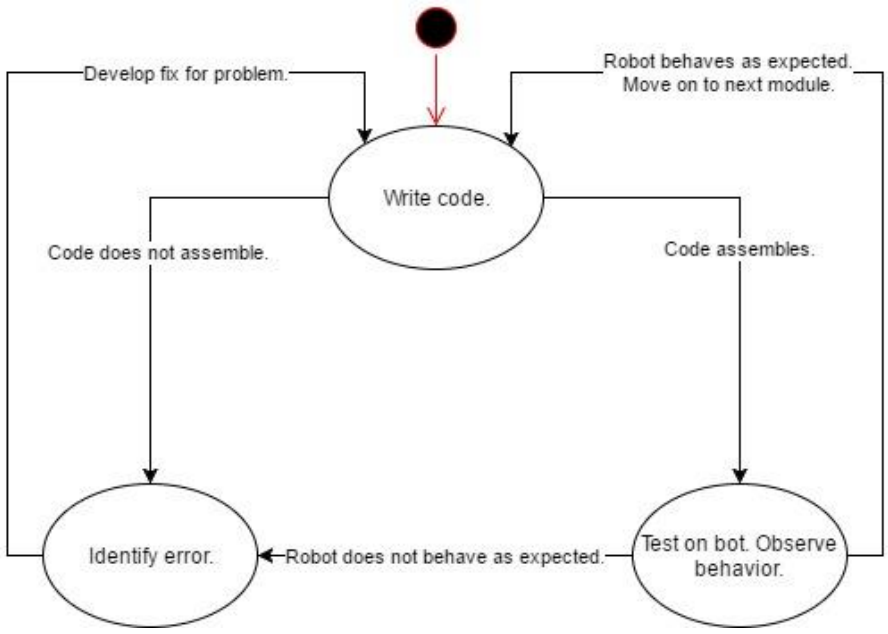

Figure 10. Flowchart demonstrating testing and debugging process undertaken by the group.

Changes From Proposal

It was stated in the proposal that the final design would include a sonar interrupt that checks to see if the parking space is occupied before the parking sequence is initiated. This would be accomplished by reading the value of the sonar sensor that faces the parking spot, and checking if the value received is less than the expected distance for an empty space. The expected distance would be the approximate distance from the robot to the wall. The code for this functionality was written, however there was not enough time to test and debug it, as the focus was on perfecting the parking sequences.

Project Management

Our approach to completing the project in the given time relied mostly on specialization, as seen in the Gantt Chart in Appendix A. With such a small codebase, it wasn't feasible to have all 5 members of the team work on it simultaneously. Instead, after discussing the overall program structure as a group, we

never had more than three people working directly with the code at a time. Between meetings, a few members would work on pseudocode and first-draft assembly ideas, while others would work on written deliverables or brainstorming future aspects of the code. In the lab, part of the group would work on implementing and testing the code written outside of class, while others would work on collecting data (field dimensions, button codes, ongoing issues with the robot). In meetings outside class, everyone contributed to the brainstorming stage and discussed how to approach the work ahead. This strategy allowed everyone to focus on relatively small portions of the project and get familiar with them, while still all being directly involved with and aware of the other aspects.

Results

Testing

The code was tested in two stages. The first stage was testing manual control to ensure that all base-level subroutines were correctly programmed. It allowed for corrections in forward movement and turn control that would be required for the automatic parking subroutines. The second stage was testing the automatic subroutines for all seven possible parking spots to determine any adjustments necessary to achieve the required distances of travel to each spot.

Although the manual controls were eventually prompt and accurate, the testing was not without problems. Minor errors overlooked in the code, such as the substitution of a LOADI instruction with a LOAD instruction, or incorrect reading of the IR addresses passed to the robot, prevented some manual functions from initially working, delaying testing. This first testing stage also alerted the team to three problems that would propagate error in the automatic parking sequence. The first was the inaccuracy of the odometer in determining the current heading. Initially, the robot was programmed with a 5° right and left turns, to make minor manual course corrections. However, the team found that the robot was unresponsive to such minor adjustments. The turn functions were then incremented to 15° right and left

turns, alerting us to a ±5° heading error. The second problem was an occasional overshoot during 90° right and left turns, which would throw the robot off-course over a long distance. The third and most important was noticing that the robot tended to drift in either direction and wobble during continuous forward and backward movement, which was chalked down to error accumulation in the odometer. These led to large error margins in the final position, as seen in Figure 11.

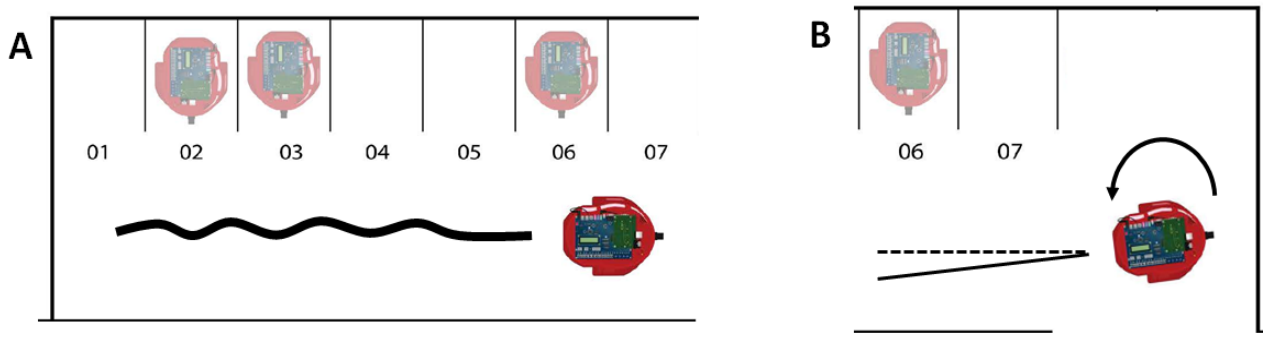

Figure 11. Robot "wobble" (A) and turn overshoots (B) observed in manual control testing that led to further propagation of errors in the automatic parking sequence.

The second stage of testing helped the team determine the required distances to each spot, as well as the distance required to move into each spot once lined up. The hardcoded distances were decreased if the robot overshot a spot, and increased if a robot fell short. Approximately 20 trials were conducted to determine these required distances. An issue to note was that these distances were only accurate if the robot corrected its course and did not drift either left or right; else the final position of the robot would be incorrect. The robot failed to correct its course on turning and drifted either too far left or too far right. Of 15 trials conducted to test the automatic sequence to the first spot, 11 of them failed, giving a 26.7% success rate for the sequence. The team attempted to correct this wobble and the turn overshoots by decreasing the maximum allowed motor velocity down to 150 units. This allowed the robot more time to correct its heading to the desired one during turns, and the reduced velocity prevented the wobbling seen during forward movement. The success rate for testing with the first spot increased to 6 successful tries out of 10, a 222% increase over the previous success rate. The unsuccessful tries were due to robot drift,

the direction of which was dependent on the initial placement and orientation of the robot as seen in Figure 12. This was an error that could not always be avoided, and was an aspect of the automatic sequence prone to human error.

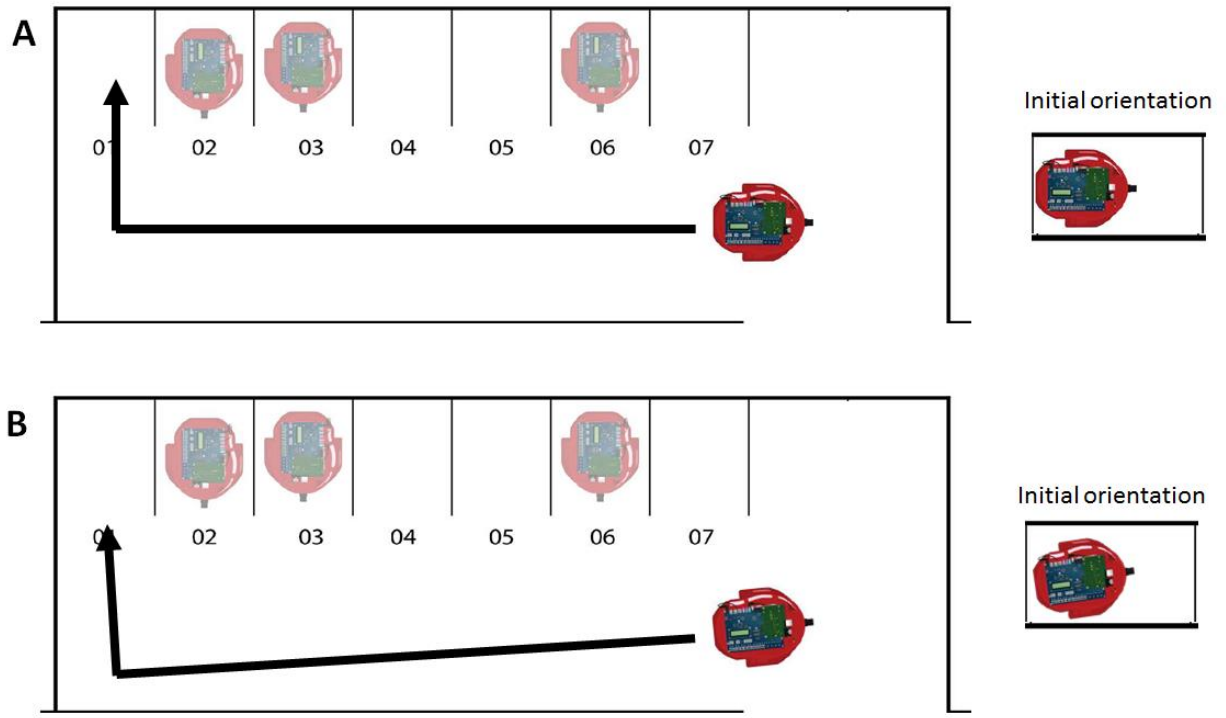

Figure 12. Diagram of how initial orientation of robot affected final position. If oriented straight (A), the robot reached its designated spot. If oriented askew (B), the robot would fail to reach with the same distances.

Demo

The team had to be mindful of the error that the initial robot orientation and placement could propagate during the automatic sequence, and as such, the pre-demo testing was focused on making sure the automatic sequence worked. However, the robot drift once again caused significant errors in the final position during tests. Another robot was selected and tested with the same code, and showed very little error, leading the team to the conclusion that the error was hardware-dependent, rather than software-dependent.

The results of the demo can be seen in Figure X. The demo had three perfect runs (one for each kind of parking), and one run leading to a slight collision with the curb, which was a result of human error (poor judgment of distance from the spot). The total time taken was approximately 4 minutes and 30 seconds, with an average time of approximately 41.5 seconds per run. All the code ran as was expected, and the robot displayed minimal drift during the automatic parking run, which could have been due to either correct initial placement of the robot, or better hardware in one robot than another.

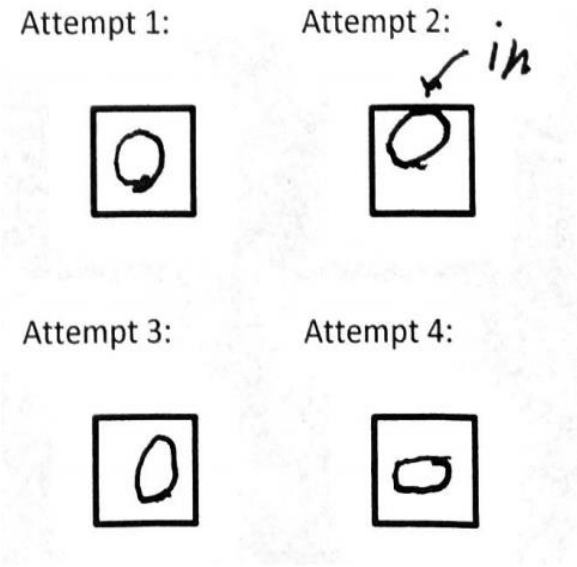

Figure 13. Demo run results: automatic perpendicular parking (Attempt 1), parallel parking (Attempts 2 & 4), perpendicular parking (Attempt 3). The circles represent the position of the robot at the end of each sequence. Attempt 4 replaced the partially unsuccessful Attempt 2 (slight collision with curb)

Conclusions

Both the manual and automatic functions worked as expected and were extremely precise in the final product created. The manual controls were highly responsive and allowed rapid manipulation of the robot. The automatic sequences worked correctly most times, unless the original orientation and placement of the robot was incorrect. Flawless automatic runs were possible because the code was designed such that if the robot overshot or undershot the target, only a few numbers needed to be changed. This allowed for a

quick turnaround during the debugging phase, where the group would need to reprogram the robot before every test. One weakness in the design of the code was that the performance of the robot degraded when using a different robot with the same code. This was evident when, on the demonstration day, a different robot than the one that was used for debugging was assigned to our group. The new robot veered off course and crashed into obstacles where the previous one had shown no major issues. However, with the same code programmed into the robot used for debugging, everything worked as expected.

This difference in performance between robots suggests an improvement for future implementation of automatic parking. A subroutine that corrects the course of the robot if it veers off course can be added to the code structure. Another improvement could be to measure distances with the sonar sensor instead of hard-coding the values, so that the robot can park in different parking lot arenas, and not just the one in the project.

A weakness in the design was the absence of a course correction mechanism. Hence, to make sure the robot did not veer off course, the speed had to be reduced considerably. This meant more time was required for each parking trial, reducing efficiency of the sequence.

If the project were to be repeated, one thing we would do differently is to regularly check for common typos. Some of our major setbacks were because of simple typos in the code; misuse of the OUT and LOAD instructions, and mistaking a ‘6’ for a ‘B’ on the seven-segment displays on the robot set us back a few valuable hours of lab time.

Self-parking technology is currently being implemented in automobiles. Many cars have the option to include a parking assistant that only requires the driver to apply the brakes, while the car turns the wheel automatically. In the future, parking may become obsolete when one can exit one’s car at a destination, and have the car drive itself to the parking spot.

Appendix A: Gantt Chart for Project Management Plan

Task	Duration	Start	Finish
Project introduction	15 days	Monday 3/16/2017	Thursday 3/30/2017
Implement initial test functions - forward and backward arcs, in-place turns	1 day	Monday 3/16/2017	Monday 3/16/2017
Create Github repository, Google Drive for central file sharing	1 day	Tuesday 3/17/2017	Tuesday 3/17/2017
Go over DE2Bot Manual, ASM file and flesh out brainstorming ideas	7 days	Monday 3/20/2017	Sunday 3/26/2017
Discuss and decide most effective implementation of parking sequence	4 days	Monday 3/27/2017	Thursday 3/30/2017
Compile list of required functions, IR button codes, test area dimensions	1 day	Thursday 3/30/2017	Thursday 3/30/2017
Function Implementations	15 days	Thursday 3/30/2017	Thursday 4/13/2017
Write pseudocode for required manual and automatic parking functions	4 days	Thursday 3/30/2017	Monday 4/3/2017
Code manual driving controls into ASM file	4 days	Monday 4/3/2017	Thursday 4/6/2017
Code odometry-based automatic parking functions into ASM file	7 days	Thursday 4/6/2017	Thursday 4/13/2017
Debug manual controls	1 day	Thursday 4/6/2017	Thursday 4/6/2017
Implement sonar interrupts for parking failsafe	4 days	Monday 4/10/2017	Thursday 4/13/2017
Debug automatic sequence, sonar interrupts	1 day	Thursday 4/13/2017	Thursday 4/13/2017
Testing and Execution	8 days	Thursday 4/13/2017	Thursday 4/20/2017
Test and confirm full functionality of manual and automatic sequences	7 days	Thursday 4/13/2017	Wednesday 4/19/2017
Project final demo	1 day	Thursday 4/20/2017	Thursday 4/20/2017

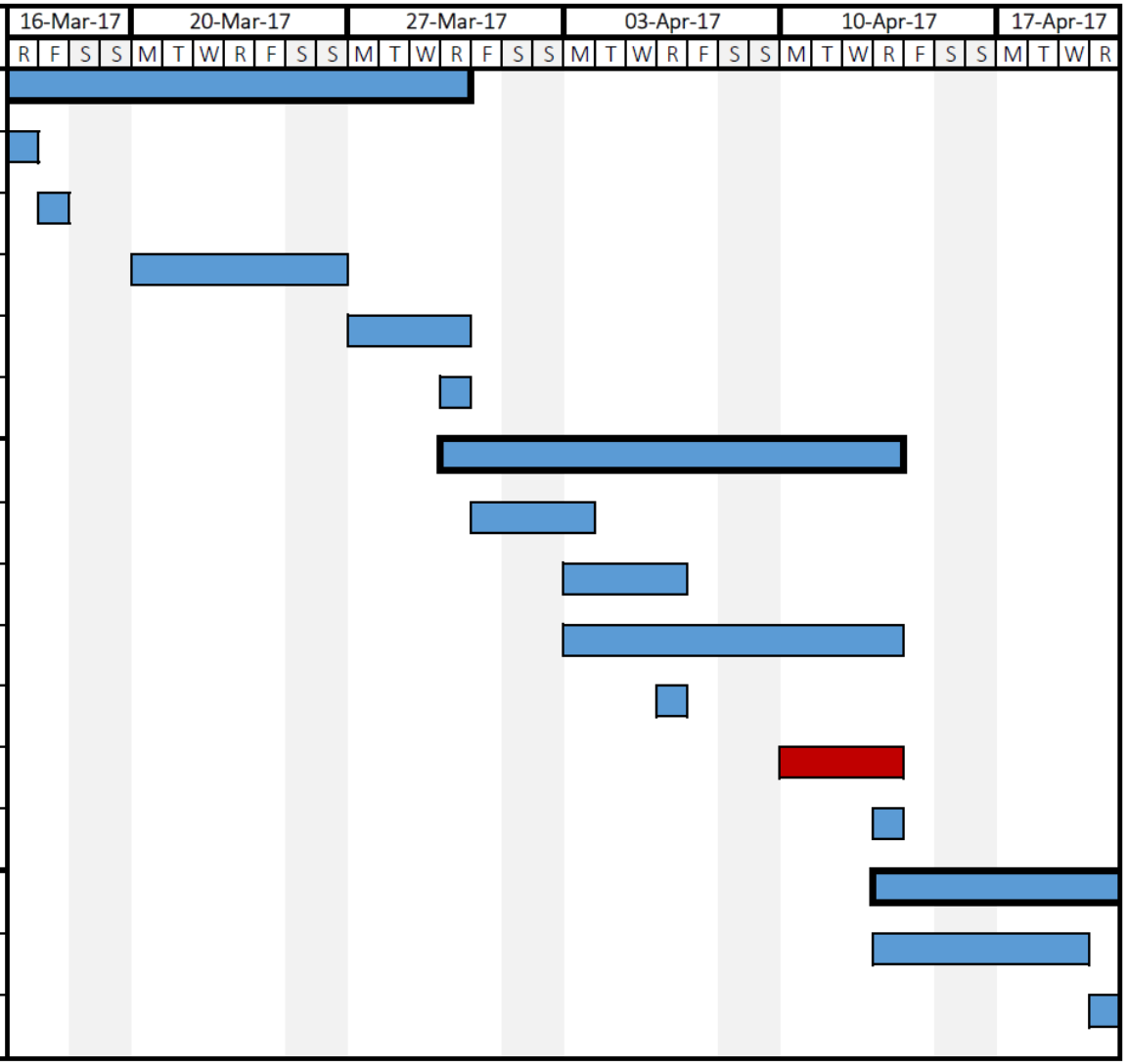

Figure 1. Gantt Chart displaying timeline of major events within the management plan. Completed tasks in the timeline are shaded blue, while forecasted but incomplete tasks are shaded red.

Appendix B: Assembly Code for Subroutine Library

```
;===================================================================
; GTPTS Subroutine & Function Library
; For DE2Bot Parking Controller
; Written by:
;   Noah Roberts
;   Zachary Russell
;   Arjun Sabnis
;   Justin Vuong
;   Thomas Wyatt
;===================================================================

; FUNCTIONS FOR MANUAL PARKING
; NOTE: JUMPS USED TO COMPACT MANUAL DRIVER
;       CODE IN STOP STATE.
; ===============[MANUAL RIGHT TURN]===============
;   TURNS ROBOT 90 DEGREES CLOCKWISE.
Turn_90CW:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; desired forward velocity
    OUT    LVELCMD     ; Stop motors
    OUT    RVELCMD
    LOADI  270
    STORE  DTheta      ; desired heading
 subTurn_90CW:
    CALL   GetThetaErr
    CALL   Abs
    ADDI   -2
    JPOS   subTurn_90CW
    ;RETURN             ; return to caller
    JUMP Stop

; ===============[MANUAL LEFT TURN]===============
;   TURNS ROBOT 90 DEGREES COUNTER-CLOCKWISE.
Turn_90CCW:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; desired forward velocity
    OUT    LVELCMD     ; Stop motors
    OUT    RVELCMD
    LOADI  90
    STORE  DTheta      ; desired heading
subTurn_90CCW:
    CALL   GetThetaErr
    CALL   Abs
    ADDI   -2
    JPOS   subTurn_90CCW
    ;RETURN             ; return to caller
    JUMP Stop

; ==============[MANUAL INCREMENTAL RIGHT TURN]==============
;   TURNS ROBOT 5 DEGREES CLOCKWISE.
```

```
Turn_5CW:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; desired forward velocity
    OUT    LVELCMD     ; Stop motors
    OUT    RVELCMD
    LOADI  345
    STORE  DTheta      ; desired heading
 subTurn_5CW:
    CALL   GetThetaErr
    CALL   Abs
    ADDI   -2
    JPOS   subTurn_5CW
    ;RETURN             ; return to caller
    JUMP Stop

; ==============[MANUAL INCREMENTAL LEFT TURN]===============
;   TURNS ROBOT 5 DEGREES COUNTER-CLOCKWISE.
Turn_5CCW:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; desired forward velocity
    OUT    LVELCMD     ; Stop motors
    OUT    RVELCMD
    LOADI  15
    STORE  DTheta      ; desired heading
subTurn_5CCW:
    CALL   GetThetaErr
    CALL   Abs
    ADDI   -2
    JPOS   subTurn_5CCW
    ;RETURN             ; return to caller
    JUMP Stop

; ==============[CONTINUOUS FORWARD MOVEMENT]===============
;   FUNCTION TO MOVE ROBOT FORWARD AT VELOCITY = 200 mm/s
;   UNTIL USER INPUTS STOP COMMAND.
;   PLAY: FORWARD      0: STOP
Forward:
    OUT RESETPOS       ; Reset odometry
    LOADI  200         ; 200 mm/s
    STORE  Dvel        ; Desired forward velocity
    LOAD   Zero        ; 0 degree turn (straight)
    STORE  DTheta      ; Desired heading
    OUT IR_LO
WaitStopF:             ; Wait for stop input
    CALL   IRDisp      ; Display IR code
    IN     IR_LO       ; Load most-significant half of IR word (address)
    SUB    RBstop      ; Subtract button 0 hex value
    JNEG   WaitStopF
    JPOS   WaitStopF
    LOAD Zero
```

```
    STORE DVel
    JUMP Stop ; RETURN TO CALLER

; ==============[CONTINUOUS BACKWARD MOVEMENT]===============
;   FUNCTION TO MOVE ROBOT BACKWARD AT VELOCITY = 100 mm/s
;   UNTIL USER INPUTS STOP COMMAND.
;   PLAY: FORWARD      0: STOP
Backward:
    OUT RESETPOS       ; Reset odometry
    LOAD   RSlow       ; -100 mm/s
    STORE  Dvel        ; Desired forward velocity
    LOAD   Zero        ; 0 degree turn (straight)
    STORE  DTheta      ; Desired heading
    OUT IR_LO
WaitStopR:             ; Wait for stop input
    CALL   IRDisp      ; Display IR code
    IN     IR_LO       ; Load most-significant half of IR word (address)
    SUB    RBstop          ; Subtract button 0 hex value
    JNEG   WaitStopR
    JPOS   WaitStopR
    LOAD Zero
    STORE DVel
    JUMP Stop ; RETURN TO CALLER


;===============================================================
; SUBROUTINES FOR AUTO PARKING
; ==============[AUTO RIGHT TURN]===============
;   SUBROUTINE TO TURN ROBOT 90 DEGREES CLOCKWISE
Turn_90CW_AUTO:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; Desired forward velocity
    LOADI  270
    STORE  DTheta      ; Desired heading
 subTurn_90CW_AUTO:
    CALL   GetThetaErr
    CALL   Abs
    ADDI   -1
    JPOS   subTurn_90CW_AUTO
    RETURN             ; Return to caller

; ==============[AUTO LEFT TURN]===============
;   SUBROUTINE TO TURN ROBOT 90 DEGREES COUNTER-CLOCKWISE
Turn_90CCW_AUTO:
    LOAD Zero
    OUT RESETPOS
    STORE  DVel        ; Desired forward velocity
    LOADI  90
    STORE  DTheta      ; Desired heading
subTurn_90CCW_AUTO:
    CALL   GetThetaErr
```

```
    CALL   Abs
    ADDI   -1
    JPOS   subTurn_90CCW_AUTO
    RETURN             ; Return to caller

; ===============[AUTOMATIC FORWARD]===============
;   MOVES BOT A CERTAIN DISTANCE FORWARD (PASSED IN)
;   DESIRED DISTANCE STORED IN: AutoFdist
;   DESIRED VELOCITY STORED IN: AutoFvel
AutoFvel:     DW 0
AutoFDist:    DW 0
AutoForward:
    LOADI 0
    OUT RESETPOS    ; Reset odometry
    LOAD AutoFvel
    STORE DVel      ; Desired velocity
    LOADI 0
    STORE Dtheta    ; Desired heading
AutoFLoop:

    IN XPOS         ; Load distance bot traveled
    SUB AutoFdist   ; Desired distance to travel
    JNEG AutoFLoop  ; Loop condition
    LOADI 0
    STORE DVel
    RETURN

; ===============[PARALLEL]===============
;   BOT POSITIONED A DISTANCE [X] FROM CANDIDATE SPOT.
;   PARKING STEPS:
;       1. TURN 90 DEGREES RIGHT
;       2. MOVE FORWARD 290 TICKS
;       3. TURN 90 DEGREES LEFT
ParallelPark:
    ; STEP 1
    CALL Turn_90CW_AUTO

    ; STEP 2
    LOADI 290           ; Distance to travel into space
    STORE AutoFdist     ; Desired distance
    LOAD FSlow          ; 100 mm/s
    STORE AutoFvel
    CALL AutoForward

    ; STEP 3
    CALL Turn_90CCW_AUTO
    RETURN

; ===============[PERPENDICULAR]===============
;   BOT POSITIONED A DISTANCE [X] FROM CANDIDATE SPOT.
;   PARKING STEPS:
;       1. TURN 90 DEGREES RIGHT
```

```
;       2. MOVE FORWARD 400 TICKS
PerpendicularPark:
    ; STEP 1
    CALL Turn_90CW_AUTO

    ; STEP 2
    LOADI 400           ; Distance to travel into space
    STORE AutoFdist     ; Desired distance
    LOAD FSlow          ; 100 mm/s
    STORE AutoFvel
    CALL AutoForward
    LOADI 0
    STORE DVel
    OUT    LVELCMD     ; Stop motors
    OUT    RVELCMD
    RETURN

; ==============[AUTOMATIC PARKING]==============
;   BOT STARTS IN SET POSITION. MUST TRAVEL TO DESIGNATED
;   PERPENDICULAR SPOT AND PARK. ONLY USER INPUT IS
;   FOR SPOT (BUTTONS 1-7)
;   PARKING STEPS:
;       1. TRAVEL 376 FORWARD
;       2. TURN RIGHT
;       3. TRAVEL 860 FORWARD
;       4. TURN LEFT
;       5. PROCEED SPECIFIED DISTANCE TO SPACE
;       6. PARK IN SPACE
;   SpaceDist ==> PASSED IN DIST TO SPACE.
;                 IF GREATER THAN 1023, SPLIT
;                 INTO EQUAL SEGMENTS.
;   FIterations ==> REQUIRED ITERATIONS TO TRAVEL
;                   TOTAL DIST. PASSED IN.
SpaceDist:    DW 0
FIterations:  DW 0
AutomaticMode:
    ; STEP 1
    LOADI 390           ; Distance to travel forward out of start
    STORE AutoFdist     ; Desired distance
    LOADI 150           ; 150 mm/s
    STORE AutoFvel
    CALL AutoForward

    ; STEP 2
    CALL Turn_90CW_AUTO

    ; STEP 3
    LOADI 880           ; Distance to travel toward row of perp spaces
    STORE AutoFdist     ; Desired distance
    LOADI 150           ; 150 mm/s
    STORE AutoFvel
    CALL AutoForward
```

```
    ; STEP 4
    CALL Turn_90CCW_AUTO

    ; STEP 5
    LOAD Zero
    STORE Temp
AutoModeLoop:
    LOAD SpaceDist      ; Segmented distance to travel to designated space
                        ; [Value determined by user button press]
    STORE AutoFdist     ; Desired distance
    LOADI 150           ; 150 mm/s
    STORE AutoFVel      ; Desired velocity
    CALL AutoForward
    LOAD Temp
    ADDI 1
    STORE Temp          ; Temp++
    SUB FIterations
    JNEG AutoModeLoop   ; Loop condition => while(Temp<FIterations)

    ; STEP 6
    CALL PerpendicularPark
    RETURN
```

Appendix C: Measurements and Distances Required for Automatic Perpendicular Parking

Table 1
Pseudocode and Measured Distances for Automatic Sequence

	Pseudocode	**XPOS**
Main Sequence	Position at bottom left corner of box, wait for IR	-
	Reset odometer	0
	Move left 458.79 mm	376
	Stop, turn 90 CW, reset odometer	0
	Move up 941.4 mm	860
	Stop, turn 90 CCW, reset odometer	0
Spot 7	Move left 564.34 mm	401
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 6	Move left 911.42 mm	732
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 5	Move left 1326.33 mm	1127
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 4	Move left 1700.23 mm	1483
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 3	Move left 2076.51 mm	1842
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 2	Move left 2454.31 mm	2200
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364
Spot 1	Move left 2830.53 mm	2560
	Stop, turn 90 CW, reset odometer	0
	Move up 446.10 mm, stop	364

Appendix D: IR Button Address Table and Button Mapping

TABLE 1
IR ADDRESSES RECEIVED BY DE2BOT AND BUTTON MAPPING

Button	Hex Address	Subroutine
OFF/ON	8C7300FF	-
VOL UP	20DF40BF	-
VOL DOWN	20DFC03F	-
PREV CHAN	8C7342BD	Parallel park
MUTE	20DF906F	Perpendicular park
CHAN UP	8C73807F	-
CHAN DOWN	8C7340BF	-
1	8C7320DF	Park in space 1
2	8C73A05F	Park in space 2
3	8C73609F	Park in space 3
4	8C73E01F	Park in space 4
5	8C7330CF	Park in space 5
6	8C73604F	Park in space 6
7	8C73708F	Park in space 7
8	8C73F00F	-
9	8C7338C7	-
0	8C736847	RT90 (90° CW)
ENTER	8C733AC5	LT90 (90° CCW)
TV/VCR	8C73FF00	-
REW	8C734867	LT15 (15° CCW)
PLAY	8C7328D7	FW (Forward)
FF	8C73C837	RT15 (10° CCW)
REC	8C73A857	-
PAUSE	8C738877	BW (Reverse)
STOP	8C7308F7	Stop

Appendix E: Brainstorming and Logbook Sheets