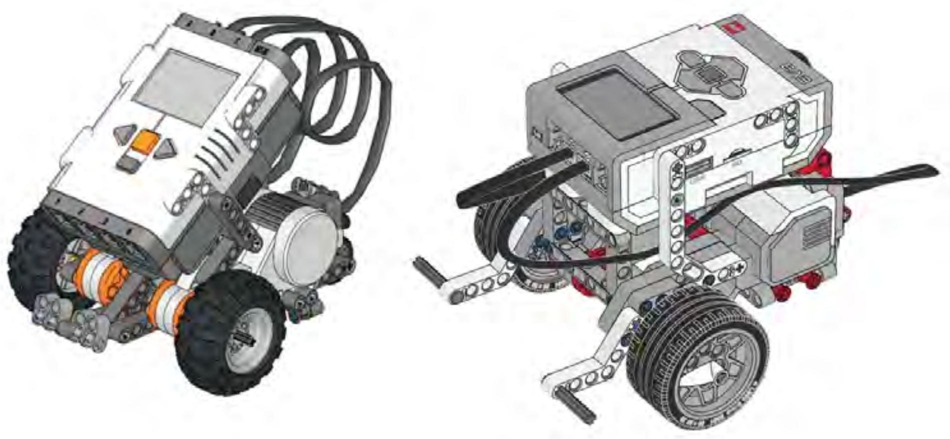
Learning Robot Programming with Lego Mindstorms for the Absolute Beginner

Harry H. Cheng
UC Davis Center for Integrated Computing and STEM Education (C-STEM)
University of California-Davis
http://c-stem.ucdavis.edu



February 16, 2018

Table of Contents

Preface xi

1 Introduction **1**

1.1 Introduction 1
1.1.1 Summary 2
1.1.2 Terminology 2
1.1.3 Exercises 3
1.2 C-STEM Studio 3
1.2.1 Summary 4
1.2.2 Terminology 4
1.2.3 Exercises 4
1.3 RoboPlay Competition 4
1.3.1 Summary 5
1.3.2 Terminology 5
1.3.3 Exercises 5
1.4 Major Features of Mindstorms and Ch Mindstorms Package 6
1.4.1 Summary 7
1.4.2 Terminology 7
1.4.3 Exercises 7

2 Controlling a Mindstorms Using the Motion Control Panel **8**

2.1 Configuring Mindstorms NXT/EV3 for Wireless Control 8
2.1.1 Turning a Mindstorms Brick On and 0ff 8
2.1.2 Check Bluetooth Support on Computers 9
2.1.3 Enable Bluetooth on NXTs/EV3s 10
2.1.4 Launch “Ch Mindstorms Controller (CMC)” 11
2.1.5 Configure Robots 13
2.1.6 Connect a Robot from Computer 17
2.1.7 Summary 17
2.1.8 Terminology 17

2.1.9 Exercises . . . 17
2.2 Control a Mindstorms Using the Motion Control Panel . . . 18
2.3 Control Multiple Mindstorms . . . 19
2.3.1 Summary . . . 20
2.3.2 Terminology . . . 20
2.3.3 Exercises . . . 20

3 Getting Started With Programming Mindstorms **22**

3.1 Get Started with Ch for Computer Programming . . . 22
3.1.1 Getting Started with ChIDE . . . 22
3.1.2 Copy Code in Curriculum . . . 23
3.1.3 The First Ch Program . . . 23
3.1.4 Opening Programs in ChIDE from Windows Explorer . . . 25
3.1.5 Editing Programs . . . 26
3.1.6 Running Programs and Stopping Their Execution . . . 26
3.1.7 Output from Execution of Programs . . . 26
3.1.8 Newline Character . . . 27
3.1.9 Copying a Program to Another Program in C-STEM Studio . . . 27
3.1.10 Correcting Errors in Programs . . . 28
3.1.11 Summary . . . 28
3.1.12 Terminology . . . 29
3.1.13 Exercises . . . 29
3.2 Drive Forward and Backward by Angle Relative to its Current Joint Position . . . 30
3.2.1 Summary . . . 32
3.2.2 Terminology . . . 32
3.2.3 Exercises . . . 32
3.3 Control a Robot in Debug Mode . . . 33
3.3.1 Summary . . . 33
3.3.2 Terminology . . . 33
3.3.3 Exercises . . . 33
3.4 Drive a Distance for a Two-Wheel Robot . . . 34
3.4.1 Summary . . . 35
3.4.2 Terminology . . . 35
3.4.3 Exercises . . . 35
3.5 Play Melody . . . 35
3.5.1 Summary . . . 37
3.5.2 Terminology . . . 37
3.5.3 Exercises . . . 37

4 Robot Simulation with RoboSim **38**

4.1 RoboSim GUI . . . 38
4.1.1 Platform . . . 39
4.1.2 Units . . . 40
4.1.3 Tracing . . . 40
4.1.4 Grid Configuration . . . 40
4.1.5 Individual Robot Configuration . . . 41
4.1.6 Summary . . . 43

4.1.7 Terminology 43
4.1.8 Exercises 44
4.2 Run a Ch Program with RoboSim 44
4.2.1 Summary 46
4.2.2 Terminology 46
4.2.3 Exercises 46
4.3 Interact with a RoboSim Scene 46
4.3.1 Keyboard Input 47
4.3.2 Mouse Input 48
4.3.3 Summary 49
4.3.4 Terminology 50
4.3.5 Exercises 50

5 Using Variables and Generating Robot Programs Using RoboBlockly 51

5.1 Use Variables and Drive a Distance 51
5.1.1 Declaration of Variables and Data Type **double** for Decimals 52
5.1.2 Initialization 53
5.1.3 Data Type **int** for Integers 53
5.1.4 Summary 55
5.1.5 Terminology 55
5.1.6 Exercises 55
5.2 Turn Left and Turn Right 56
5.2.1 Summary 57
5.2.2 Terminology 57
5.2.3 Exercises 57
5.3 Robot Simulation and Generating Robot Programs with RoboBlockly 57
5.3.1 Summary 60
5.3.2 Terminology 60
5.3.3 Exercises 61

6 Interacting with a Mindstorms at Runtime through Variables and Input/Output Functions 62

6.1 The Output Function **printf**() 62
6.1.1 Precision of Decimal Numbers 64
6.1.2 Summary 66
6.1.3 Terminology 66
6.1.4 Exercises 66
6.2 Input into Programs Using Function **scanf**() 67
6.2.1 Summary 69
6.2.2 Terminology 69
6.2.3 Exercises 70
6.3 Number Line for Distances 70
6.3.1 Summary 73
6.3.2 Terminology 74
6.3.3 Exercises 74

7 Writing Programs to Control a Group of Mindstorms to Perform Identical Tasks 75

7.1 Control a Group of Mindstorms with Identical Movements . . . 75
7.1.1 Summary . . . 77
7.1.2 Terminology . . . 77
7.1.3 Exercises . . . 77
7.2 Control an Array of Mindstorms with Identical Movements . . . 78
7.2.1 Summary . . . 80
7.2.2 Terminology . . . 81
7.2.3 Exercises . . . 81

8 Controlling a Mindstorms as a Two-Wheel Robot 82

8.1 Move a Two-Wheel Robot with the Specified Distance . . . 83
8.1.1 Move a Two-Wheel Robot with the Specified Speed, Joint Angles, and Distance . . . 83
8.1.2 Control a Mindstorms with the Speed and Distance Input from the User Using the Function **scanf**() . . . 84
8.1.3 Estimate the Error in Distance and Use the Member Function **getDistance()** . . . 85
8.1.4 ‡ Use the Functions **distance2angle**() and **angle2distance**() . . . 87
8.1.5 Measure the Clock Time Using the Member Function **systemTime**() . . . 89
8.1.6 Summary . . . 90
8.1.7 Terminology . . . 90
8.1.8 Exercises . . . 91
8.2 Plot a Curve Using the Plotting Member Functions **scattern**() and **data2DCurve**() . . . 91
8.2.1 Summary . . . 95
8.2.2 Terminology . . . 96
8.2.3 Exercises . . . 96
8.3 Plot Distances versus Time . . . 96
8.3.1 Plot Distances versus Time for a Two-Wheel Mindstorms with the Specified Speed and Distance . . . 97
8.3.2 Plot Robot Distance in Number Line . . . 103
8.3.3 Plot Distances versus Time with an Offset for the Initial Position . . . 106
8.3.4 Plot Robot Distances in Number Line with an Offset for the Initial Position . . . 110
8.3.5 Summary . . . 112
8.3.6 Terminology . . . 113
8.3.7 Exercises . . . 113
8.4 Move a Two-Wheel Robot with the Specified Time . . . 117
8.4.1 Control a Mindstorms with the Speed and Time Input from the User Using the Function **scanf**() . . . 117
8.4.2 Get the Moved Distance Based on the Specified Speed and Time . . . 118
8.4.3 Plot Distances versus Time for a Two-Wheel Mindstorms with the Specified Speed and Time . . . 119
8.4.4 Summary . . . 122
8.4.5 Terminology . . . 123
8.4.6 Exercises . . . 123
8.5 Use Different Units for Speed, Radius, and Distance . . . 124
8.5.1 Summary . . . 125
8.5.2 Terminology . . . 126
8.5.3 Exercises . . . 126

9 Moving a Single Robot in a Coordinate System 128

9.1 Move a Mindstorms in a Coordinate System . . . 128
9.1.1 Move a Mindstorms in a Coordinate System . . . 128
9.1.2 Move a Hardware Mindstorms Not Starting in its Default Initial Position . . . 130
9.1.3 Summary . . . 132
9.1.4 Terminology . . . 132
9.1.5 Exercises . . . 133
9.2 Plot Positions of a Robot . . . 134
9.2.1 Summary . . . 141
9.2.2 Terminology . . . 141
9.2.3 Exercises . . . 141
9.3 Move a Mindstorms Along a Trajectory . . . 143
9.3.1 Move a Mindstorms Along a Trajectory Using an Expression . . . 143
9.3.2 Move a Mindstorms Along a Trajectory Using a Function . . . 145
9.3.3 Summary . . . 148
9.3.4 Terminology . . . 149
9.3.5 Exercises . . . 149
9.4 Trace the Positions of a Mindstorms . . . 150
9.4.1 Summary . . . 151
9.4.2 Terminology . . . 151
9.4.3 Exercises . . . 151
9.5 Record the Positions of a Mindstorms . . . 152
9.5.1 Summary . . . 154
9.5.2 Terminology . . . 154
9.5.3 Exercises . . . 154
9.6 Create an Obstacle Course with Points, Lines, and Text on a RoboSim Scene . . . 155
9.6.1 Summary . . . 159
9.6.2 Terminology . . . 159
9.6.3 Exercises . . . 159

10 Writing Programs to Control a Single Mindstorms 162

10.1 Move Joints Relative to their Current Positions . . . 162
10.1.1 Summary . . . 164
10.1.2 Terminology . . . 164
10.1.3 Exercises . . . 164
10.2 Multiple Motion Statements in a Program . . . 164
10.2.1 Summary . . . 165
10.2.2 Terminology . . . 165
10.2.3 Exercises . . . 165
10.3 Move Joints to their Absolute Positions and Reset to the Zero Positions . . . 165
10.3.1 Summary . . . 167
10.3.2 Terminology . . . 167
10.3.3 Exercises . . . 167
10.4 Get Joint Angles . . . 167
10.4.1 Get a Joint Angle . . . 167
10.4.2 Get Multiple Joint Angles . . . 169
10.4.3 Summary . . . 170

10.4.4 Terminology . . . 170
10.4.5 Exercises . . . 170
10.5 Move a Single Joint . . . 171
10.5.1 Move a Single Joint to the Absolute Position . . . 171
10.5.2 Move a Single Joint Relative to the Current Position . . . 172
10.5.3 Delay the Motion of a Mindstorms Using the Member Function **delaySeconds**() . . . 172
10.5.4 Summary . . . 173
10.5.5 Terminology . . . 173
10.5.6 Exercises . . . 173
10.6 Get and Set Joint Speeds . . . 174
10.6.1 Get and Set a Joint Speed . . . 174
10.6.2 Get and Set Joint Speeds . . . 175
10.6.3 Get and Set a Joint Angular Speed Ratio . . . 177
10.6.4 Get and Set Joint Speed Ratios . . . 178
10.6.5 Summary . . . 179
10.6.6 Terminology . . . 180
10.6.7 Exercises . . . 180
10.7 ‡ Convert Units of Angles between Degrees and Radians . . . 181
10.7.1 Summary . . . 183
10.7.2 Terminology . . . 183
10.7.3 Exercises . . . 183

11 Writing Advanced Programs to Control a Single Mindstorms 184

11.1 Plot Recorded Joint Angles and Time . . . 184
11.1.1 Summary . . . 188
11.1.2 Terminology . . . 189
11.1.3 Exercises . . . 189
11.2 Move Joints with a Specified Time . . . 190
11.2.1 Summary . . . 191
11.2.2 Terminology . . . 192
11.2.3 Exercises . . . 192
11.3 Hold the Joints At Exit . . . 192
11.3.1 Summary . . . 192
11.3.2 Terminology . . . 193
11.3.3 Exercises . . . 193
11.4 Measure the Clock Time for Moving a Joint Using the Member Function **systemTime**() . . . 193
11.4.1 Summary . . . 194
11.4.2 Terminology . . . 195
11.4.3 Exercises . . . 195

12 Sensory Information for a Mindstorms 196

12.1 Set and Get the LED Color by Name Using the Data Type **string_t** . . . 196
12.1.1 Summary . . . 198
12.1.2 Terminology . . . 198
12.1.3 Exercises . . . 199
12.2 Set the Buzzer Frequency of a Linkbot . . . 199
12.2.1 Set the Buzzer to a Specified Frequency . . . 199

12.2.2 Set the Buzzer Multiple Times Using a **while** Loop with a User Specified Frequency 200
12.2.3 Change the Buzzer Frequency for a Predetermined Number of Times 201
12.2.4 Summary 202
12.2.5 Terminology 203
12.2.6 Exercises 203
12.3 Set the Buzzers Frequencies of Multiple Robots 203
12.3.1 Summary 204
12.3.2 Exercises 204
12.4 Play Melody and Drive a Robot at the Same Time 205
12.4.1 Summary 207
12.4.2 Terminology 207
12.4.3 Exercises 207
12.5 Play Melodies with Multiple Robots 207
12.5.1 Summary 208
12.5.2 Exercises 208
12.6 Play Music Notes and Create Melody Files 208
12.6.1 Music Basics 209
12.6.2 Defining a Music Note in Ch 210
12.6.3 Playing an Array of Music Notes 210
12.6.4 Write and Play Customized Melody Files 211
12.6.5 Summary 213
12.6.6 Terminology 213
12.6.7 Exercises 213
12.7 Play Music Notes with Multiple Robots 213
12.7.1 Summary 214
12.7.2 Exercises 215
12.8 Get the Accelerometer Data 215
12.8.1 Summary 217
12.8.2 Terminology 217
12.8.3 Exercises 217
12.9 Get the Battery Voltage 217
12.9.1 Summary 220
12.9.2 Terminology 220
12.9.3 Exercises 220

13 Writing Programs to Control Multiple Individual Mindstorms 221

13.1 Control Multiple Mindstorms Using the Robot Control Panel 221
13.1.1 Summary 221
13.1.2 Terminology 221
13.1.3 Exercises 221
13.2 Control Multiple Mindstorms Using a Program 222
13.2.1 Exercises 222
13.2.2 Summary 223
13.2.3 Terminology 223
13.3 Blocking versus Non-Blocking Functions 223
13.3.1 Summary 226
13.3.2 Terminology 227

13.3.3 Exercises . . . 227
13.4 Record and Plot Two Joint Angles versus Time . . . 227
13.4.1 Summary . . . 231
13.4.2 Terminology . . . 231
13.4.3 Exercises . . . 231
13.5 Synchronize the Motion of Multiple Mindstorms . . . 232
13.5.1 Summary . . . 235
13.5.2 Terminology . . . 236
13.5.3 Exercises . . . 236
13.6 Move Multiple Mindstorms with Specified Distances or Joint Angles . . . 236
13.6.1 Summary . . . 239
13.6.2 Terminology . . . 239
13.6.3 Exercises . . . 239
13.7 Plot Recorded Distances and Time for Multiple Mindstorms . . . 240
13.7.1 Summary . . . 244
13.7.2 Terminology . . . 245
13.7.3 Exercises . . . 245
13.8 Move Multiple Mindstorms with Specified Time . . . 245
13.8.1 Summary . . . 249
13.8.2 Terminology . . . 250
13.8.3 Exercises . . . 250
13.9 Move Multiple Mindstorms Forever . . . 252
13.9.1 A Baton Passing Robot Relay Race . . . 255
13.9.2 Summary . . . 257
13.9.3 Terminology . . . 258
13.9.4 Exercises . . . 258
13.10 Plot Recorded Distances and Time for Multiple Mindstorms with an Offset for the Distance . 259
13.10.1 Summary . . . 266
13.10.2 Terminology . . . 267
13.10.3 Exercises . . . 267
13.11 ‡ Copy Motions of a Controller Mindstorms to a Controlled Mindstorms . . . 270
13.11.1 Repeat Actions Using a **while** Loop . . . 270
13.11.2 Summary . . . 273
13.11.3 Terminology . . . 273
13.11.4 Exercises . . . 274

14 Moving Multiple Robots in a Coordinate System 275

14.1 Move Multiple Mindstorms in a Coordinate System . . . 275
14.1.1 Summary . . . 277
14.1.2 Terminology . . . 278
14.1.3 Exercises . . . 278
14.2 Trace and Record the Positions of Multiple Mindstorms in RoboSim . . . 279
14.2.1 Exercises . . . 283
14.3 Move Multiple Mindstorms Along Different Trajectories . . . 284
14.3.1 Move Multiple Mindstorms Along Different Trajectories Using Expressions . . . 284
14.3.2 Move Multiple Mindstorms Along Different Trajectories Using Functions . . . 287
14.3.3 Move Multiple Mindstorms Along Different Trajectories Using a Loop . . . 289

14.3.4 Summary . . . 293
14.3.5 Terminology . . . 293
14.3.6 Exercises . . . 294

15 Writing Programs to Control One or Multiple Groups of Mindstorms 295

15.1 Copy Motions from One Mindstorms to Multiple Mindstorms with Identical Movements . . . 295
15.1.1 Summary . . . 297
15.1.2 Terminology . . . 297
15.1.3 Exercises . . . 297
15.2 Control Multiple Groups of Mindstorms . . . 297
15.2.1 Summary . . . 301
15.2.2 Terminology . . . 301
15.2.3 Exercises . . . 302

16 Advanced Sensory Information for a Mindstorms 303

16.1 Retrieving Information From Sensor Panel . . . 303
16.1.1 Sensor Ports . . . 303
16.1.2 Sensors Panel . . . 304
16.1.3 Summary . . . 306
16.1.4 Terminology . . . 307
16.1.5 Exercises . . . 307
16.2 Retrieving Information From Sensor Panel . . . 307
16.2.1 Types of Sensors and their Modes . . . 307
16.2.2 Sample Programs for Initializing and Retrieving Information From Sensors . . . 314
16.2.3 Summary . . . 317
16.2.4 Terminology . . . 318
16.2.5 Exercises . . . 318
16.3 Advanced Sensor Functions . . . 319
16.3.1 Summary . . . 323
16.3.2 Terminology . . . 323
16.3.3 Exercises . . . 323

Appendix 324

A Using Advanced Programming Features 325

A.1 Make a Decision Using an **if** and **else if** Statements . . . 325
A.2 Use a **while**-Loop for Repeating Motions . . . 328
A.3 Use a **for**-Loop for Repeating Motions . . . 329
A.4 Use a Function with an Argument of the **CMindstorms** Class . . . 330
A.4.1 Exercises . . . 331

B Controlling Mindstorms and Linkbots in a Single Program 332

B.0.1 Exercises . . . 335

C Melodies and Music Notes Defined in Ch **337**
C.1 Melodies in Ch 337
C.2 Music Notes Defined in Ch 338

D Quick References to Ch **340**
D.1 Ch Language 340
D.1.1 Reserved Keywords 340
D.1.2 Reserved Names 340
D.1.3 Punctuators 340
D.1.4 Comments 340
D.1.5 Header Files 340
D.1.6 Declaration of Variables for Integers and Decimals 341
D.1.7 Declaration of Arrays 341
D.1.8 Declaration of Functions 341
D.1.9 Define Macros 341
D.1.10 The **if-else** Statement 341
D.1.11 The **while** Loop 341
D.1.12 Input Function **scanf**() and Output Function **printf**() 341
D.1.13 Math Operators 342
D.1.14 Relational Operators 342
D.1.15 Math Functions **abs**(), **sqrt**(), **pow**(), and **hypot**() 342
D.2 Member Functions of the Plotting Class **CPlot** 342

E Quick Reference to Mindstorms Member Functions **347**
E.1 Data Types 347
E.2 Member Functions Available in Class **CMindstorms** 348
E.3 Member Functions Available only in the Class **CMindstorms** for a Two-Wheel Robot . . . 355
E.4 Member Functions Available in Class **CMindstormsGroup** 357
E.5 Member Functions Using x-y Coordinates for Two-Wheel Robots 359
E.6 Member Functions Available only in RoboSim and RoboBlockly 361
E.7 Member Functions Related to Sensors for Mindstorms 361
E.8 Member Functions Not Presented in This Book 365

F Common Mistakes in Writing Ch Programs **366**

Index **368**

PREFACE

Robotics can easily get students engaged and excited about learning science, technology, engineering, and math (STEM) concepts while having fun. However, most robotic systems and programming environment are not suitable for formal math and science education because of their complexity. Working with its industrial partners, the UC Davis Center for Integrated Computing and STEM Education (C-STEM) has developed innovative computing and robotics technologies to transform math education using coding, making, and robotics. C-STEM Studio is a user-friendly platform for learning STEM in C using a user friendly C/C++ interpreter Ch. A single Ch program can control multiple robots of different types with a variety of different configurations, as shown in Appendix B. Based on our over two decades of research on computer and robot programming, I believe robot programming in Ch presented in this book is the simplest possible approach to program a single robot or multiple robots in text-based programming. C-STEM Studio can be freely downloaded from the C-STEM web site at **(http://c-stem.ucdavis.edu)**. C-STEM Studio are specially designed for integrating computing, robotics, and engineering into K-14 math and science education in both formal and informal programs. Currently, C-STEM Studio can be used to control both modular robots of Barobo Linkbot, Lego Mindstorms NXT and EV3, as well as Raspberry Pi and Arduino conveniently. It is especially suited for increasing student motivation and success in learning math and science with hands-on real-world problem solving and sparking their interest in STEM subjects leading to STEM related careers and post-secondary study.

This book is a gentle introduction to robot programming with NXT and/or EV3. It teaches the absolute beginners the underlying working principles of robotics and robot programming, with an emphasis on learning math, science, technology, and engineering (STEM) using robots. The book is a step-by-step guide on how to use NXT and/or EV3 to solve applied problems. The programming technique for controlling Linkbot and NXT/EV3 is the same, as shown in the companion textbook *Learning Robot Programming with Linkbot for the Absolute Beginner*. Therefore, the concepts and ideas that students learned to program one type of robot can be applied to other type. A simple program can control both Linkbot and NXT/EV3 in different configurations. The contents in this book can be readily integrated into teaching various STEM subjects for personalized and collaborative learning in classroom, afterschool and out-of-school programs. The materials are presented in such a manner that they can be adapted by instructors to meet the unique needs of their students.

Prerequisites

The mathematical prerequisite for the book is basic math taught in elementary school. No prior computer programming and robotics experience is required. Therefore, **anyone can use this book to learn robot programming.**

Organization of the Book

The topics in the manuscript are carefully selected and organized for the best information flow for beginners to learn how to use and program robots for solving practical and realistic problems while having fun. I believe that students who have mastered the topics and working principles presented in the book shall be able to embark on applying the robotics concepts to various STEM subjects and applications. The manuscript is organized as follows:

Chapter 1 is an introduction to robotics, RoboPlay Competition **(http://www.roboplay.org)**, and starting to use the NXT/EV3.

Chapter 2 uses a user friendly graphical user interface called Ch Mindtosrms to control NXT/EV3.

Chapter 3 introduces the robot programming using a C/C++ interpreter Ch.

Chapter 4 describes how to use RoboSim for robot simulation.

Chapter 5 presents basic programming features about using variables and generating Ch robot programs using RoboBlockly.

Chapter 6 introduces input/output functions and their applications in robot programming, and number line for distance.

Chapter 7 describes how to write programs to control a group of Mindstorms to perform identical tasks such as dancing.

Chapter 8 describes how to control a Mindstorms configured as a two-wheel robot. The two-wheel robot is particularly suitable for learning math and science concepts.

Chapter 9 describes features available only in RoboSim and their applications for driving a two-wheel robot. Section 9.1 can be introduced right after Chapter 6.

Chapter 10 describes how to write programs to control a single Mindstorms with different motion characteristics.

Chapter 11 describes how to write advanced programs to control a single Mindstorms.

Chapter 12 describes how to write programs to process some basic sensory information for Mindstorms.

Chapter 13 describes how to write programs to control multiple individual Mindstorms.

Chapter 14 describes features available only in RoboSim and their applications for driving multiple two-wheel robots.

Chapter 15 describes how to write programs to control one or multiple group of Mindstorms.

Chapter 16 describes how to write advanced programs to process different sensory information for Mindstorms.

Appendix A presents a few sample programs using programming features not covered in this book.

Appendix C lists the melodies and music notes defined in Ch.

Appendix D contains quick references to Ch features used in the book.

Appendix E contains quick references to member functions of the Mindstorms class **CMindstorms**.

Appendix F lists common mistakes in writing Ch programs.

Appendix B gives examples how to control Linkots and Lego Mindstorms NXT/EV3 in a single program.

The subsection **Summary** at the end of each section summarizes what you should have learned in the section. The subsection **Terminology** summarizes all terminologies and topics presented in the section.

Symbols and Notations Used in the Book

This book was typeset by the author using LaTeX. Programs in the book are displayed with the light blue background and syntax highlighting as shown in the following line of the code.

```
    printf("Hello, world!\n");
```

The output from programs are displayed with the grey background as shown in the following output.

```
    Hello, world!
```

The interactive execution of programs is displayed with the dark blue background as shown in the following interactive execution.

```
    Enter the weight in ounces.
    4.5
    The ice cream costs $2.11
```

The definition of new functions and member functions is displayed with the light pink background as shown in the following definition.

```
plot.title("title");
```

Special notes and important points are highlighted with the yellow background. Keywords such as **int** and **double** in C are in red color. Reserved words such as **sqrt** and **printf** are in pink color. The ***definition*** for a word is in green color.

The Common Core State Standards in titles of subsections, such as **A-REI.6**, are also in green color.

Sections marked with the double dagger symbol '‡' use concepts beyond Algebra I or advanved robotics concepts. They can be skipped as they do not include prerequisite skills necessary for later chapters.

The exercise symbol E indicates the location to pause for students to solve problems in the exercise section.

Using this Book as a Textbook or Supplementary Textbook

This is a comprehensive book on robot programming for solving applied problems in engineering, math, and science. Below are some possible ways to use the book. The book can be used as a textbook for courses on **Robotics**, **Engineering**, **Computer Programming**, **Computer Technology**, etc. It can also be used as a supplementary textbook for **Math 6, Math 7, Math 8, Pre-Algebra, Algebra I, Integrated Math I**, and **Physical Science**. In addition, the book can be used for afterschool programs as well as computing and robotics camps.

For teaching students in elementary schools or a few hour introductory robotics activities, only materials in Chapter 1 and Chapter 2, without programming may be covered.

Available Teaching Resources

To use this manuscript for teaching, instructors can contact the author to obtain related teaching materials, including the source code for all programs presented in this manuscript, PowerPoint slides for classroom presentation, and solutions for exercises.

The PDF file of this book is available in the C-STEM Studio. RoboPlay Challenge booklets for previous years are also distributed in C-STEM Studio. These challenge tasks can be used as additional exercises.

Copyright and Permission to Use

Acknowledgment

This material was supported in part by the National Science Foundation under Grant No. CNS-1132709, IIS-1208690, and IIS-1256780. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contacting the Author

I appreciate any criticisms, comments, identification of errors in the text or programs, and suggestions for improvement of this manuscript from both instructors and students. I can be reached over the Internet at

info@c-stem.ucdavis.edu

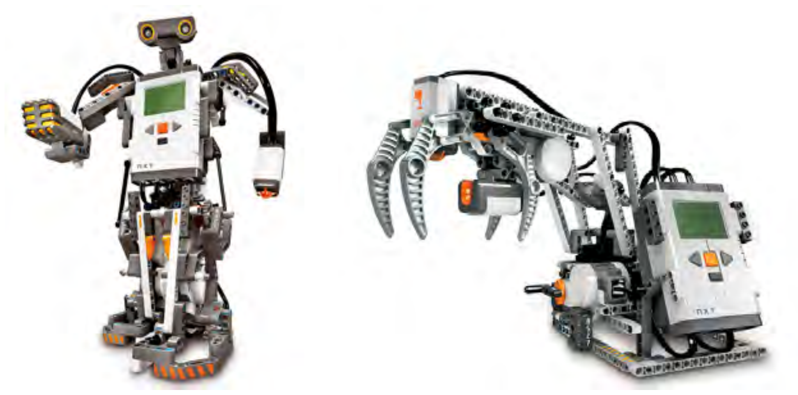
Harry H. Cheng

CHAPTER 1

Introduction

1.1 Introduction

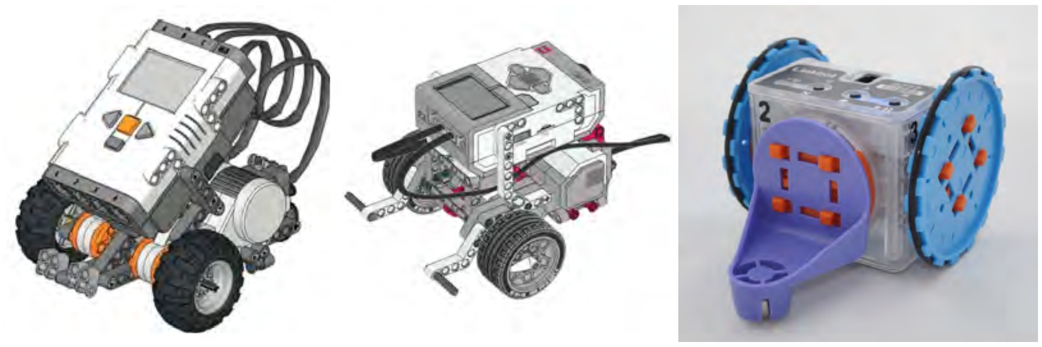
A ***robot*** is a re-programmable machine that is able to move, sense, and react to its environment. ***Robotics*** is a branch of technology that deals with the design, construction, operation and application of robots and the related computing systems for the control, sensing, and information processing. Robotics can be used to help learn science, technology, engineering, and math (STEM) concepts while having fun. The **Mindstorms brick** is designed as a building block. A single Lego Mindstorms brick can connect to motors and sensors that allow it to perform a multitude of tasks. Mindstorms can be configured into various geometries for different applications, such as a crane, and a humanoid, as shown in Figure 1.1



(a) Humanoid (b) Machine

Figure 1.1: Different configurations built with Mindstorms.

There are two versions of Mindstorms bricks, called EV3 and NXT. The shapes for both bricks are similar. There are four locations where motors can be connected on the EV3, and three on the NXT. Other than the difference in the number of motor ports, the interface and programming for both EV3 and NXT are the same. The term Mindstorms in this manuscript refers to both EV3 and NXT. Mindstorms can also be programmed to be used together with low-cost modular robots called Linkbots. An EV3 brick, NXT brick, and Linkbot-I are shown together in their vehicle configurations in Figure 1.2.



(a) NXT Vehicle (b) EV3 Vehicle (c) Linkbot-I

Figure 1.2: Mindstorms robots and Linkbot

A robot can only perform tasks it has been programmed to do. When a robot does something smart, it is because a smart person has written a smart program to control the device. This introductory book teaches the absolute beginners without any prior computer programming and robotics experience the underlying working principles of robotics and robot programming. You will learn how to write your own programs so that a robot will do what you want it to do. You will also be able to write programs to control multiple Mindstorms.
Do Exercises 1, 2, 3, 4, and 5 on page 3.

1.1.1 Summary

This section summarizes what you should have learned in this session.

1. A robot is a re-programmable machine that moves, senses, and reacts to its environment.
   (a) Robots can only perform tasks they have been programmed to accomplish
   (b) Robotics is a branch of technology that deals with design, construction, operation and application of robots and the related computing systems for the control, sensing, and information processing for robots.
2. A Mindstorms brick is designed as a building block, and can connect to multiple motors and sensors. There are two types of Mindstorms bricks.
   (a) EV3 brick: four motor ports.
   (b) NXT brick: three motor ports.
   (c) Programming for EV3 and NXT is the same.

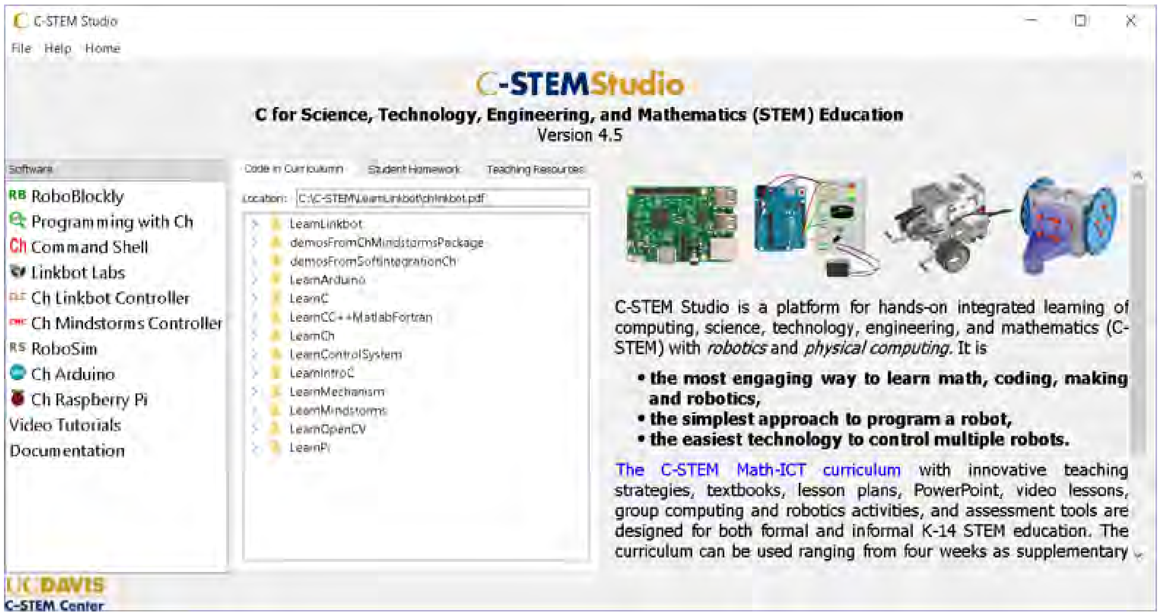
1.1.2 Terminology

robot, robotics, Mindstorms brick, EV3, NXT, Linkbot-I, robot, robotics.

1.1.3 Exercises

1. What is a robot?
2. Where have you seen a robot?
3. What is robotics?
4. Explain what the differences are between the EV3, and the NXT?
5. List at least 2 configurations that the Mindstorms can be put into for various applications.

1.2 C-STEM Studio



C for Science, Technology, Engineering, and Mathematics (STEM) Education
Version 4.5

Location: C:\C-STEM\LearnLinkbot\chlinkbot.pdf

C-STEM Studio is a platform for hands-on integrated learning of computing, science, technology, engineering, and mathematics (C-STEM) with *robotics* and *physical computing*. It is

- **the most engaging way to learn math, coding, making and robotics,**
- **the simplest approach to program a robot,**
- **the easiest technology to control multiple robots.**

The C-STEM Math-ICT curriculum with innovative teaching strategies, textbooks, lesson plans, PowerPoint, video lessons, group computing and robotics activities, and assessment tools are designed for both formal and informal K-14 STEM education. The curriculum can be used ranging from four weeks as supplementary

Figure 1.3: C-STEM Studio.

C-STEM Studio is a platform, specially designed for the absolute beginners, for hands-on integrated learning of computing, science, technology, engineering, and mathematics (C-STEM) with robotics.

C-STEM Studio is a user-friendly platform for using the C-STEM integrated curriculum by university faculty and students, K-12 teachers and students, parents, volunteers, etc. It is integrated with the innovative educational computing and robotics technologies for learning STEM subjects, including C/C++ interpreter Ch, Linkbot Labs, Ch Linkbot Controller, Ch Mindstorms Package and Controller for Lego Mindstorms NXT/EV3, RoboSim, RoboBlockly, Ch Arduino, and Ch Raspberry Pi, as shown in Figure 1.3.

C-STEM Studio also includes the code, comprehensive documentations, teacher' guides, and textbooks used in the C-STEM curriculum. C-STEM Studio can be used to easily control multiple Linkbots, NXT, and EV3 in a single program with only a few lines of code. Users can learn STEM subjects by solving complex real-world problems with coding and robotics.

In this book, we will use C-STEM Studio to learn robot programming with Mindstorms. Most programming features are illustrated using NXT or EV3 configured in a two-wheel vehicle as shown in Figure 1.2.

C-STEM Studio contains the instructions on how to construct these two-wheel robots. C-STEM Studio can be freely downloaded from the C-STEM web site at **http://c-stem.ucdavis.edu**.

To launch the C-STEM Studio as shown in Figure 1.3, in Windows, click on the icon labeled "C-STEM Studio" on your desktop, as shown in Figure 1.4. On Mac OS X systems, C-STEM Studio application is located inside the "Applications" folder in Finder.

Figure 1.4: The icon for C-STEM Studio.

Do Exercise 1 on page 4.

1.2.1 Summary

C-STEM Studio

1.2.2 Terminology

C-STEM Studio.

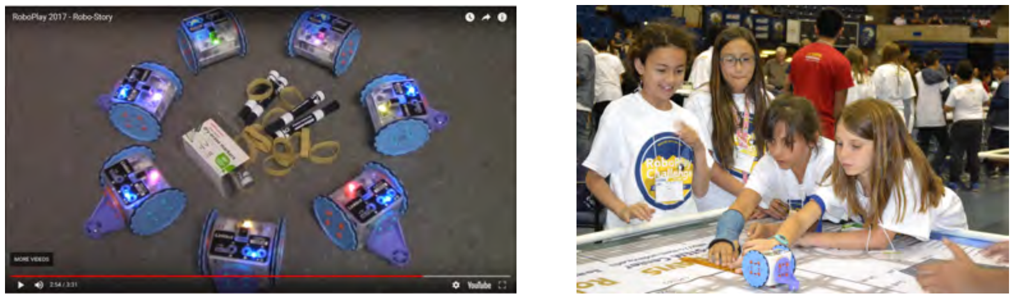
1.2.3 Exercises

1. Watch the video tutorial "Introduction to C-STEM Studio" in http://c-stem.ucdavis.edu/studio/tutorial/.

1.3 RoboPlay Competition

After learning the topics in this book, students shall be able to participate in RoboPlay Competition with more detailed information available at **http://www.roboplay.org**. There are two categories for RoboPlay Competition: RoboPlay Video Competition and RoboPlay Challenge Competition.

The RoboPlay Video Competition is a robotics-centric video competition for K-14 students. It designed for students to learn robotics while having fun and exploring their creativity in writing, storytelling, art, music, choreography, design, video editing and film production, and at the same time seamlessly learning C-STEM subjects. The necessary robot coordination to match the movement of multiple modules to music requires not only teamwork in designing a well-organized visual performance, but also the math and programming skills to produce the desired actions. The competitions enable students with different interests to explore the basic concepts of C-STEM in conjunction with their artistic and music talents. A student of average skill should be able to reproduce a video with the documentation submitted for the video.



(a) RoboPlay Video Competition (b) RoboPlay Challenge Competition

Figure 1.5: RoboPlay Competition.

The RoboPlay Challenge Competition is a theme-based level playing field robotics competition for K-14 students, held on each May. It is designed for students to showcase their real-world problem solving skills in a competitive environment. This competition simulates a real-world problem, such as *space exploration, search and rescue operation,* where a robotic solution must be quickly developed and deployed, using only existing resources with the constraint of the time. The competition challenges students to creatively use modular robots and accessories to complete various tasks. The competition arena and specific challenges will be unknown to participants until the day of the competition. Using their math, programming, and problem solving skills, student teams try to most efficiently obtain the highest score for each task on their own.

Both Linkbot and/or NXT/EV3 are allowed for RoboPlay Video Competition. Although only Linkbots are allowed for RoboPlay Challenge Competition, most challenges can also be solved using NXT/EV3.

RoboPlay Challenge booklets with all previous RoboPlay Challenge tasks are distributed through C-STEM Studio. You may work these challenge tasks as additional exercises.

Do Exercises 2 and 3 on page 5.

1.3.1 Summary

1. The RoboPlay Competition includes two categories: RoboPlay Video Competition and RoboPlay Challenge Competition.
2. C-STEM Day.

1.3.2 Terminology

RoboPlay Competition, C-STEM Day.

1.3.3 Exercises

1. Watch a RoboPlay video in http://www.roboplay.org.
2. What are RoboPlay Competitions? How many competitions do they consist of?
3. When is the RoboPlay Competition? When would you need to start preparing to compete in the RoboPlay Competition?

1.4 Major Features of Mindstorms and Ch Mindstorms Package

A wide range of technologies have been integrated into Mindstorms. A Mindstorms brick can connect to multiple different motors and sensors at the same time. Ports A-C on NXT and A-D on EV3 accept Lego Mindstorms motors. Motors come in two different sizes, large and medium, and are responsible for the movement of the robot. Ports 1-4 on the NXT and EV3 accept Lego Mindstorms sensors. There are many types of sensors that can be connected to a Mindstorms brick- color, touch, light, sound, force, angle, ultrasonic, accelerometer, etc- that return information related to the Mindstorms or its surroundings. Figure 1.6 shows an NXT brick with motors and sensors attatched. The setup of the EV3 brick is similar.

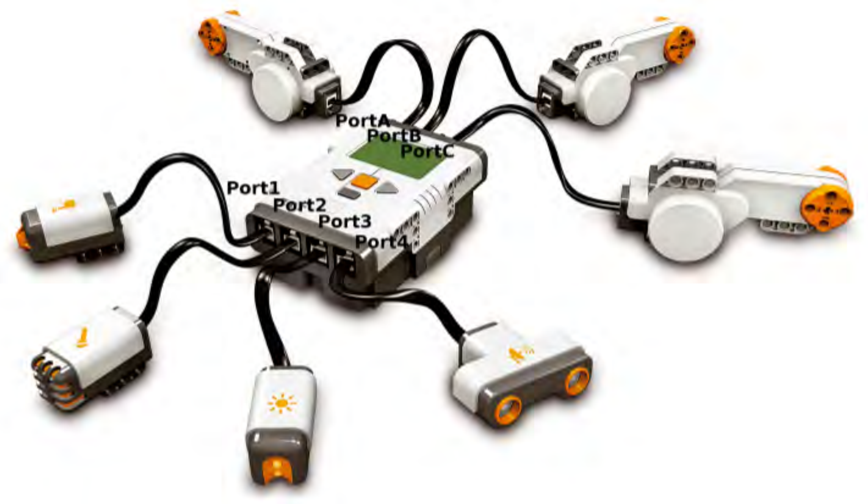


Figure 1.6: A Mindstorms NXT brick with three motors and four sensors attatched.

The Ch Mindstorms Package consists of a set of Application Programming Interface (API) functions enabling programmers to write programs in C or C++ that can access and control many features of the Lego Mindstorms brick. The API converts the complex messaging tasks required to communicate with robots into easy to use functions, allowing users to focus their efforts on their robotic application, rather than the details of communication. The API of the Ch Mindstorms Package was designed to support and augment all of the functionality found in the Ch Mindstorms Controller. The Ch package further enhances the capabilities of the Ch Mindstorms Controller by adding data collection and plotting capabilities. Additionally a program, written in C source code can be directly run from any platform in Ch without tedious compile/link/execute/debug cycles.

The communication between the user, the computer, the robot controller, the sensors, and the motors can be described in Figure 1.7. Once a robot is connected to a computer and a controlling program has started, the program instructions are sent from the computer to the robot. The robot controller will process these instructions perform appropriate tasks by sending commands to the motors or receiving data from the sensors. The robot can collect sensor data and motor encoder counts, and the data can be sent back to the computer for further manipulation, display, or stored in the computer for the user.

With Ch Mindstorms Package, you can quickly develop a Lego Mindstorms robotic application and log

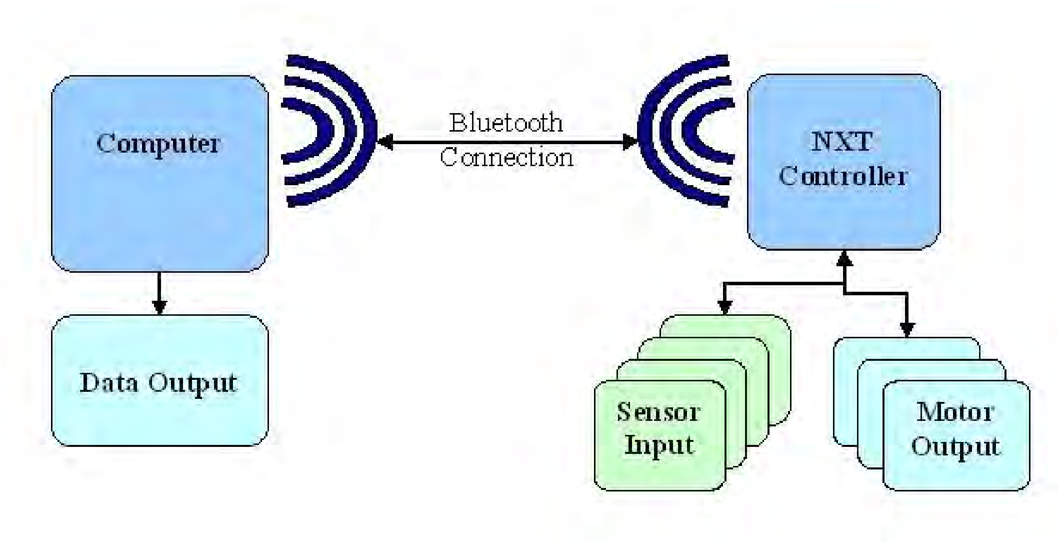


Figure 1.7: Communication diagram of Lego Mindstorms.

your results. The ease of design and added functionality makes the Ch package a good candidate for any Lego Mindstorms robotic programming applications.

E Do Exercises 1, 2, 3, 4, and 5 on page 7.

1.4.1 Summary

1. A Mindstorms brick can connect to multiple different motors and sensors at the same time.
2. Ports A-C on NXT and A-D on EV3 accept Lego Mindstorms motors, which are responsible for movement of the robot.
3. Ports 1-4 on the NXT and EV3 accept Lego Mindstorms sensors that return information related to the Mindstorms or its surroundings.
4. With Ch Mindstorms Package, you can quickly develop a Lego Mindstorms robotic application and log your results.

1.4.2 Terminology

Motor, Sensor, Ch Mindstorms Package, Ch Mindstorms Controller.

1.4.3 Exercises

1. What do Mindstorms use for wireless communication?
2. How many motors can an NXT connect to?
3. How many motors can an EV3 connect to?
4. How many sensors can an NXT connect to?
5. How many sensors can an EV3 connect to?

CHAPTER 2

Controlling a Mindstorms Using the Motion Control Panel

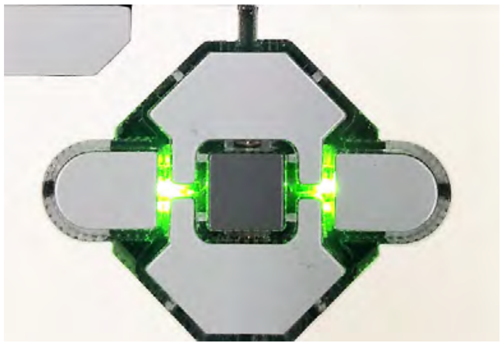
A Mindstorms can be conveniently controlled without writing a computer program. This chapter presents detailed steps to control a Mindstorms through a user friendly graphical user interface called Ch Mindstorms Controller (CMC).

2.1 Configuring Mindstorms NXT/EV3 for Wireless Control

Before you can use CMC to control robots, you will need to enable Bluetooth on both computer and robot sides. Then NXTs/EV3s can be paired with the computer through Bluetooth. After successfully paired with the computer, robots are able to be connected and controlled by the computer.

2.1.1 Turning a Mindstorms Brick On and 0ff

To turn on a Mindstom brick, simply press the square center button. On an EV3, this will be dark grey; on an NXT, this will be orange, as shown in Figure 2.1.


(a) The buttons on an EV3.

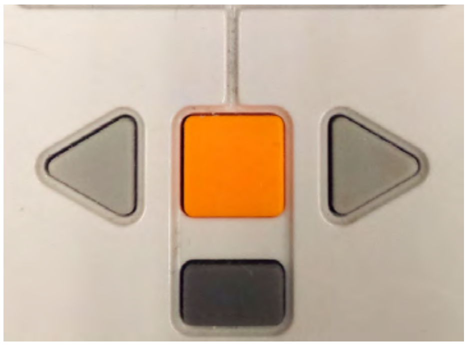
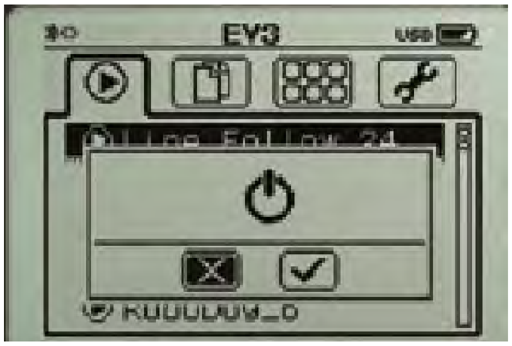
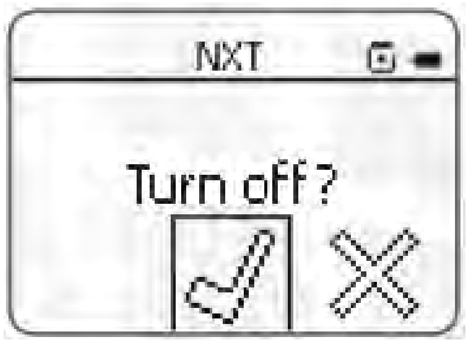

(b) The buttons on an NXT.

Figure 2.1: Buttons on Mindstorms bricks.

To turn off an EV3 brick, press the top left button until the message power symbol appears, as shown in Figure 2.2a. Press the center button to select check mark.



(a) Turning off an EV3.



(b) Turning off an NXT.

Figure 2.2: Turning off Mindstorms robots

To turn off an NXT brick, press dark grey bottom button until the message “Turn off?” appears, as shown in Figure 2.2b. Press the center button to select the check mark.

2.1.2 Check Bluetooth Support on Computers

A Bluetooth-enabled computer should have a Bluetooth icon, as figure 2.3 shown, locating at the lower right corner of the “Desktop”. The bluetooth icons might be a little different for different operating systems such as Windows, Mac OS X, etc. If there is no bluetooth support on your computer, you need a Bluetooth dongle like Figure 2.4 shown.

Figure 2.3: The Bluetooth icon.

Figure 2.4: A Bluetooth dongle.

2.1.3 Enable Bluetooth on NXTs/EV3s

If Bluetooth is enabled and visible to other devices, the robot should show the symbol on the top left corner of its display. Otherwise, you will need to turn Bluetooth on manually on the NXT/EV3 brick.

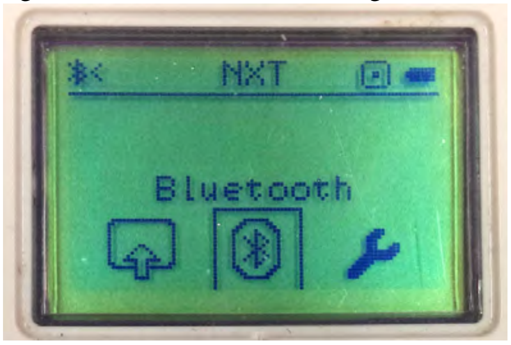
Bluetooth status in NXTs/EV3s NXTs/EV3s have the following Bluetooth status displayed on their screen,

- Bluetooth is enabled but not connected or visible to other Bluetooth devices.
- Bluetooth is enabled and visible to other Bluetooth devices but not connected.
- Bluetooth is enabled, visible to other Bluetooth devices and connected to other Bluetooth devices.

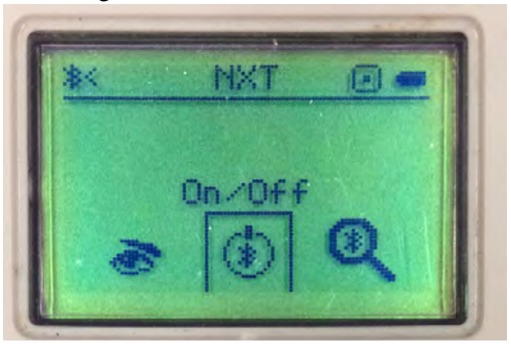
If non-of these status shows on your NXT/EV3 bricks, you need to enable Bluetooth manually.

Enable Bluetooth on NXT Bricks

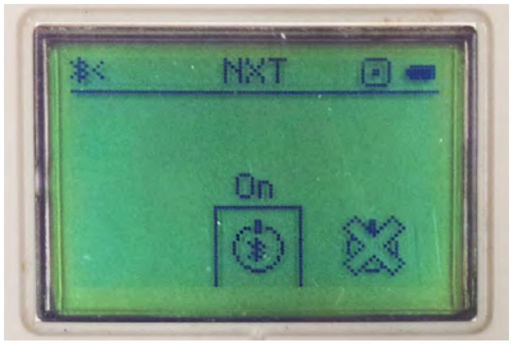
- On the main screen of a NXT, use right and left buttons to select the word "Bluetooth" as the following figure shown and click the orange button.



- Then use the right and left buttons to select the word "On/Off" as the following figure shown and click the orange button.

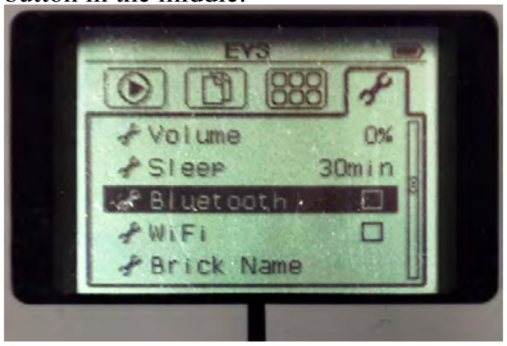


- Use right and left buttons to select the word "On" as the following figure shown and push the orange button.

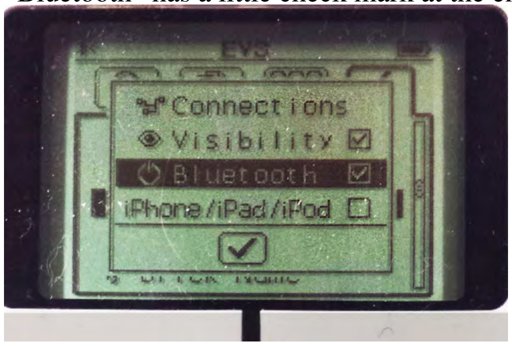


Enable Bluetooth on EV3 Bricks

- On the main screen of a EV3, use the right and left buttons to switch to “Settings” panel and use up and down buttons to select the word “Bluetooth” as the following figure shown and push the square button in the middle.



- In the popup dialog, use up and down buttons to select the word “Bluetooth” as the following figure shown and push the square button in the middle. Please make sure both the word “Visibility” and “Bluetooth” has a little check mark at the end of the word.

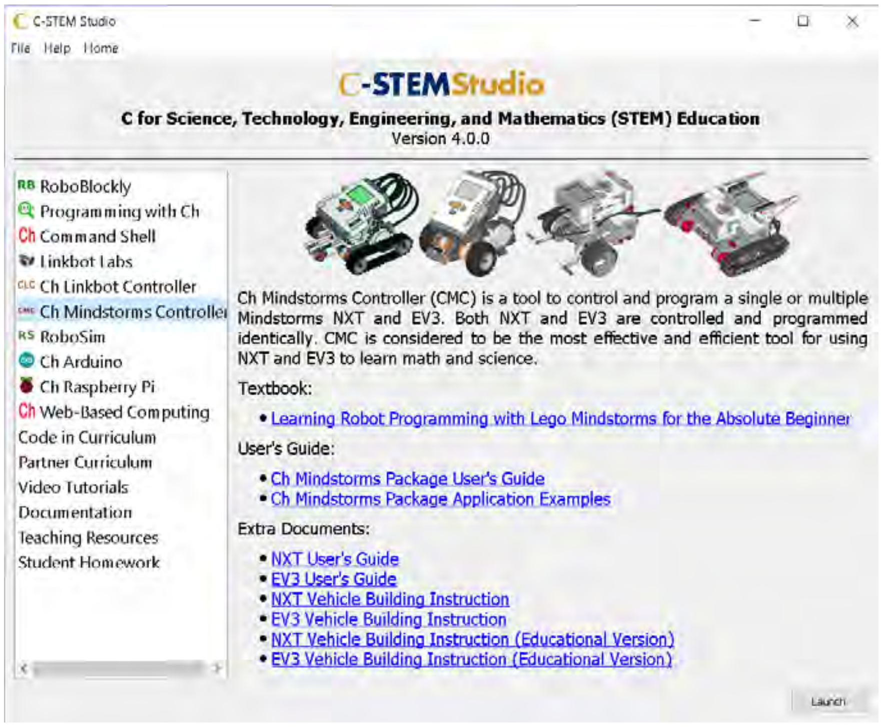


Make sure to uncheck the Bluetooth for iPhone and iPad.

2.1.4 Launch “Ch Mindstorms Controller (CMC)”

To access the “Ch Mindstorms Controller”, first open C-STEM Studio, located on the desktop for Windows and in the “Applications” folder in Finder for Mac OS X systems. On the left panel of C-STEM Studio, double click on “Ch Mindstorms Controller”, or click once and press the “Launch” button at the lower right corner of C-STEM Studio, as shown in Figure 2.5. The CMC constains hyperlinks to this textbook in PDF file, Ch Mindstorms Package Users’ Guide, advanced applications using Mindstorms, and Mindstorms

User's Guide, instructions on how to construct NXT and EV3 as two-wheel robot vehicles used in this document.



C for Science, Technology, Engineering, and Mathematics (STEM) Education
Version 4.0.0

Ch Mindstorms Controller (CMC) is a tool to control and program a single or multiple Mindstorms NXT and EV3. Both NXT and EV3 are controlled and programmed identically. CMC is considered to be the most effective and efficient tool for using NXT and EV3 to learn math and science.

Textbook:

- Learning Robot Programming with Lego Mindstorms for the Absolute Beginner

User's Guide:

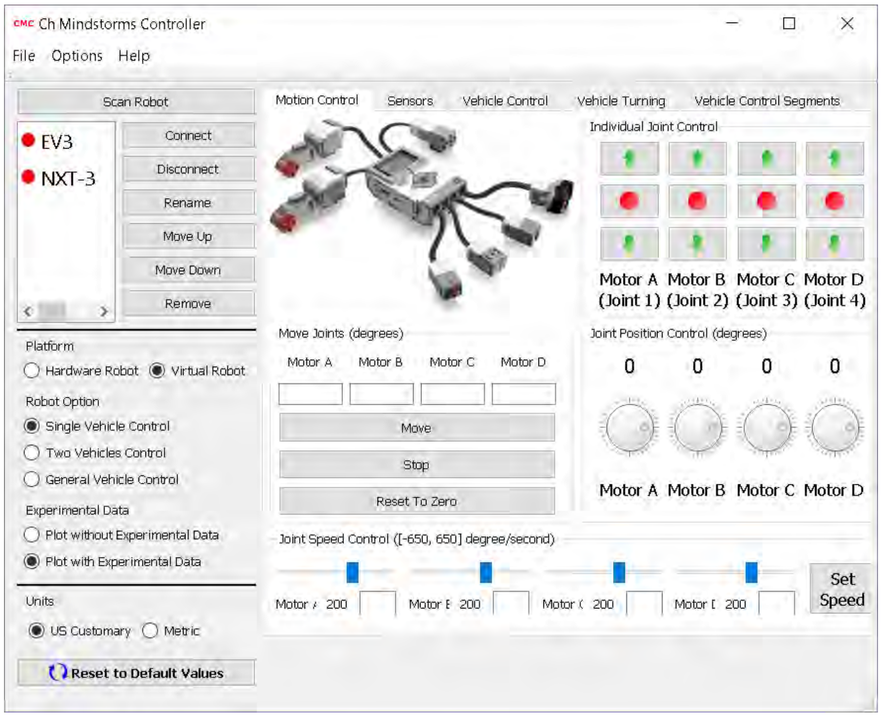
- Ch Mindstorms Package User's Guide
- Ch Mindstorms Package Application Examples

Extra Documents:

- NXT User's Guide
- EV3 User's Guide
- NXT Vehicle Building Instruction
- EV3 Vehicle Building Instruction
- NXT Vehicle Building Instruction (Educational Version)
- EV3 Vehicle Building Instruction (Educational Version)

Figure 2.5: Launching Ch Mindstorms Controller (CMC) from C-STEM Studio.

2.1.5 Configure Robots



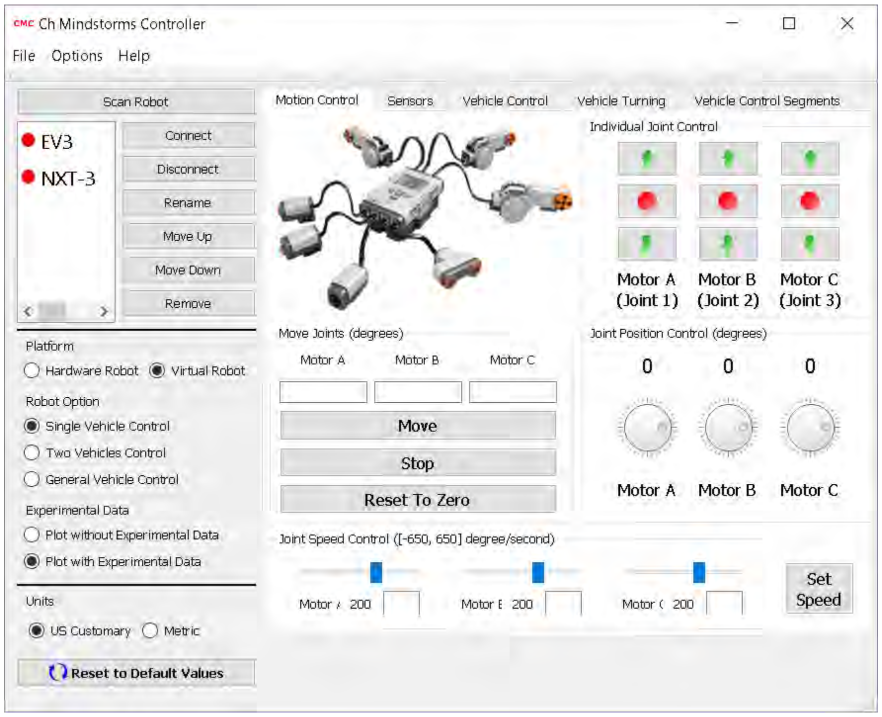
Platform: Hardware Robot [✓ Virtual Robot]

Robot Option: [✓ Single Vehicle Control] Two Vehicles Control General Vehicle Control

Experimental Data: Plot without Experimental Data [✓ Plot with Experimental Data]

Units: [✓ US Customary] Metric

Figure 2.6: The main window of the "Ch Mindstorms Controller" for EV3.



Platform: Hardware Robot [✓ Virtual Robot]

Robot Option: [✓ Single Vehicle Control] Two Vehicles Control General Vehicle Control

Experimental Data: Plot without Experimental Data [✓ Plot with Experimental Data]

Units: [✓ US Customary] Metric

Figure 2.7: The main window of the "Ch Mindstorms Controller" for NXT.

Scan Robots

After launching CMC, the Graphical User's Interface (GUI) should display as Figure 2.6 for EV3 and Figure 2.7 for NXT. Click the "Scan Robot" button on the GUI to start the robot scanner

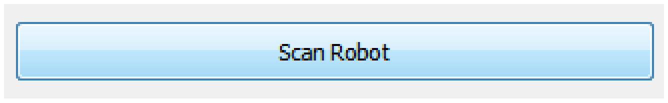


Figure 2.8: The Robot Scanner Button.

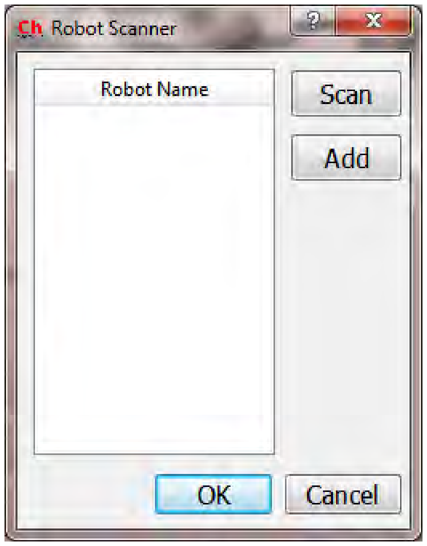
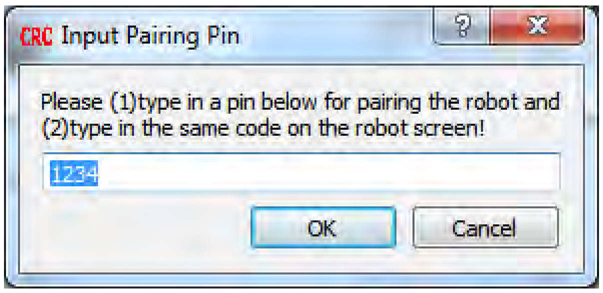


Figure 2.9: The Robot Scanner Dialog.

Click "Scan" button on the dialog to search nearby robots and a list of robots should be listed. Move the robot closer to the computer and scan again if your robot is not listed.

Add and Pair a Robot

Select a robot from the list and click "Add" button to add the robot to the control panel. You will need to pair the robot with your computer for the first time. A dialog will pop up requiring a pairing code as the following figure shows.



Please (1)type in a pin below for pairing the robot and (2)type in the same code on the robot screen!

1234

Figure 2.10: A dialog requiring pairing code.

On the robot, you will need to accept the connection from the computer and type in the same code. It may take a few seconds or more before the pairing code dialog or connection confirmation dialog shows on the robot display. For NXTs, you can directly type in the pairing code

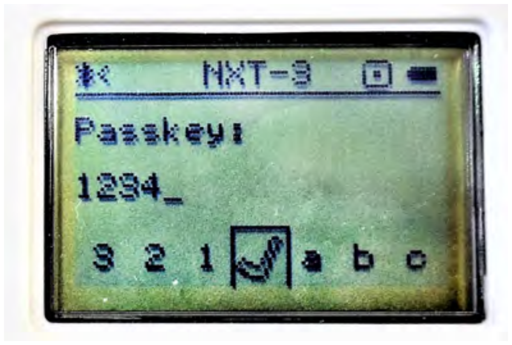
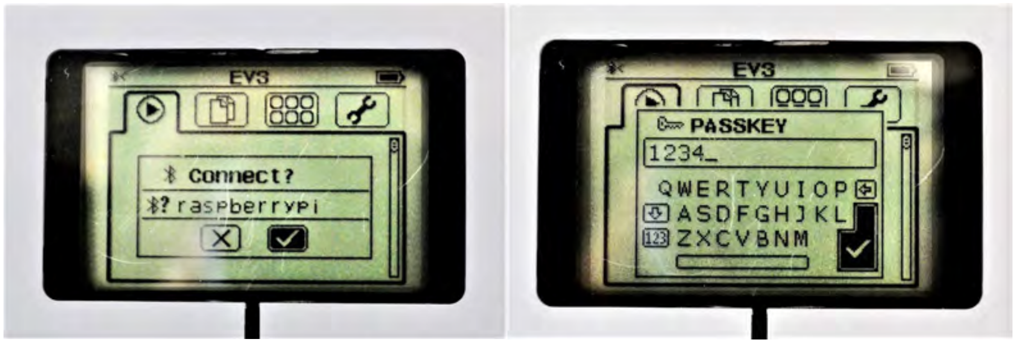


Figure 2.11: Type in pairing code on a NXT.

For EV3s, you need to confirm the connection before you can type in the pairing code



(a) Confirm Bluetooth Connection on a EV3. (b) Type in pairing code on a EV3.

Figure 2.12: Connecting EV3.

Afterwards, the system will pair the robot automatically, and the robot will be added into CMC control panel.

Rename a Robot

After a robot has been successfully added into the control panel, the name of the robot will show on the list as shown in Figure 2.13.

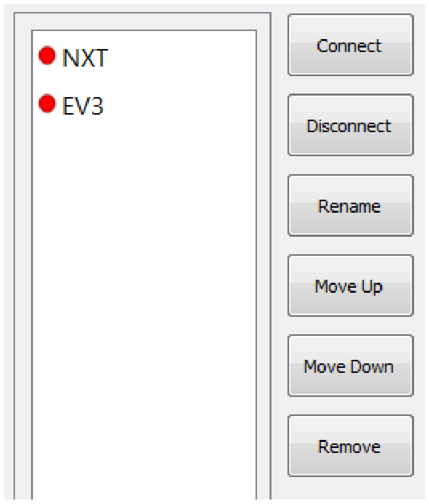
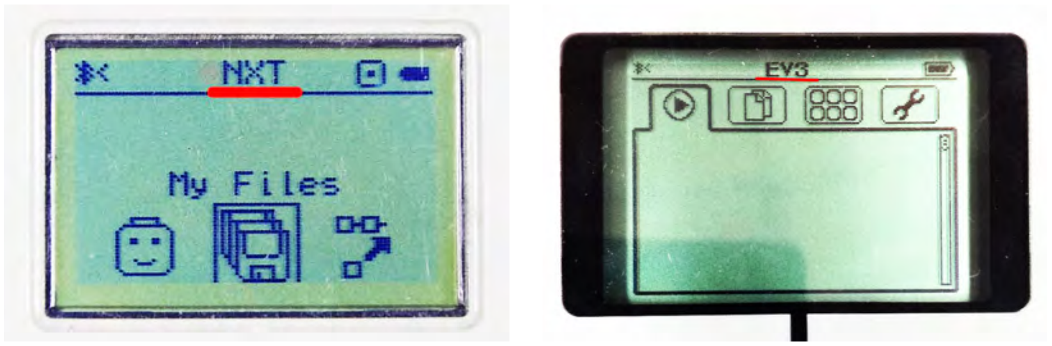


Figure 2.13: The list of robots.

The default name of a robot is “EV3” for Mindstorms EV3 and “NXT” for Mindstorms NXT. The name will also display on the robot side as following figures show.

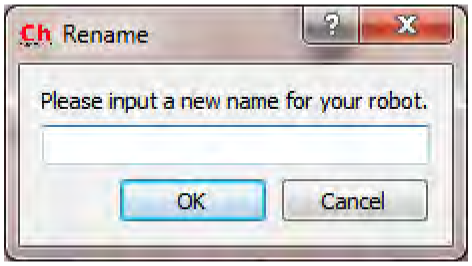


(a) NXT Name.

(b) EV3 Name.

Figure 2.14: Robot names.

However, if multiple robots have been added, it is better to give each robot a unique name. Select a robot from the list and click “Rename” button on the CMC control panel. A dialog will pop up requiring a new name for the selected robot as the following figure shows.



Please input a new name for your robot.

Figure 2.15: A dialog lequiring new name for a robot.

It may take a few seconds to rename a robot.

For a classroom with multiple robots, it is important to rename robots so that each robot has a unique name. Therefore, a computer may only connect to robot(s) to be controlled.
E Do Exercises 1 and 2, on page 17

2.1.6 Connect a Robot from Computer

To control a Mindstorms NXT or EV3, you need to connect it from a computer. To connect a NXT or EV3, you can click the button "Connect" in CMC shown in Figure 2.13.

2.1.7 Summary

This section summarizes what you should have learned in this session.

1. Before connecting to a Mindstorms robot, make sure that it has Bluetooth enabled.
2. Scan for, and connect to, a robot using Ch Mindstorms Controller (CMC).
3. CMC can connect to multiple robots at a time.
4. Mindstorms robots can be renamed.

2.1.8 Terminology

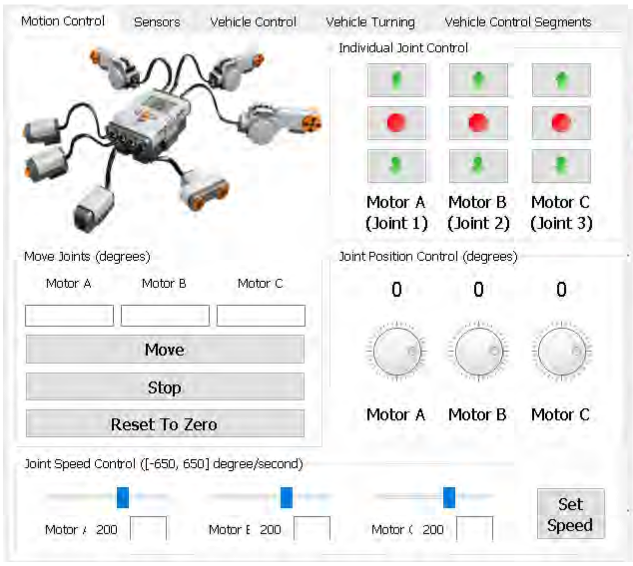
Ch Mindstorms Controller, CMC, Robot Scanner

2.1.9 Exercises

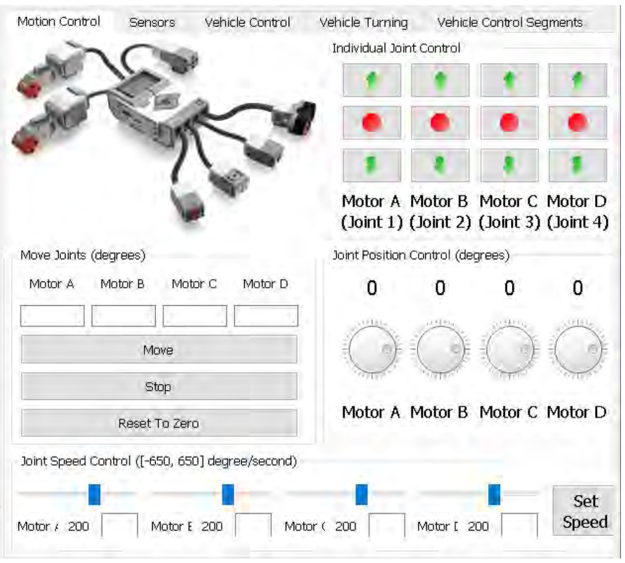
1. Watch the video tutorial "Setup for Ch Mindstorms Controller" in http://c-stem.ucdavis.edu/studio/tutorial/.
2. Connect the computer to your Mindstorms by scanning for the robot, adding the robot to the robot manager, and clicking connect.

2.2 Control a Mindstorms Using the Motion Control Panel

Motion Control Panel



(a) The "Motion Control" Panel for NXT.



(b) The "Motion Control" Panel for EV3

Figure 2.16: The "Motion Control" Panel of CMC

Once a robot is connected in CMC, the motor angles and speeds of the robot are displayed as shown in Figure 2.16. This dialog is located under the first tab of CMC, labeled "Motion Control". The "Motion Control" tab can be used not only to display the information about robot's motor positions but also to control the speeds and positions of the motors. The tab is divided up into 5 sections, including the "robot figure", the "Individual Joint Control" section, the "Joint Position Control" section, the "Move Joints" section and the "Joint Speed Control" section.

Individual Joint Control Section In the "Individual Joint Control" section, there are three sets of buttons as shown in Figure 2.17. Each set of buttons can be used to control the corresponding motor of the connected robot. When the up or down arrows are clicked, the robot begins to move the corresponding motor in either the positive or negative direction. The motor will move continuously until the stop button (located between the up and down arrows) is clicked.

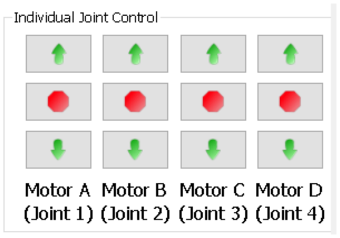


Figure 2.17: Individual Joint Control section

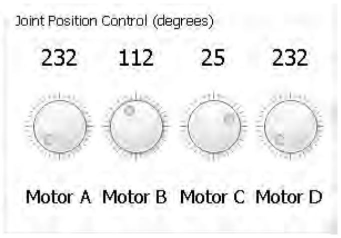


Figure 2.18: Joint Position Control section

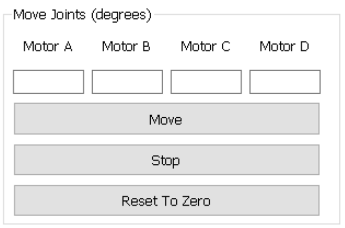


Figure 2.19: Move Joints section

Joint Position Control Section The “Joint Position Control” section, shown in Figure 2.18, displays and controls the positions of each of motors of a robot. The motor positions are displayed in the numerical text browser located above each dial panel. The displayed motor positions are in units of degrees. The dial panels also display the positions of motors. Users can also drag the dial panels and the motor will move to the dropped position. The motor angles are updated automatically when motors are controlled either from the control panel or a Ch program. However, in Mac OS X, this section is not updated when a Ch program is running due to the communication performance issue.

Move Joints Section This section, as shown in Figure 2.19, contains three editors for typing angles for corresponding motors to move and buttons for starting moving motors. Once the button “Move” is clicked, the robot will move each motor by the corresponding angle submitted. The stop button causes the robot to stop where it is and the “zero” button causes each motor reset back to zero position.

Joint Speed Control Section The “Joint Speed Control” section, as shown in Figure 2.20, displays and controls the motor speeds of the robot. The joint speeds are in units of degrees per second and a valid speed should be in the range of [−650, 650] degree/second for hardware robots, [−570, 570] degree/second for virtual robots in RoboSim. To set a specific desired motor speed for a particular motor, the motor speed may be typed directly into the edit boxes below the sliders, and the “Set Speed” button should be clicked. Dragging each slider can also modify the speed of corresponding motor.

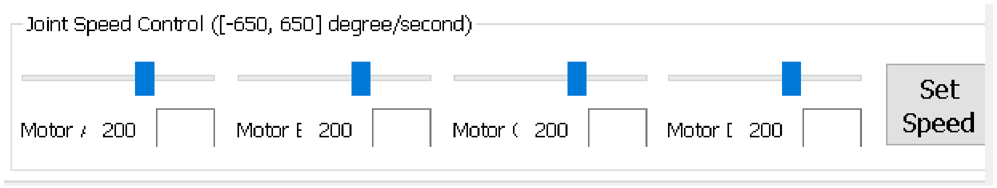


Figure 2.20: Joint Speed Control section

E Do Exercises 1, 2, 3, 4, and 5 on page 20.

2.3 Control Multiple Mindstorms

A computer can connect to multiple Mindstorms as described in section 2.1. The Motion Control Panel can be used to control multiple Mindstorms, one at a time. If a computer is connected to multiple Mindstorms as shown in Figure 2.21, an individual Mindstorms can be selected to be controlled by clicking the Mindstorms ID on the Robot Manager.

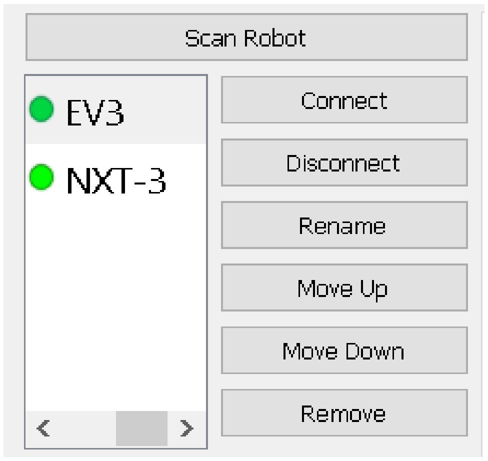


Figure 2.21: Two Mindstorms on the Motion Control Panel.

E Do Exercise 6 on page 21.

2.3.1 Summary

This section summarizes what you should have learned in this session.

1. Control the motion of joints of a Mindstorms individually through the “Motion Control Panel”.
2. Control the motion of a Mindstorms as a two-wheel vehicle.
3. Set the joint speeds of a Mindstorms. Joint speeds are specified in degrees per second.
4. Connect to multiple Mindstorms from a computer through CMC and control one at a time through the Motion Control Panel.

2.3.2 Terminology

Joint angle, joint speed, individual joint control, joint speed control, move joints section, and joint position control.

2.3.3 Exercises

1. Watch the Ch Mindstorms video tutorial “Motion Control” in http://c-stem.ucdavis.edu/studio/tutorial/.

2. Connect your computer to a Mindstorms through Ch Mindstorms Controller (CMC). Make sure the Mindstorms is in vehicle configuration. When the robot is in vehicle configuration, motor B will be referred to as ”joint 2”, and motor C will be referred to as ”joint 3”. Move joint 2 in the positive direction through the Individual Joint Control. Move joint 3 in the positive direction. Stop the motion of joints 2 and 3. Move joints 2 and 3 in the negative direction. then stop both joints.

3. Use the Move Joints section to reset all joints to 0 degrees. Use the Joint Postion Control to move motor B to a positive number between 0 and 180 degrees. Then use the Move Joints section to move motor C by 180 degrees.

4. First, roll the joints 2 and 3 (motors B and C) forward through the Individual Joint Control. Change the speed of joints 2 and 3 to 30 degrees per second through the Joint Speed Control while the robot is moving.

5. A Mindstorm, as a two-wheel robot, turns its two wheels at 45 degrees per second. (a) How long will it take for the robot to rotate its wheels two full rotations (720 degrees)? You may set the joint speeds first, then move the joints 2 and 3 to the specified position while timing the motion using a stop watch. (b) If the radius of the wheel is 1.1 inches, what is the distance that the robot has moved forward?

6. Work with your partner to connect a computer to two Mindstorms. Control the connected two Mindstorms individually using the Motion Control Panel of CMC on the computer.

CHAPTER 3

Getting Started With Programming Mindstorms

3.1 Get Started with Ch for Computer Programming

In Chapter 2, we learned how to control a Mindstorms using the Robot Control Panel in CMC. However, in order for a Mindstorms to solve complicated problems, we need to write a computer program to control the Mindstorms. Unlike using the Robot Control Panel, a computer program can be saved in a file for later use. It can also conveniently be copied to a new file and modified to solve similar problems.

The Mindstorms can be controlled using a C/C++ program through Ch, a C/C++ interpreter. Ch is user-friendly and specially designed for beginners to learn computer and robot programming. For example, when an error occurs, Ch will give an insightful error message, instead of confusing messages or crashing. Ch is available from SoftIntegration, Inc. at **http://www.softintegration.com**. Ch programs presented in this book are available through the C-STEM Studio.

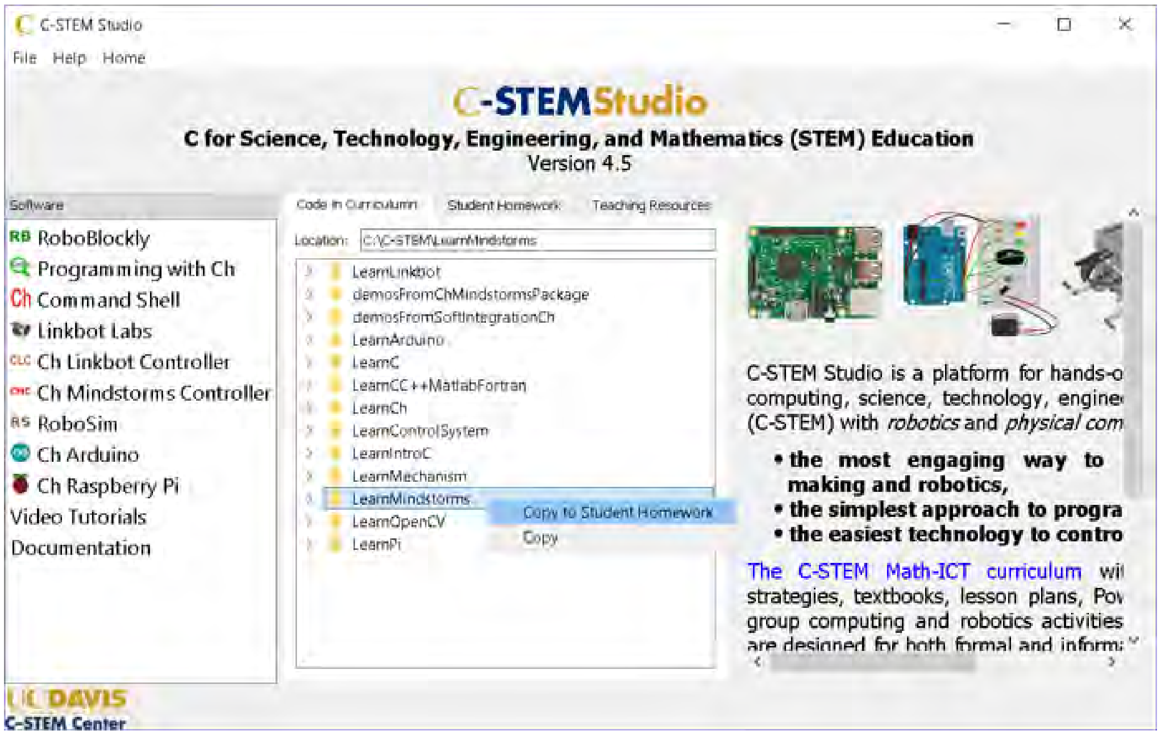
In this chapter, we will learn how to write computer programs in Ch to solve applied problems and control a Mindstorms.

3.1.1 Getting Started with ChIDE

An Integrated Development Environment (IDE) can be used to develop computer programs. ChIDE in Ch is an IDE to edit, debug, and run C/Ch/C++ programs. ChIDE can be conveniently launched by double clicking its icon on the desktop or in the C-STEM Studio shown in Figure 1.3. A layout of ChIDE is displayed in Figure 3.2, which also shows various terms used to describe ChIDE in this book.

3.1.2 Copy Code in Curriculum

All programs presented in this book are available in the C-STEM Studio through the menu “Code in Curriculum”, as shown in Figure 3.1. They are typically located in the folder `C:\C-STEM\LearnMindstorms` in Windows, `/opt/C-STEM/LearnMindstorms` in Mac OS X, and `/usr/local/C-STEM/LearnMindstorms` in Linux.



C-STEM Studio is a platform for hands-o computing, science, technology, engine (C-STEM) with *robotics* and *physical com*

- **the most engaging way to making and robotics,**
- **the simplest approach to progra**
- **the easiest technology to contro**

The C-STEM Math-ICT curriculum wi strategies, textbooks, lesson plans, Pov group computing and robotics activities are designed for both formal and inform

Figure 3.1: Copy the LearnMindstorms folder from “Code in Curriulum” in the C-STEM Studio.

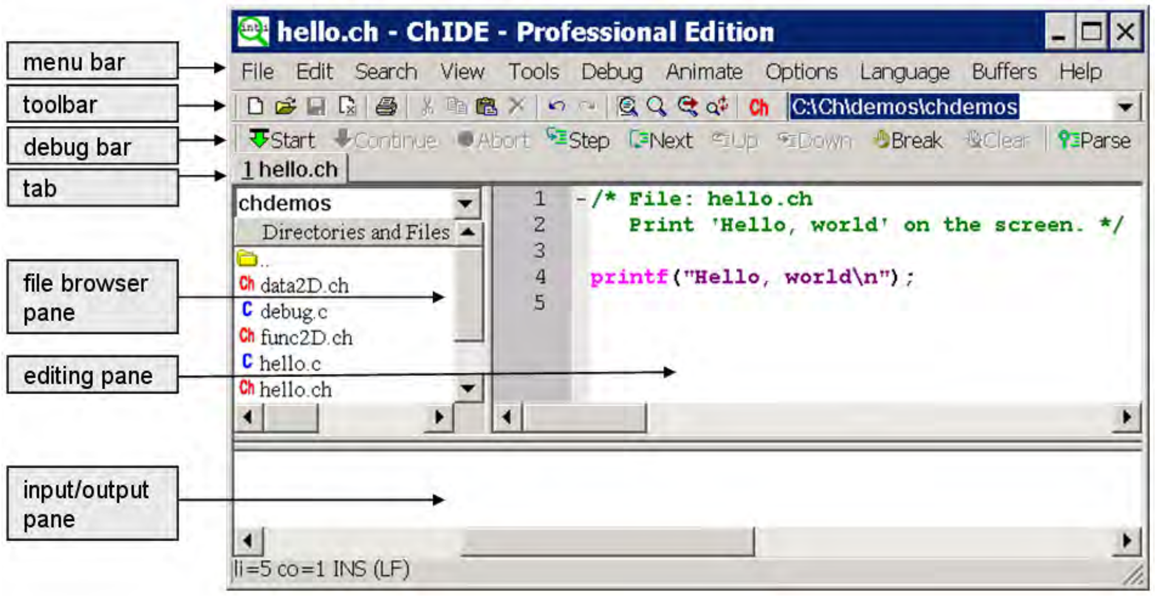
As programs in “Code in Curriculum” in C-STEM Studio are shared by all users in a computer lab, they cannot be modified. If a user would like to modify the sample programs in this book to solve the similar problems or better understand the programming and robotics concepts, the entire folder `LearnMindstorms` can be copied to the “Student Homework” folder. If you right click to copy the folder as shown in Figure 3.1. Then, click the “Student Homework” in C-STEM Studio to bring up the student homework folder with the copied “LearnMindstorms” folder.

3.1.3 The First Ch Program

Let’s get started with programming in Ch! We will write a simple program shown in Program 3.1. The program will display the following output on the screen when it is executed:

```
 Hello, world
```

To run the code in Program 3.1, the source code needs to be written first. ***Source code*** is plain text, which contains instructions of a program. If the text in Program 3.1 is typed in the editing pane in ChIDE, the program will appear colored due to syntax highlighting and with line numbers, as shown in Figure 3.2.



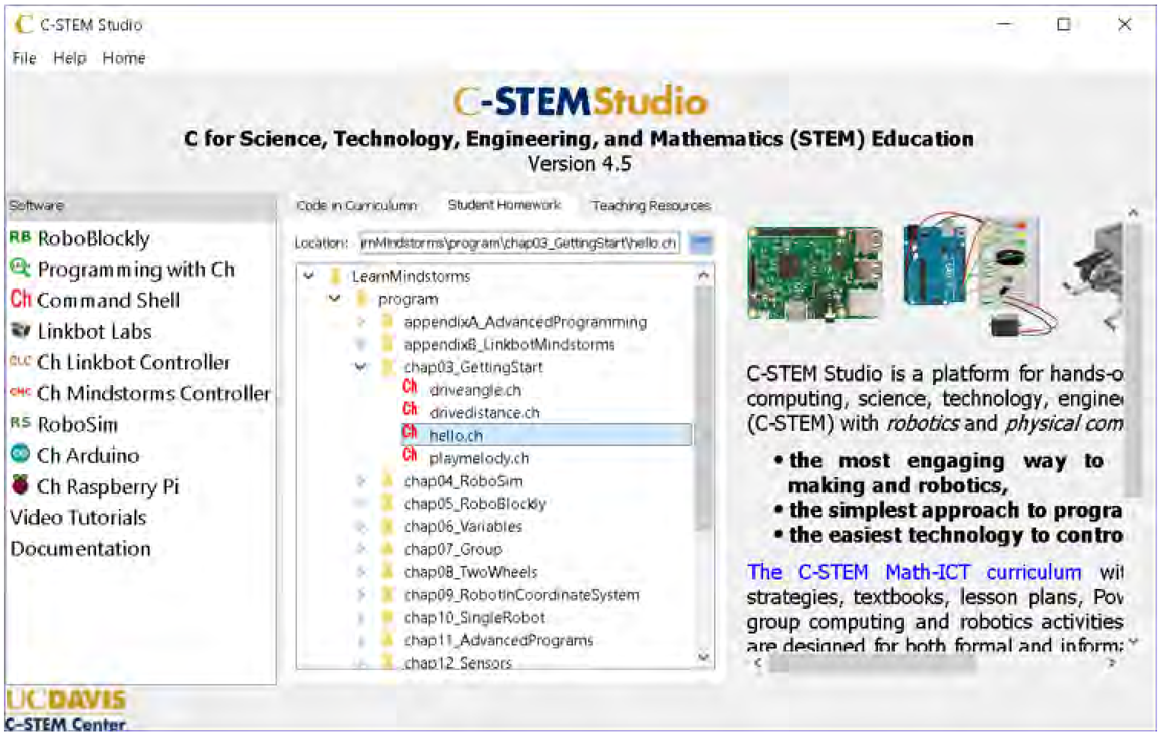
```
/* File: hello.ch
   Print 'Hello, world' on the screen. */

printf("Hello, world\n");
```

Figure 3.2: A layout and related terminologies in ChIDE.

```
/* File: hello.ch
   Print 'Hello, world' on the screen. */

printf("Hello, world\n");
```

Program 3.1: The first Ch program `hello.ch`.

We can also launch the program in ChIDE by double clicking the file `hello.ch` in this chapter as shown in Figure 3.3



C-STEM Studio is a platform for hands-o
computing, science, technology, engine
(C-STEM) with *robotics* and *physical com*

- **the most engaging way to making and robotics,**
- **the simplest approach to progra**
- **the easiest technology to contro**

The C-STEM Math-ICT curriculum wi
strategies, textbooks, lesson plans, Pov
group computing and robotics activities
are designed for both formal and inform:

Figure 3.3: Launch the program `hello.ch` in C-STEM Studio.

We will explain each line in Program 3.1 in detail. Contents that begin with `/*` and end with `*/` are comments. ***Comments*** are used to document a program to make the code more readable. When comments are processed by Ch, they are ignored and no action is taken relating to the comments. The first two lines, listed below, in Program 3.1 are comments.

```
/* File: hello.ch
   Print 'Hello, world' on the screen */
```

They document that the file name of the program is `hello.ch` and the purpose of the program is to display the message `Hello, world` on the screen.

A Ch program typically ends with ".ch", which is called the ***file extension***. A file name generally does not contain a space.

A ***function*** is the basic executable module in a program. Asking a function to perform its assigned tasks is known as ***calling*** the function.

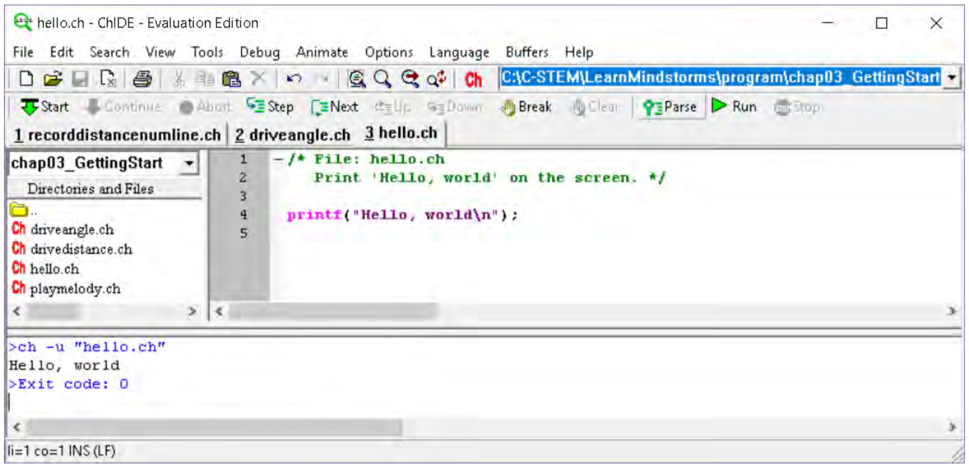
In the statement

```
printf("Hello, world\n");
```

The function **printf**() is used to display `Hello, world` on the screen. The symbol `\n` will be explained in section 3.1.8. Each statement in a program must end with a semicolon.

3.1.4 Opening Programs in ChIDE from Windows Explorer

In Windows, a program listed in the Windows explorer can be opened in the editing pane of ChIDE by clicking on the program. The program can also be opened in the editing pane by dragging and dropping it on to the ChIDE icon on the desktop.


hello.ch - ChIDE - Evaluation Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\C-STEM\LearnMindstorms\program\chap03_GettingStart
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 recorddistancenumline.ch 2 driveangle.ch 3 hello.ch
chap03_GettingStart
Directories and Files
driveangle.ch
drivedistance.ch
hello.ch
playmelody.ch
li=1 co=1 INS (LF)

```
/* File: hello.ch
   Print 'Hello, world' on the screen. */

printf("Hello, world\n");
```

```
>ch -u "hello.ch"
Hello, world
>Exit code: 0
```

Figure 3.4: Running the program inside the editing pane in ChIDE and its output.

.

3.1.5 Editing Programs

Text editing in ChIDE works similarly to most Windows or Mac text editors, such as Microsoft Word. As an example, open a new document by clicking the command `File->New` on the menu bar, or the first icon on the toolbar that looks like a little piece of paper with a folded corner, as shown in Figure 3.2.

You can save the document as a file named `hello.ch` by the command `File->Save As`. Follow the instruction and type the file name `hello.ch` to save as a new program. You can also right click the file name on the tab bar, located below the debug bar, and then select the command `Save As` to save the program.

3.1.6 Running Programs and Stopping Their Execution

Click `Run` on the toolbar, as shown in Figure 3.4, to execute the program `hello.ch`. This will cause the interpreter to read the code and provide an output on the bottom of the ChIDE window as shown in Figure 3.4. Pressing the function key `F2` will also execute the program. If you are editing a program, pressing F2 will save the edited program first and then run the saved program.

If the command execution has failed or execution is taking too long to complete, then the `Stop` command on the toolbar can be used to stop the program.

3.1.7 Output from Execution of Programs

The ***editing pane*** on the top is for writing and editing a program source file or any text file. The ***input/output pane*** is located below the editing pane, and is initially hidden. It can be made larger by dragging the divider between it and the editing pane. The output from the program is directed into the input/output pane when it is executed using the command `Run`, as shown in Figure 3.4. When the program `hello.ch` is executed, the input/output pane will be made visible if it is not already visible and will display the following three lines, as shown in Figure 3.4.

```
    >ch -u "hello.ch"  // use the command ch for Ch to execute hello.ch
    Hello, world       // the output from executing the program hello.ch
    >Exit code: 0      // display the exit code for the program
```

An exit code of 0 indicates that the program has terminated successfully. If a failure had occurred during the execution of the program, the exit code would be -1.
E Do Exercises 1, 2, 3, and 4 on page 29.

3.1.8 Newline Character

The symbol `\n` used in the function **printf**() in Program 3.1 means a ***newline character***. It instructs the computer to start writing on a new line, like the `Enter` key, which can be illustrated by changing the line

```
printf("Hello, world\n");
```

to

```
printf("Hello, world\nWelcome to Ch!\n");
```

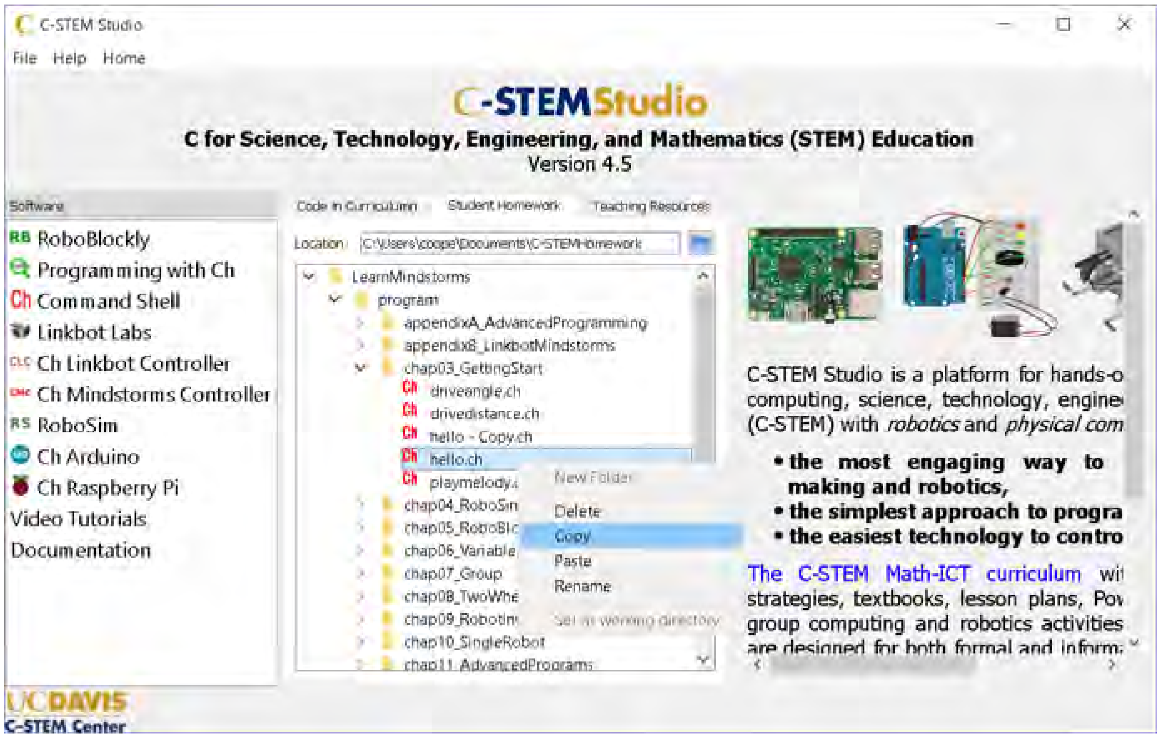
The output of the new program will become

```
Hello, world
Welcome to Ch!
```

After the newline character, the string `Welcome to Ch!` is displayed at the beginning of the next line on the screen.

3.1.9 Copying a Program to Another Program in C-STEM Studio

Unlike a calculator, an existing Ch program can be copied to a new file as another program conveniently. This process of creating a new program can save a lot of typing. Below are the step-by-step instructions on how to create a program `hello2.ch` to produce the output described in section 3.1.8. It copies the file `hello.ch` in Program 3.1 to a new program `hello2.ch` in C-STEM Stuio to solve the above problem.



C-STEM Studio is a platform for hands-o
computing, science, technology, engine
(C-STEM) with *robotics* and *physical com*

- **the most engaging way to**
  making and robotics,
- **the simplest approach to progra**
- **the easiest technology to contro**

The C-STEM Math-ICT curriculum wi
strategies, textbooks, lesson plans, Pov
group computing and robotics activities
are designed for both formal and inform

Figure 3.5: Copy the program `hello.ch`, paste, and rename to create a new program `hello2.ch`.

1. Right click the file `hello.ch` to bring up the menu as shown in Figure 3.5, and select “Copy” on the menu to copy the file into the buffer.
2. If you would like to copy the file in a different folder, click the folder where you would like the file to be copied to. Or skip this step if the file will be copied in the same folder.
3. Right click to bring up the menu as shown in Figure 3.5, and select “Paste” on the menu to copy the file from the buffer.
4. If the file with the same name already exists, a file with a new name, appendded with “- Copy”, will be created. For example, a copied file with the name “hello - Copy.ch” is created, as shown in Figure 3.5.
5. Right click to bring up the menu, and select “Rename” to rename the copied file with the new name “hello2.ch”.
6. Double click the file name `hello2.ch` and modify it with the statement below.

   ```
       printf("Hello, world\nWelcome to Ch!\n");
   ```

E Do Exercises 5 and 6 on page 30.

3.1.10 Correcting Errors in Programs

ChIDE can identify errors that occur in the source code and provide helpful responses that aid the user in finding and correcting these errors. To see this, we will create an error in the program `hello.ch` by changing the line

```
    printf("Hello, world\n");
```

to

```
    printf("Hello, world\n";
```

Notice that in the second statement the closing parenthesis is missing. When the program is executed, the results should look like the input/output pane in Figure 3.6. The line with incorrect syntax in the editing pane and the corresponding error message in the input/output pane will be highlighted with a yellow background. The first error message at the line

```
    ERROR: missing ')' before ';'
```

indicates that a closing parenthesis is missing before the semicolon ';'.

Because the program fails to execute, the exit code -1 is displayed at the end of the input/output pane as

```
    >Exit code: -1
```

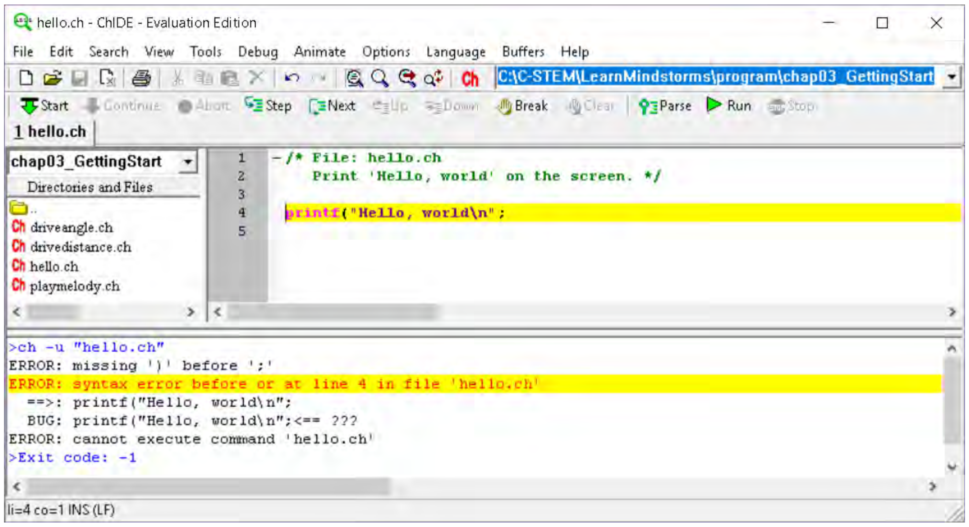
Errors in computer programs are called ***bugs***. The process of finding and reducing the number of bugs is called ***debug*** or ***debugging***. ChIDE is especially helpful for testing and debugging programs.

E Do Exercise 7 on page 30.

3.1.11 Summary

This section summarizes what you should have learned in this session.

1. **A Ch program has a file name with a file extension ”.ch”. A file name generally does not contain a space**.


hello.ch - ChIDE - Evaluation Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\C-STEM\LearnMindstorms\program\chap03_GettingStart
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 hello.ch
chap03_GettingStart
Directories and Files
driveangle.ch
drivedistance.ch
hello.ch
playmelody.ch
li=4 co=1 INS (LF)

```
/* File: hello.ch
   Print 'Hello, world' on the screen. */

printf("Hello, world\n";
```

```
>ch -u "hello.ch"
ERROR: missing ')' before ';'
ERROR: syntax error before or at line 4 in file 'hello.ch'
  ==>: printf("Hello, world\n";
  BUG: printf("Hello, world\n";<== ???
ERROR: cannot execute command 'hello.ch'
>Exit code: -1
```

Figure 3.6: The error line in output from executing program hello.c.

2. The comments in a Ch program begin with `/*` and end with `*/`.
3. Use the output function **printf**() to print a string.
4. Each statement in a Ch program ends with a semicolon.
5. Use the escape character `'\n'` as a newline character.
6. Use the integrated development environment ChIDE to edit and run Ch programs.
7. Edit, save, and run Ch programs in ChIDE.
8. Run a program by pressing the function key F2.
9. Copy a folder or file in C-STEM Studio.
10. Find the corresponding lines in a program with error messages and fix bugs in the program.

3.1.12 Terminology

bugs, calling the function, ChIDE, comment, copy program, debug, debugging, editing pane, error message, exit code, file extension, IDE, Integrated Development Environment, newline character, input/output pane, **printf**(), Run, source code, Stop

3.1.13 Exercises

1. Watch the video tutorial "Introduction to Ch and ChIDE" in http://c-stem.ucdavis.edu/studio/tutorial/.
2. Create a folder called `learnRobot` to keep Ch programs that you will develop. You may use an alternative folder name and location that your instructor specifies.
3. What is wrong with this line of code:

```
printf('cool!";
```

4. Write a program `cool.ch` to display

```
This is cool!
```

Based on the instructions described in section 3.1.9 to copy the program `hello.ch` to the program `cool.ch`. Save your program in the folder created in Exercise 2. The program calls the function **printf**() to display the output on the screen. Execute the program in ChIDE by the command `Run`.

5. Write a program `welcome.ch` to display

```
    Hello, world.
    Welcome to Ch!
    This is cool.
       by [your_name, today's date]
```

   Based on the instructions described in section 3.1.9 to copy the program `cool.ch`, developed in Exercise 4, to the program `welcome.ch`. The program calls the function **printf**() four times, one for each output line. Run the program in ChIDE.

6. Write a program `welcome2.ch` to display the same output as that from the program `welcome.ch` developed in Exercise 5. But, the program `welcome2.ch` shall call the function **printf**() *only once*. Based on the instructions described in section 3.1.9 to copy the program `welcome.ch` to the program `welcome2.ch`. Run the program in ChIDE.

7. Modify the program `welcome.ch` developed in Exercise 5 to introduce a bug by removing a closing parenthesis ')'. Run the modified program in ChIDE by pressing the function key F2. Find the line corresponding to the first error message in the editing pane. Then, fix the bug.

3.2 Drive Forward and Backward by Angle Relative to its Current Joint Position

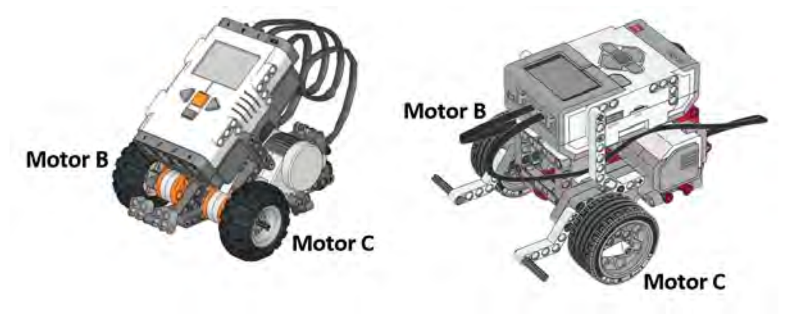


Figure 3.7: Mindstorms NXT and EV3 in two-wheeel vehicle configuration.

A Ch program can be developed to control a Mindstorms NXT or EV3, configured as a two-wheel vehicle using motors B and C as shown in Figure 3.7. Program 3.2 contains the code typically used for controlling a Mindstorms. We will explain the functionality of each statement in this robot program.

```
/* File: driveangle.ch
   Drive forward and backward for Mindstorms as a two-wheel vehicle */
#include <mindstorms.h>
CMindstorms robot;

printf("Here comes a robot!\n");

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(360);

/* drive backward by rolling two wheels for 360 degrees */
robot.driveAngle(-360);

printf("Cool!\n");
```

Program 3.2: The first program to control a Mindstorms by rolling two wheels.

A line that starts with a **#** has a special meaning, which depends on the symbol following it. The line

```
#include <mindstorms.h>
```

instructs Ch to include the contents of the header file **mindstorms.h** in the program. The contents made available via **#include** is called a ***header*** or ***header file***. This line appears in every Mindstorms program to allow control of the Mindstorms through the class **CMindstorms**. A class is a user defined data type in Ch. The symbol **CMindstorms** can be used to create a Mindstorms object. The line

```
CMindstorms robot;
```

creates the variable `robot` for controlling a Mindstorms. The statement also connects the variable `robot` to a Mindstorms that has been previously configured with the computer as described in Section 2.1 on page 8.

A class has functions associated with it. The functions associated with a class are called ***member functions***. For example, the function **robot.driveAngle()** or **driveAngle()** is a member function of the class **CMindstorms**. In Program 3.2, this member function of the class **CMindstorms** is called to move joints of a Mindstorms.

The line

```
printf("Here comes a robot!\n");
```

displays the following output in the input/output pane.

```
Here comes a robot!
```

A Mindstorms can be configured as a two-wheel robot. In this case, both joints 2 and 3 can rotate together to roll forward or backward. The member function **driveAngle**() causes both joints 2 and 3 to drive the Mindstorms forward. The syntax of the member function **driveAngle**() is as follows.

```
robot.driveAngle(angle);
```

The amount to roll the wheels forward relative to their current positions is specified by the argument `angle`. If the value of the argument of the member function **driveAngle**() is negative, it will drive a robot backward.

For example, Program 3.2 first drives the Mindstorms forward 360 degrees for both joints 2 and 3 by

```
robot.driveAngle(360);
```

Then, it drives the Mindstorms backward 360 degrees for both joints 2 and 3 by

```
robot.driveAngle(-360);
```

with a negative value -360 for the argument of the member function **driveAngle**().

Note that the prefix **drive** for a name of a member function is reserved for member functions to drive a Mindstorms configured as a two-wheel robot.

All member functions of the class **CMindstorms** for motion including **driveAngle()** expect input angles in degrees. In section 10.7, we will learn how to handle joint angles specified in radians.

The last line

```
printf("Cool!\n");
```

displays `Cool!` in the input/output pane.

After your computer is connected to a Mindstorms as described in the previous chapter, when Program 3.2 is executed, the following output will be displayed in the input/output pane first,

```
Here comes a robot!
```

Then, the Mindstorms will make a full rotation for both joints 2 and 3. Finally, the following output will be displayed in the input/output pane.

```
Cool!
```

E Do Exercise 1 on page 32.

3.2.1 Summary

This section summarizes what you should have learned in this session.

1. A Mindstorms program typically begins with the following statements.

   ```
   #include <mindstorms.h>
   CMindstorms robot;
   ```

   to declare the variable `robot` and connect it to a Mindstorms.

2. Call the **CMindstorms** member function

   ```
   robot.driveAngle(angle);
   ```

   to drive a Mindstorms forward or backward by rolling both joints 2 and 3 with the specified angle, relative to their current positions for both joints.

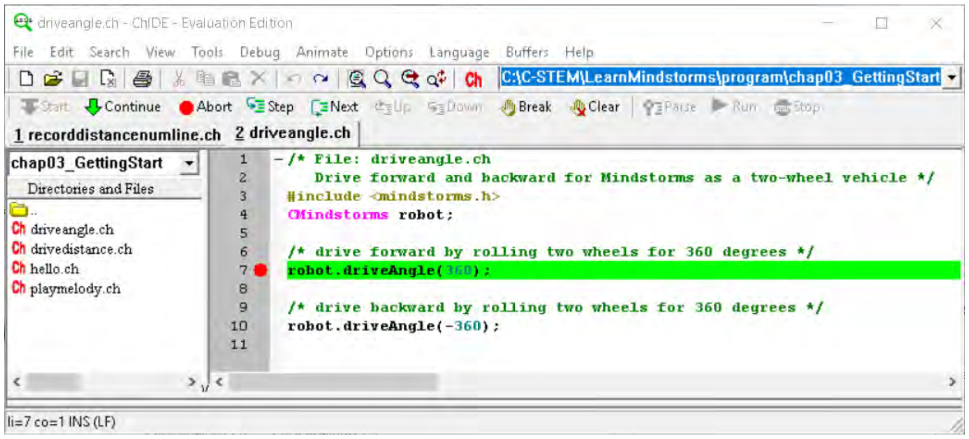
3. Joint angles in arguments of the **CMindstorms** member functions, such as **driveAngle**(), are specified in degrees.

3.2.2 Terminology

#include <mindstorms.h>, header, header file, class, **CMindstorms**, member function, relative position, **robot.driveAngle**(), drive forward, drive backward.

3.2.3 Exercises

1. Write a program `driveangle2.ch` to drive backward a Mindstorms by rolling joints 2 and 3 by 180 degrees, then drive it forward by rolling joints 2 and 3 by 360 degrees.


driveangle.ch - ChIDE - Evaluation Edition
File Edit Search View Tools Debug Animate Options Language Buffers Help
C:\C-STEM\LearnMindstorms\program\chap03_GettingStart
Start Continue Abort Step Next Up Down Break Clear Parse Run Stop
1 recorddistancenumline.ch 2 driveangle.ch
chap03_GettingStart
Directories and Files
driveangle.ch
drivedistance.ch
hello.ch
playmelody.ch
li=7 co=1 INS (LF)

```
/* File: driveangle.ch
   Drive forward and backward for Mindstorms as a two-wheel vehicle */
#include <mindstorms.h>
CMindstorms robot;

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(360);

/* drive backward by rolling two wheels for 360 degrees */
robot.driveAngle(-360);
```

Figure 3.8: Running the program `driveangle.ch` in debug mode.

3.3 Control a Robot in Debug Mode

When a program is executed in the debug mode by the command `Next` in ChIDE, the program will be executed line by line. The currently executed statement is highlighted in the green color. For example, Figure 3.8 shows that Program 3.2 is executed in the debug mode. The currently executed statement

```
    robot.driveAngle(360);
```

is highlighted in the green color. Until you click `Next` to execute the next statement, the joint angles of the robot will remain in their current goal positions.

When a program is executed in the debug mode, the command `Continue` can be clicked to continue the execution of the program until the program ends.

E Do Exercises 1 and 2 on page 33.

3.3.1 Summary

1. Execute a Mindstorms program line-by-line using the command `Next` on the debug bar in ChIDE while monitoring joint angles of a Mindstorms on the Robot Control Panel in CMC.
2. Use the command `Continue` to finish the execution of the remaining part of the program non-stop.

3.3.2 Terminology

Debug mode, Next, Continue,

3.3.3 Exercises

1. Watch the video tutorial "Debug in ChIDe" in http://c-stem.ucdavis.edu/studio/tutorial/.
2. Write a program `monitormotion.ch` to drive a Mindstorms with the following motion statements.

```
    robot.driveAngle(360);
    robot.driveAngle(-720);
```

   Run the program in ChIDE in debug mode with the command `Next` on the debug bar, as you monitor the change of joint angles on the Robot Control Panel in Ch Mindstorms Controller.

3.4 Drive a Distance for a Two-Wheel Robot

In this section, we will learn how to use the member function **driveDistance**() to control a Mindstorms as a two-wheel robot as shown in Figure 3.9. More advanced control of a two-wheel robot will be described in Chapter 8.

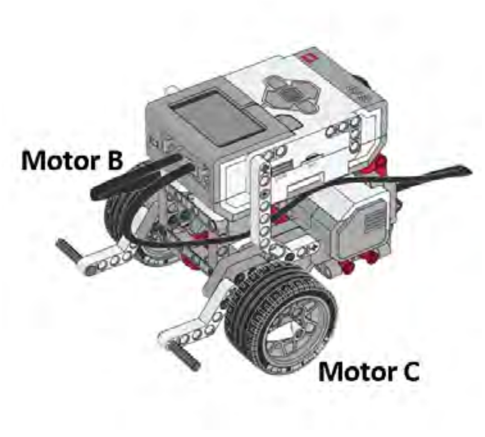


Figure 3.9: A two-wheel robot.

Like the member function **driveAngle**(), the member function **driveDistance**() causes both motors B and C (joints 2 and 3) to roll the Mindstorms forward. The syntax of the member function **driveDistance**() is as follows.

```
    robot.driveDistance(distance, radius);
```

Unlike the member function **driveAngle**(), the distance for the Mindstorms to drive forward is specified by the first argument `distance`. If the value of the first argument of the member function **driveAngle**() is negative, it will drive a robot backward. The radius of the two wheels, attached to the joints of the Mindstorms, is specified by the second argument `radius`. The units for both distance and radius must be the same. They can be inches, feet, centimeters, meters, etc.

```
/* File: drivedistance.ch
   Drive a robot as a two-wheel robot for a given distance. */
#include <mindstorms.h>
CMindstorms robot;

/* drive forward for 8 inches with the radius 1.1 inches for two wheels */
robot.driveDistance(8, 1.1);

/* drive backward for 5 inches with the radius 1.1 inches for two wheels */
robot.driveDistance(-5, 1.1);
```

Program 3.3: Moving a Mindstorms with a specified distance using **driveDistance()**.

For example, Program 3.3 drives a Mindstorms configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. The program drives the Mindstorms forward for 8 inches by the statement

```
    robot.driveDistance(8, 1.1);
```

Then, it drives the Mindstorms backward for 5 inches

```
    robot.driveDistance(-5, 1.1);
```

with a negative value -5 for the distance of the first argument of the member function **driveDistance**().
E Do Exercise 1 on page 35.

3.4.1 Summary

1. Call the **CMindstorms** member function

   ```
       robot.driveDistance(distance, radius);
   ```

   to drive a Mindstorms forward by the specified distance and radius for two wheels relative to its current position.

3.4.2 Terminology

robot.driveDistance(), distance, drive a distance, radius of a wheel.

3.4.3 Exercises

1. Write a program `drivedistance2.ch` to control a Mindstorms. The program calls the function **driveDistance**() to drive the robot forward for 6 inches, then drive the robot backward for -8 inches. next drive the robot forward for 2 inches. Assume the radius of wheels is 1.1 inches.

3.5 Play Melody

In addition to LED color, a Mindstorms contains a buzzer, which can be used to play melody and music notes. By default, EV3 is readily to play melody whereas the buzzer in NXT is set to mute.

To raise the volume on the NXT, press the grey arrow until you find Settings. Then select Settings by pressing the orange square. Once at Settings, click on the Volume with the orange square. To raise and lower the volume use the grey arrows — the maximum volume level is 4. Once you select the volume level you prefer, press the orange square one last time. Now you can hear the NXT play melodies.

The **CMindstorms** member function **playMelody()** can be called to play a melody. The general syntax of this member function is as follows.

```
    robot.playMelody(melody, speedFactor);
```

The first argument `melodoy` specifies the melody to be played. Table 3.1 lists commonly used melodies. Table C.1 in Appendix C lists all available melodies. The user can develop your own melodies as decribed in section 12.6 later. The second argument `speedFactor` is the speed factor, which determines how fast to play the melody. The value 1 is for the normal speed. If the value for the speed factor is larger than 1, the melody will be played faster than the normal speed. If the value for the speed factor is less than 1 but larger than 0, the melody will be played slower. For example, the value 2 doubles the speed whereas the value 0.5 is an half of the normal speed.

Table 3.1: Names of songs and their corresponding melody names in Ch.

Name of Song	Name of Melody in Ch
B-I-N-G-O	Bingo
Do Re Mi	DoReMi
Happy Birthday	HappyBirthday
The Ice Cream Truck Jingle	IceCream
Jingle Bells	JingleBells
Marry Had A Little Lamb	LittleLamb
Twinkle Twinkle Little Star	LittleStar
Merry Christmas	MerryChristmas
Old Mc Donald Had A Farm	OldMcDonald
A Typical Phone Ring Tone	RingTone
Row Your Boat	RowYourBoat
Dance Music	Techno
The Wheels On The Bus	WheelsOnTheBus
Mario Theme	MarioTheme
The Ants Go Marching	AntsGoMarching
The Ants Go Marching (high pitch)	AntsGoMarchingHighPitch

Program 3.4 plays the melody of Jingle Bells at different speed. The program first plays the melody at the normal speed by the statement

```
    robot.playMelody(JingleBells, 1);
```

Then, move the robot forward 5 inches by the member function

```
    robot.driveDistance(5, 1.1);
```

Afterwards, the melody is played twice as fast as the normal speed by the statement

```
    robot.playMelody(JingleBells, 2);
```

Finally, the melody is played again at an half of the original speed by the statement

```
    robot.playMelody(JingleBells, 0.5);
```

```
/* File: playmelody.ch
   Play a melody */
#include <mindstorms.h>
CMindstorms robot;

/* play "Jingle Bells" at the normal speed */
robot.playMelody(JingleBells, 1);

/* drive forward for 5 inches with the radius 1.1 inches for two wheels */
robot.driveDistance(5, 1.1);

/* double the normal speed */
robot.playMelody(JingleBells, 2);

/* half of the normal speed */
robot.playMelody(JingleBells, 0.5);
```

Program 3.4: Playing the JingleBells in a Mindstorms.

In section 12.6, you will learn how to create your own melodies for the member function **playMelody**() to play.
E Do Exercise 1 on page 37.

3.5.1 Summary

1. Call the **CMindstorms**() member function

   ```
   robot.playMelody(melody, speedFactor);
   ```

   play a melody.

3.5.2 Terminology

song, melody, **robot.playMelody()**.

3.5.3 Exercises

1. Write a program `playmelody2.ch`, based on Program 3.4, to first play the melodies of “Happy Birthday” at the normal speed and “Do Re Mi” at three times of the normal speed. Then, drive the robot forward 8 inches. Finally, play “Merry Christmas” at an half of the normal speed.

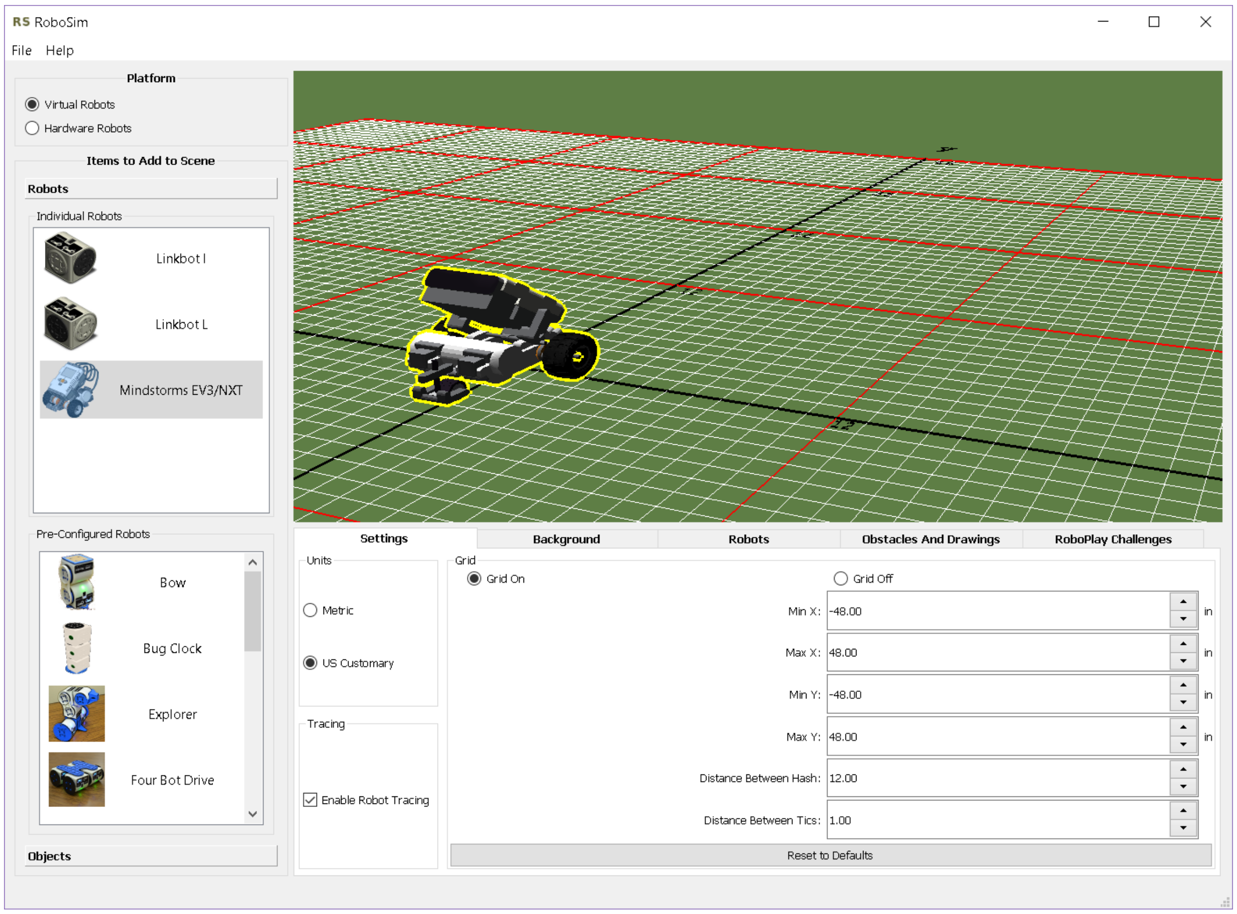
CHAPTER 4

Robot Simulation with RoboSim

RoboSim is a robot simulation environment for programming Lego Mindstorms NXT/EV3 and Linkbots. Most program that can control hardware Mindstorms in a two-wheel robot can be used to run virtual robots in RoboSim without any modification. Also all programs that can control virtual robots in RoboSim can run on hardware Mindstorms without any change.

4.1 RoboSim GUI

RoboSim can be conveniently launched by double clicking its icon RS on C-STEM Studio as shown in Figure 1.3. The RoboSim graphical user interface (GUI), shown in Figure 4.1, allows the user to change between hardware and virtual robots when a Ch robot program is executed. Different backgrounds, including RoboPlay Challenge boards, are available for RoboSim. Figure 4.1 shows a robot in the default background of outdoors. There is no save button within the GUI, all changes made are automatically saved.



Platform: [✓ Virtual Robots] Hardware Robots

Units: Metric [✓ US Customary]

Tracing: [✓ Enable Robot Tracing]

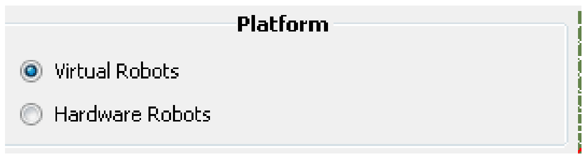
Grid: [✓ Grid On] Grid Off

Min X: -48.00 in
Max X: 48.00 in
Min Y: -48.00 in
Max Y: 48.00 in
Distance Between Hash: 12.00
Distance Between Tics: 1.00

Figure 4.1: The RoboSim GUI.

4.1.1 Platform

The **Platform** entry, as shown in Figure 4.2, allows the user to decide whether a Ch program controls the hardware or virtual robots. Each time a new Ch program is started, it will check the setup based on this entry. For a Ch robot program to control a virtual robot, check the box for **Virtual Robots**. If the box for **Hardware Robots** is checked, a Ch program will control the physical hardware robots.



Platform: [✓ Virtual Robots] Hardware Robots

Figure 4.2: The entry for selecting a simulation or hardware platform.

4.1.2 Units

Simulations within RoboSim can be run either in **US Customary** units consisting of inches, degrees, and seconds or **Metric** units with centimeters, degrees, and seconds. Changing units will effect the grid spacing drawn beneath the robots and the spacing between robots. Changing between these two options will change the labels within the GUI to indicate the units being used.

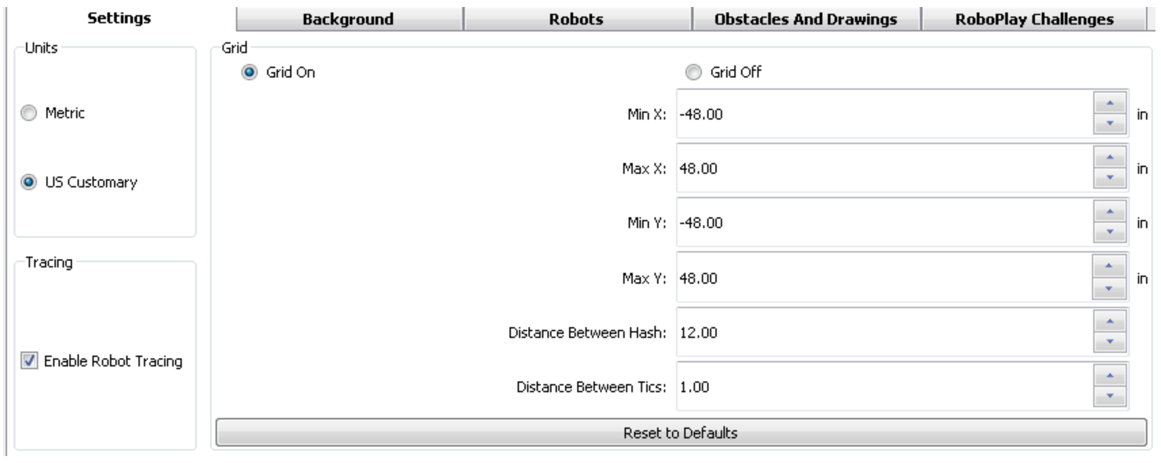
4.1.3 Tracing

Tracing where robots have been can be enabled by selecting the check box “Enable Robot Tracing”, as shown in Figure 4.1. When the tracing is enabled, green lines for robot trajectories will be drawn for each robot. Once a simulation is running, the tracing line can be enabled or disabled by pressing the ’t’ key.

4.1.4 Grid Configuration

To be able to see how far robots have moved, a grid is enabled under the robots. There are six options to alter the layout of the grid lines under the **Grid Configuration**. The minimum and maximum extends of the grid for both the X and Y directions can be specified individually. Rectangular grids of any size can be created in any of the quadrants. Hashmarks are the red lines drawn within the configuration images. By default, the distance between two hashmarks is 12 inches in US Customary units and 50 centimeters in Metric units. Tics are the most frequent lines drawn in a light gray. By default, the distance between two tics is 1 inch in US Customary units and 5 centimeters in Metric units.

Switching between US Customary and Metric units will change these default values to logical starting points for the metric system. The ’Reset to Defaults’ button will allow the default values for both US Customary and Metric to be reinstated after they have been changed. Depending upon which units are currently selected from Section 4.1.2, either the US Customary defaults, shown in Figure 4.3, or the Metric defaults, as shown in Figure 4.4, will be set.



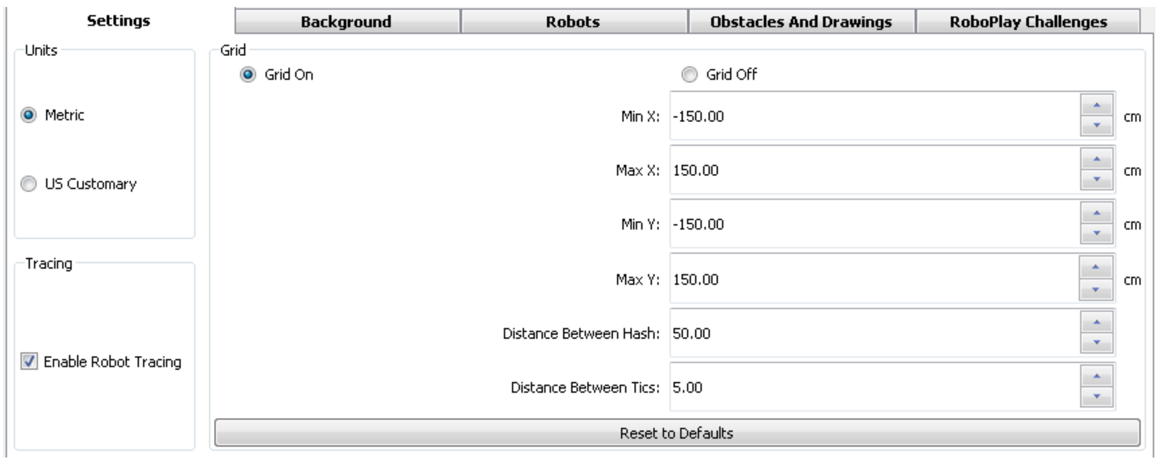
Units: Metric [✓ US Customary]

Tracing: [✓ Enable Robot Tracing]

Grid: [✓ Grid On] Grid Off

Min X: -48.00 in
Max X: 48.00 in
Min Y: -48.00 in
Max Y: 48.00 in
Distance Between Hash: 12.00
Distance Between Tics: 1.00

Figure 4.3: The default grid spacing in the US Customary units.



Units
[● Metric] US Customary

Tracing
[✓] Enable Robot Tracing

Grid
[● Grid On] Grid Off

Min X: -150.00 cm
Max X: 150.00 cm
Min Y: -150.00 cm
Max Y: 150.00 cm
Distance Between Hash: 50.00
Distance Between Tics: 5.00

Figure 4.4: The default grid spacing in the Metric units.

4.1.5 Individual Robot Configuration

Initial robot configurations can either be done through the **Individual Robots** or **Pre-Configured Robots** section for Linkbots. In the RoboSim GUI, the user can double click or drag a robot in the **Individual Robots** section shown in Figure 4.5 to the robot list shown in Figure 4.6 and scene. The robot list shown in Figure 4.6 has options to allow robots to be positioned within the RoboSim scene either with or without wheels but not attached to each other.

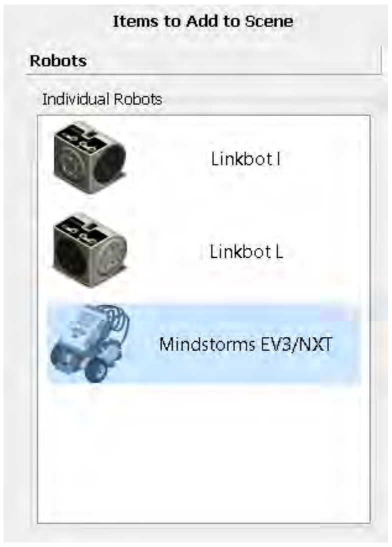
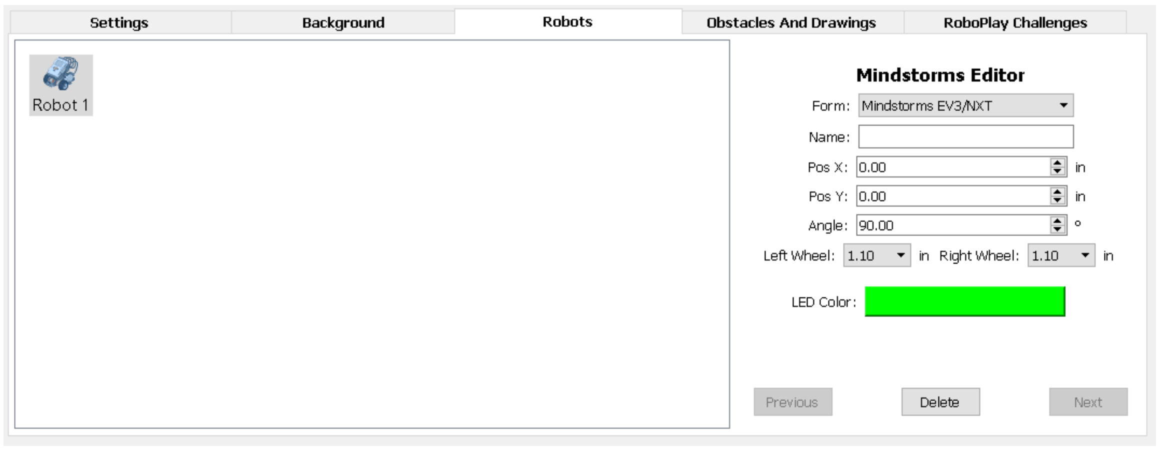


Figure 4.5: Individual robot selection.



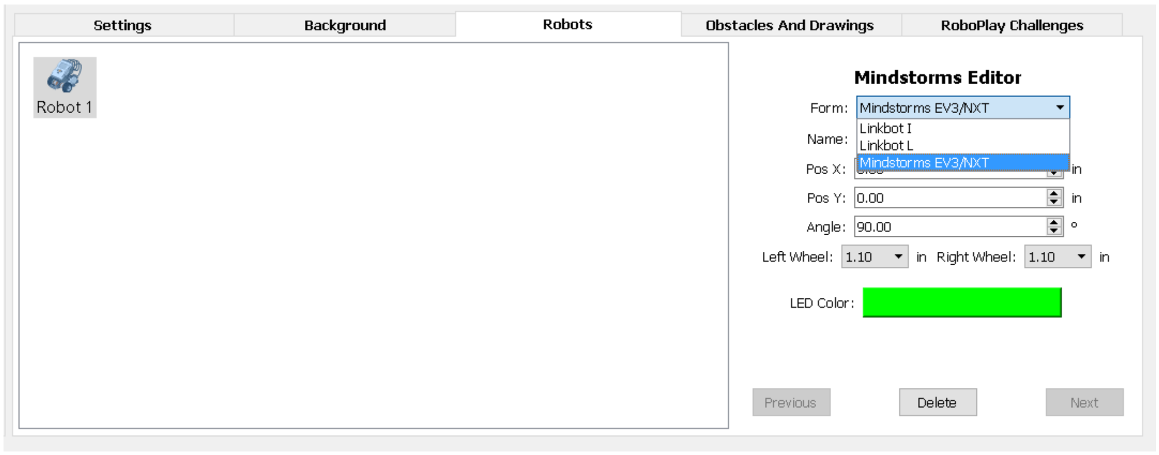
Form: Mindstorms EV3/NXT
Name:
Pos X: 0.00 in
Pos Y: 0.00 in
Angle: 90.00 °
Left Wheel: 1.10 in Right Wheel: 1.10 in
LED Color: 🟩

Figure 4.6: Individual robot configuration dialog.

The user can specify the x and y coordinates as well as the orientation angle of a virtual robot. Initially, the individual robot list is empty, but it can be populated by double clicking or dragging a robot in the **Individual Robots** to the individual robot list. Double clicking a robot each time will add a robot into the RoboSim, each offset from the previous one in the x-direction by 6 inches or 15 centimeters depending upon the units selected. The order within the robot list will be the order in which the robots will be read into the simulation program.

Robot Type

There are three options for robot type available. Linkot-I, Linkbot-L, Mindstorms NXT/EV3. The options are presented in a drop down menu as shown in Figure 4.7.



Form: Mindstorms EV3/NXT
Name:
Pos X: in
Pos Y: 0.00 in
Angle: 90.00 °
Left Wheel: 1.10 in Right Wheel: 1.10 in
LED Color: 🟩

Figure 4.7: Picking a robot type.

Robot Position

Both x and y coordinates can be chosen independently for each robot.

Robot Angle

The rotation angle from the x-axis can be used for changing the direction of the movement for the robot or the orientation of two robots respective to each other.

Wheels

Since so many times the robots are run with two wheels, a drop down menu is provided to select different wheel sizes. Table 4.1 lists the radii of the wheels typically used with Mindstorms.

Wheel Radius
1.1 inches / 2.79 centimeters
1.61 inches / 4.09 centimeters

Table 4.1: Wheel sizes.

Customized wheel sizes are not available for RoboSim v2.0.

LED Color

The LED color, which is also the color for the traced trajectory of a robot, can be set.

Delete

A robot can be deleted from the RoboSim by clicking the ’Delete’ button.
E Do Exercises 1 and 2 on page 44.

4.1.6 Summary

1. Select simulation or hardware mode in a RoboSim GUI for controlling robots from a Ch program.
2. Select US Customary units with inches, degrees, and seconds or Metric units with centimeters, degrees, and seconds for RoboSim.
3. Add robots to the RoboSim. The information for a robot includes robot type (Mindstorms EV3/NXT, Linkbot I, or Linkbot-L), the x and y coordinates as well as the orientation angle with respect to the x-axis for the robot, attached wheels of different sizes to the robot.
4. Remove a robot from the RoboSim
5. Set grid lines in the RoboSim scene with the US Customary units or Metric units.
6. Enable or disable the tracing of robot trajectories.

4.1.7 Terminology

RoboSim, RoboSim GUI, simulation, units, US Customary units, Metric units. robot type, x and y coordinates, orientation, grids for x and y coordinate systems, tracing robot trajectory.

4.1.8 Exercises

1. Watch the video tutorial RoboSim in http://c-stem.ucdavis.edu/studio/tutorial/.

2. (a) Launch a RoboSim GUI.
   (b) Add a two-wheel Mindstorms robot to the RoboSim at the x and y coordinates (6, 0) inches with an orientation angle of 30 degrees with respect to the x-axis, attach wheels with the radius of 1.1 inches to the robot.
   (c) Set the x and y coordinate system on the RoboSim scene. The total distance for x and y directions are 48 inches (4 feet) each. The distance between each Hashmark is 6 inches. The distance between each tics is 1 inch.
   (d) Track the robot trajectory when the robot moves.

4.2 Run a Ch Program with RoboSim

Once the simulation environment has been configured with the RoboSim GUI in Section 4.1, the user can run Ch programs in ChIDE to control the virtual robots. The RoboSim GUI should remain open while simulating robots. Once it is closed, the system will revert to hardware mode. The RoboSim scene with virtual robots for each simulation are created upon running a Ch program. For example, when the Ch program `driveangle3.ch`, listed in Program 4.1, is executed, a RoboSim scene shown in Figure 4.8 will be displayed. The message

```
Paused: Press any key to start
```

is displayed in the RoboSim scene to reminder the user that the virtual robot will not move until the user presses any key on the keyboard. This gives the user an opportunity to examine the RoboSim scene before the motion begins.

```
/* File: driveangle3.ch
   Drive forward for Mindstorms as a two-wheel vehicle */
#include <mindstorms.h>
CMindstorms robot;

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(360);
```

Program 4.1: Moving a Mindstorms forward by angle.

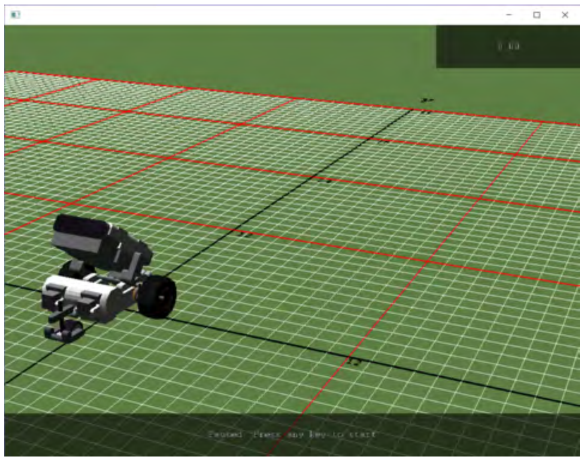


Figure 4.8: A RoboSim scene with a virtual robot at its starting position.

While a robot is moving in the RoboSim scene, the user can press any key to pause the motion of the robot. When the motion is paused, the message

```
Paused: Press any key to restart
```

will be displayed in the RoboSim scene. The user can press any key to restart the motion.

When the user presses the ’t’ key, the robot trajectory is traced in a green line in the RoboSim scene as shown in Figure 4.9.

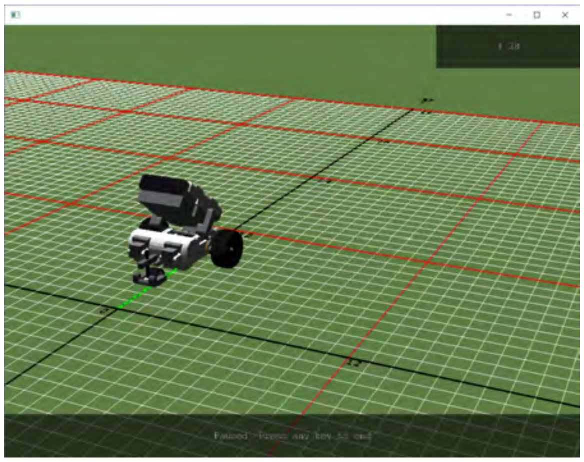


Figure 4.9: A RoboSim scene with a virtual robot and its trajectory traced.

E Do Exercise 1 on page 46.

The default green color for both LED and trajectory of a robot can be changed by the member function **setLEDColor**() in a program as shown in Program 4.2.

```
/* File: setcolor3.ch
   Change the color of the LED and trajectory of the robot to red */
#include <mindstorms.h>
CMindstorms robot;

/* change the color of the LED and trajectory to red */
robot.setLEDColor("red");

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(360);
```

Program 4.2: Change the color of the LED and trajectory of a robot to red.

When the program is finished, the message

```
Paused: Press any key to end
```

will be displayed in the RoboSim scene. Pressing any key, the RoboSim scene will disappear.
E Do Exercise 2 on page 46.

4.2.1 Summary

1. Run a Ch program to control a virtual robot in RoboSim.
2. Press the ’t’ key to trace the robot trajectory.
3. Press any key to pause and restart the motion of a robot in the RoboSim scene.

4.2.2 Terminology

RoboSim scene, tracing robot trajectories.

4.2.3 Exercises

1. (a) Write a Ch program `driveangle5.ch` to driving a Mindstorms forward by rotating both joints 2 and 3 for 720 degrees.
   (b) Based on the setup on the RoboSim GUI described in Exercise 2 on page 44, run the program `driveangle5.ch` to simulate the motion of the robot in RoboSim.
   (c) When the robot is moving in the RoboSim scene, press a key on the keyboard to pause the motion of the robot. Then, press a key to restart the motion.
   (d) When the robot is moving in the RoboSim scene, press the ’t’ key to toggle the tracing of the robot trajectory. Press the ’n’ key to toggle the tgrid numbering.
2. Modify the program `driveangle5.ch` in Exercise 1 as the program `setcolor4.ch` to change the color of the LED and trajectory to blue.
3. Run the program `drivedistance2.ch`, developed in Exercise 1 on page 35, with a virtual robot in RoboSim.

4.3 Interact with a RoboSim Scene

The user can interact with a RoboSim scene through the keyboard and mouse.

4.3.1 Keyboard Input

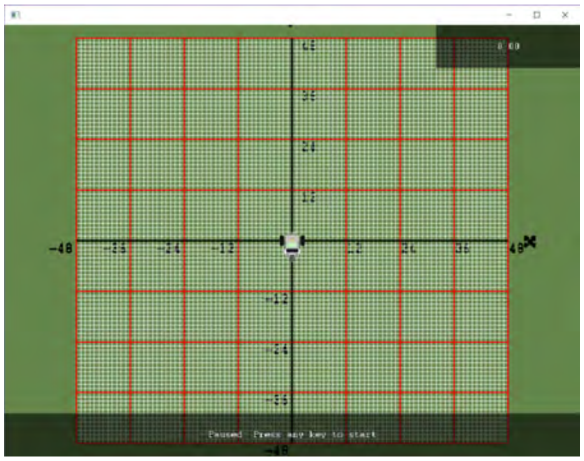
The RoboSim scene responds to keyboad input as outlined in Table 4.2.

key	action
1	set the camera to the default view
2	set the camera to the overhead view
c	Center the current robot in the view
n	Toggle grid line numbering
r	Toggle robot visibility and enable tracing
t	Toggle robot tracing
any other key	Pause and restart the motion

Table 4.2: Keyboard input for the RoboSim scene.

As described in the previous sections, the ’t’ key will toggle the tracing of robot trajectories. In addition the ’t’ key, a few other keys can be used to change the view of the RoboSim scene.

There are two views available to the user. The default view, which can be toggled with the ’1’ key, is from behind the robots looking into the first quadrant. This view can be seen in all RoboSim scene screenshots in this book, except for Figure 4.10 which shows the overhead view. The ’2’ key moves the camera directly above the origin looking down on the scene creating a 2D viewpoint of the robots.

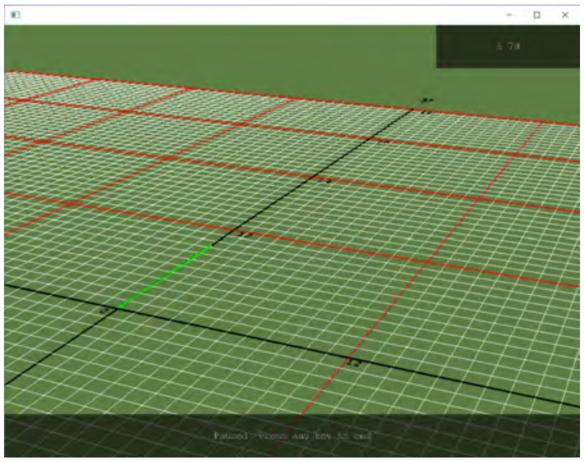


Paused. Press any key to start

Figure 4.10: A RoboSim scene with the overhead viewing angle.

The ’n’ key allows the user to toggle the display of the grid numbering. X and Y numbering is by default enabled and given for every hashmark on the grid.

The ’r’ key will toggle the display of virtual robots or robot trajectories. This feature is useful when the user would like to view a trajectory traced by a robot without the virtual root blocking the trajectory. When two virtual robots collide in a RoboSim scene, the program will stop working properly. However, without showing the virtual robots, the collision will not happen. This is useful for solving math problems such as two robots intercepting. Figure 4.11 shows a RoboSim scene with a traced robot trajectory only, without the robot displayed.



```
Paused - Press any key to end
```

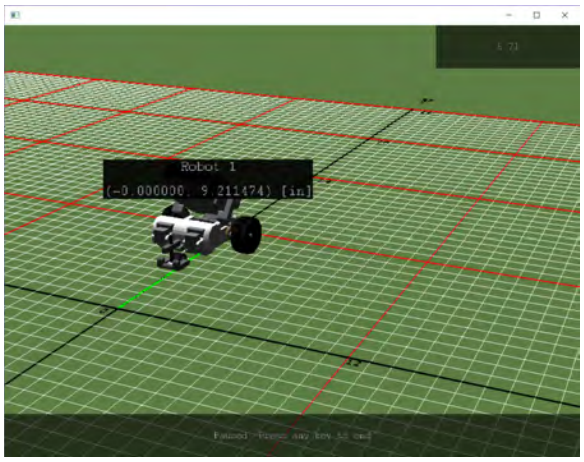
Figure 4.11: A RoboSim scene with a traced robot trajectory only.

As described in the previous section, the motion of robots in the RoboSim scene can be paused and restarted by pressing any other key on the keyboard.

4.3.2 Mouse Input

Clicking on a robot in a RoboSim scene will enable a pop up which displays the robot number and the current position of the robot, as shown in Figure 4.12 with the position (0, 7.672145) inches for the x and y coordinates for robot 1.

Clicking again, the displayed position for the robot will disappear.



```
Robot 1
(-0.000000, 9.211474) [in]
```

```
Paused - Press any key to end
```

Figure 4.12: A RoboSim scene with a virtual robot and its position displayed.

The user can execute a Ch robot program in debug mode in ChIDE, line by line, with the command

`Next`, as described in section 3.3. At the end of each motion statement, the user can click the robot in the RoboSim scene to obtain the x and y coordinates of the robot. The ability to obtain the x and y coordinates of a robot during its motion along a trajectory can be very useful for learning many math concepts. For example, the Ch program `driveangle4.ch`, listed in Program 4.3, drives forward a Mindstorms twice by calling the member function **driveAngle**() twice, the user can run the program in ChIDE in debug mode to find the distance traveled by each call. When the first **driveAngle**() is finished, the user can click on the robot to find the x and y coordinates. The y coordinate is the distance traveled by the robot.

```
/* File: driveangle4.ch
   Drive forward twice for Mindstorms as a two-wheel vehicle */
#include <mindstorms.h>
CMindstorms robot;

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(360);

/* drive forward by rolling two wheels for 360 degrees again */
robot.driveAngle(360);
```

Program 4.3: Driving a Mindstorms forward by calling the member function **driveAngle**() twice.

The mouse can be used to move the camera around the scene. Holding the left mouse button and dragging the mouse pans the camera as outlined in Table 4.3. Holding the right mouse button and dragging the mouse enables scaling of the view by zooming in and out. Holding both left and right mouse buttons and dragging changes the location of the camera within the scene.

The ground plane is for reference only. The ground plane will disappear when viewing the robots from below so that the user can inspect the movement from all angles.

button	**action**
Hold left mouse button and drag	Rotate camera
Hold right mouse button and drag	Zoom in and out
Hold both left and right buttons, and drag	Pan around scene
Click on a robot	Display the robot position

Table 4.3: Mouse input for the RoboSim scene.

E Do Exercise 2 on page 50.

4.3.3 Summary

1. Press the 'r' key to toggle the robot visibility and tracing the robot trajectory.
2. Hold the left mouse button and drag the mouse to have different view points.
3. Hold the right mouse button and drag the mouse to zoom in and out.
4. Hold both left and right mouse buttons, and drag the mouse to scale the scene.
5. Click on a robot to display the x and y coordinates of the robot.

4.3.4 Terminology

x and y coordinates for a robot.

4.3.5 Exercises

1. (a) Run the Ch program `driveangle5.ch` developed in Exercise 1 on page 50 in RoboSim. When the robot is moving in the RoboSim scene, press the key 'r' to toggle the visibility of the robot. When the robot is finished its motion, press the key 'r' to toggle the visibility of the robot.
   (b) Hold the left mouse button and drag the mouse to have different view points.
   (c) Hold the right mouse button and drag the mouse to zoom in and out.
   (d) Hold both left and right mouse buttons, and drag the mouse to scale the scene.
   (e) Click on a robot to display the x and y coordinates of the robot.

2. Write a Ch program `driveangle6.ch` to drive a Mindstorms forward by rotating both joints 2 and 3 for 720 degrees using the member function **driveAngle**(). Then, dirve the robot backward by rotating both joints 2 and 3 for 360 degrees using the member function **driveAngle**() with a negative value for its argument. Run the program in ChIDE in debug mode by clicking the command `Next` on the ChIDE. Click on the robot to obtain the positions of the robot when the robot stops its driving forward and backward.

CHAPTER 5

Using Variables and Generating Robot Programs Using RoboBlockly

RoboBlockly, available at http://www.roboblockly.org, is a web-based robot simulation environment for learning coding and math. Based on Google Blockly, it uses a simple puzzle-piece interface to program virtual Linkbot and Lego Mindstorms NXT/EV3 for beginners to learn robotics, computing, science, technology, engineering, and math (C-STEM). Blocks can be executed in debug mode step-by-step.

The RoboBlockly curriculum includes student self-guided Hour of Code, Robotics, Coding, and various theme-based activities. The teacher-lead 1st to 9th grade specific Math Activities are Common Core State Standards -Mathematics compliant. The related Teacher's Notes and comprehensive Teacher Resource Packets in PDF files for all these activities along with many video tutorial lessons help the users learn robotics, coding, and math.

Block programs can control Linkbot and and NXT/EV3 directly through the RoboBlockly program RB launched within C-STEM Studio. Constructing programs with blocks also generates C++ code that can be readily launched and run in Ch without any modification to control hardware Linkbot and Lego Mindstorms NXT/EV3, or virtual Linkbot and NXT/EV3 with RoboSim. Users can easily share the saved RoboBlockly code for collaboration and learning. Users can also create and share their Board with different background for obstacle courses.

Roboblockly can be used with any modern web browser on any computing devices including laptops, tablets, iPads, and smartphones, and is available in multiple languages.

In this chapter, we will first learn how to use variables in coding, then learn how to conduct robot simulation and generate robot programs using RoboBlockly.

5.1 Use Variables and Drive a Distance

Variables are often used in solving problems with unknown values. Variables are also a powerful tool available to programmers. Using variables to represent mathematical notation makes a program easier to modify and read. They can be used to solve complicated problems. They can also be used to obtain the information from the user at the runtime for interactive computing. Variables are also commonly used in robot programming for solving applied problems.

However, a variable has to be declared and associated with a proper data type before it can be assigned a value. In this chapter, declaration and use of variables involving commonly used data types for robot programming are described. We will also learn how to move a robot with the specified distance and radius for two wheels.

Table 5.1: Commonly used data types and their usage.

Data Type	Usage	Examples
double	decimals	`123.4567`
int	integers	`12`

5.1.1 Declaration of Variables and Data Type double for Decimals

An *identifier* is a sequence of lowercase and uppercase letters, digits, and underscores. A variable has to be declared before it can be used inside a program. A variable is declared by specifying its data type and identifier in the form

```
    type name;
```

where `type` is one of the valid data types, such as **double** and **int**, and `name` is a valid identifier. For example, the statements

```
    double t;               // the traveling time in seconds
    double distance;        // distance traveled
```

declare variables `t` and `distance` of **double** type. In this case, **double** is a keyword as a declarator for a data type and `t` and `distance` are identifiers as variable names. The symbol `//` comments out any subsequent text located on the same line.

The difference between lowercase and uppercase letters is important. In other words, variables are case sensitive. The initial character of an identifier must not be a digit. A reserved word, such as **double** and **for**, cannot be used as an identifier. Using meaningful and consistent identifiers for variable names makes a program easier to understand, develop, and maintain. A variable name typically uses lowercase letters. Table 5.2 shows some invalid identifiers.

Table 5.2: Examples of invalid identifiers.

Invalid identifier	Reason
`int`	reserved word
`double`	reserved word
`for`	reserved word
`2times`	starts with a digit
`integer#`	character # not allowed
`girl&boy`	character & not allowed
`class1+class2`	character + not allowed

Multiple variables of the same type can be declared in a single statement by a list of identifiers, each separated by a comma, as shown below for two variables `t` and `distance` of **double** type.

```
   double t, distance;  // declare variables t and distance
```

The names `x, y, z, length, width, radius`, `speed`, and `distance` are used in common practice for variables of **double** type to hold decimal numbers.

5.1.2 Initialization

Assigning a value to a declared variable for the first time is called ***initializing the variable***. You can initialize a variable in the same statement in which it is declared or you can initialize it in a separate statement. For example:

```
   double t = 5.5;    // declare t double type and initialize it with 5.5
```

and

```
   double t;          // declare t double type
   t = 5.5;           // initialize t with 5.5
```

accomplish the same goal of declaring a variable `t` of double type initializing it to 5.5.

5.1.3 Data Type int for Integers

Variables of **double** type can store decimals. Integers can be stored in a variable of **int** type. For example;

```
   int i = 2; // declare variable i of int type and initialize it with 2
   int n;     // declare variable n of int type
   n = 4+i;   // assign n with 4+i
```

The names `i, j, k, m, n, num`, and `count` are used in common practice for variables of **int** type.

Application: Using a Variable for Joint Angles

Problem Statement:
Write a program `angle.ch` to drive a Mindstorms as a two-wheel vehicle forward for 360 degrees for two wheels relative to their current positions, then backward for 360 degeees for two wheels. Use a variable to hold the joint angle.

Based on Program 3.2, we can develop the program `angle.ch` in Program 5.1. As pointed out in the previous sections, a robot program typically begins with the following statements.

```
   #include <mindstorms.h>
   CMindstorms robot;
```

to declare the variable `robot` and connect it to a Mindstorms.

Since joint angles are decimal values, in this program, the variable `angle` of **double** type is declared and assigned with the joint angle by the statement below.

```
   double angle =  360; // declare variable 'angle' for joint angle of two wheels.
```

This variable is used as an argument of the member function **driveAngle**().

The statement

```
    robot.driveAngle(angle);
```

moves joints by the angle specified in the argument `angle`. Program 5.1 behaves the same as Program 3.2.

```
/* File: angle.ch
   Use a variable to hold joint angles */
#include <mindstorms.h>
CMindstorms robot;
double angle = 360;   // declare variable 'angle' for joint angle of two wheels.

/* drive forward by rolling two wheels for 360 degrees */
robot.driveAngle(angle);

/* drive backward by rolling two wheels for 360 degrees */
robot.driveAngle(-angle);
```

Program 5.1: Using a variable for joint angles.

E Do Exercise 1 on page 55.

Application: Using Variables for Distance and Radius

Problem Statement:
Write a program `distance.ch` to drive a Mindstorms as a two-wheel vehicle forward for 5 inches, then backward for 7 inches. Use variables to hold the distance and radius.

Based on Program 3.3, we can develop the program `distance.ch` in Program 5.2.
Since the distance and radius are decimal values, variables `distance` and `radius` of **double** type are declared and assigned with the values by the statements below.

```
    double distance=5;  // the distance of 5 inches to drive forward
    double radius=1.1; // the radius of 1.1 inches of the two wheels of the robot
```

These variables are used as arguments of the function **driveDistance**().
The statement

```
    robot.driveDistance(distance, radius);
```

drives the robot forward by the distance specified in the first argument. The statement

```
    robot.driveDistance(-distance-2, radius);
```

drives the robot backward by 7 inches as the value of the expression $-distance - 2$ is -7.

```
/* File: distance.ch
   Drive a robot as a two-wheel robot for a given distance. */
#include <mindstorms.h>
CMindstorms robot;
double distance=5;  // the distance of 5 inches to drive forward
double radius=1.1; // the radius of 1.1 inches of the two wheels of the robot

/* drive forward for 5 inches with a specified radius of wheels */
robot.driveDistance(distance, radius);

/* drive backward for -7 inches with a specified radius of wheels */
robot.driveDistance(-distance-2, radius);
```

Program 5.2: Using variables for the distance and radius.

E Do Exercise 2 on page 55.

5.1.4 Summary

1. Reserved words. The words **int** and **double** are reserved words in Ch.
2. The data type **double** is for decimal numbers. Variable names `x`, `y`, and `z` are usually used for **double** type.
3. The data type **int** is for integers. Variable names `i, j, k, n,` and `m` are usually used for **int** type.
4. Learn how to use variables and variable names (identifiers).
5. Before a variable is used, it has to be declared with the data type **double** or **int**.
6. Choose descriptive names for variables.
7. Variable names are case sensitive. `x` and `X` are two different variable names.
8. Variables can be used in algebraic expressions.
9. Add a comment in a program starting with `//` until the end of the line.
10. Use the operators '+', '-', '*', and '/' for the arithmetic operations of addition, subtraction, multiplication, and division, respectively.

5.1.5 Terminology

algebraic equations, case sensitive, declare variables, **double**, evaluation, identifier, initialization, initialize variables, **int**, name, reserved words, variable

5.1.6 Exercises

1. Write a program `angle2.ch` to control a Mindstorms. The program shall use a variable `angle` to hold a joint angle of 720 degrees. The program calls the function **driveAngle**() with the variable `angle` to move the robot forward.

2. Write a program `distance2.ch` to control a Mindstorms. The program calls the function **driveDistance**(), with the first argument containing the variable `distance`, to drive the robot forward for 6 inches, then drive the robot backward for 5 inches. Assume the radius of wheels is 1.75 inches.

5.2 Turn Left and Turn Right

The **CMindstorms** class contains several other member functions with useful preprogrammed motions. The member function **turnLeft**() turns a two-wheel toward left with the syntax as follows.

```
    robot.turnLeft(angle, radius, trackwidth);
```

The amount turned by the robot is specified in the argument `angle` in degrees. The second argument is the radius of the two wheels. The third argument is the ***track width***, the distance between the two wheels as shown in Figure 5.1. In order to turn the robot with the correct angle, the radius of the two-wheels and the track width need to be specified. The units for both radius and track width must be the same. They can be inches, feet, centimeters, meters, etc.

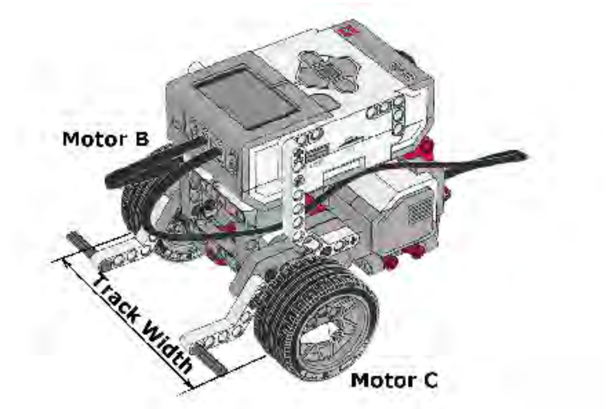


Figure 5.1: The track width for a two-wheel robot.

Similar to the member function **turnLeft**(), the member function **turnRight**() turns a robot toward right with the syntax as follows.

```
    robot.turnRight(angle, radius, trackwidth);
```

The amount turned by the robot is specified in the argument `angle` in degrees. The second argument is the radius of the two wheels. The third argument is the track width.

```
/* File: turn.ch
   Turn left and turn right */
#include <mindstorms.h>
CMindstorms robot;
double radius = 1.1;        // radius of 1.1 inches
double trackwidth = 4.54;   // the track width, the distance between two wheels

robot.driveDistance(5, radius);
robot.turnRight(90, radius, trackwidth);
robot.driveAngle(360);
robot.turnLeft(90, radius, trackwidth);
robot.driveAngle(360);
```

Program 5.3: Turning left and turning right.

For example, Program 5.3 drives a Mindstorms forward for 5 inches, turns right for 90 degrees, rolls forward for 360 degrees, turns left for 90 degrees, then rolls forward for another 360 degrees again.

E Do Exercises 1 and 2 on page 57.

5.2.1 Summary

1. Call the **CMindstorms** member function

   ```
   robot.turnLeft(angle, radius, trackwidth);
   ```

   to turn a Mindstorms toward left with the specified angle, radius for two wheels, and track width.

2. Call the **CMindstorms** member function

   ```
   robot.turnRight(angle, radius, trackwidth);
   ```

   to turn a Mindstorms toward right with the specified angle, radius for two wheels, and track width.

5.2.2 Terminology

robot.turnLeft(), **robot.turnRight()**, turn left, turn right.

5.2.3 Exercises

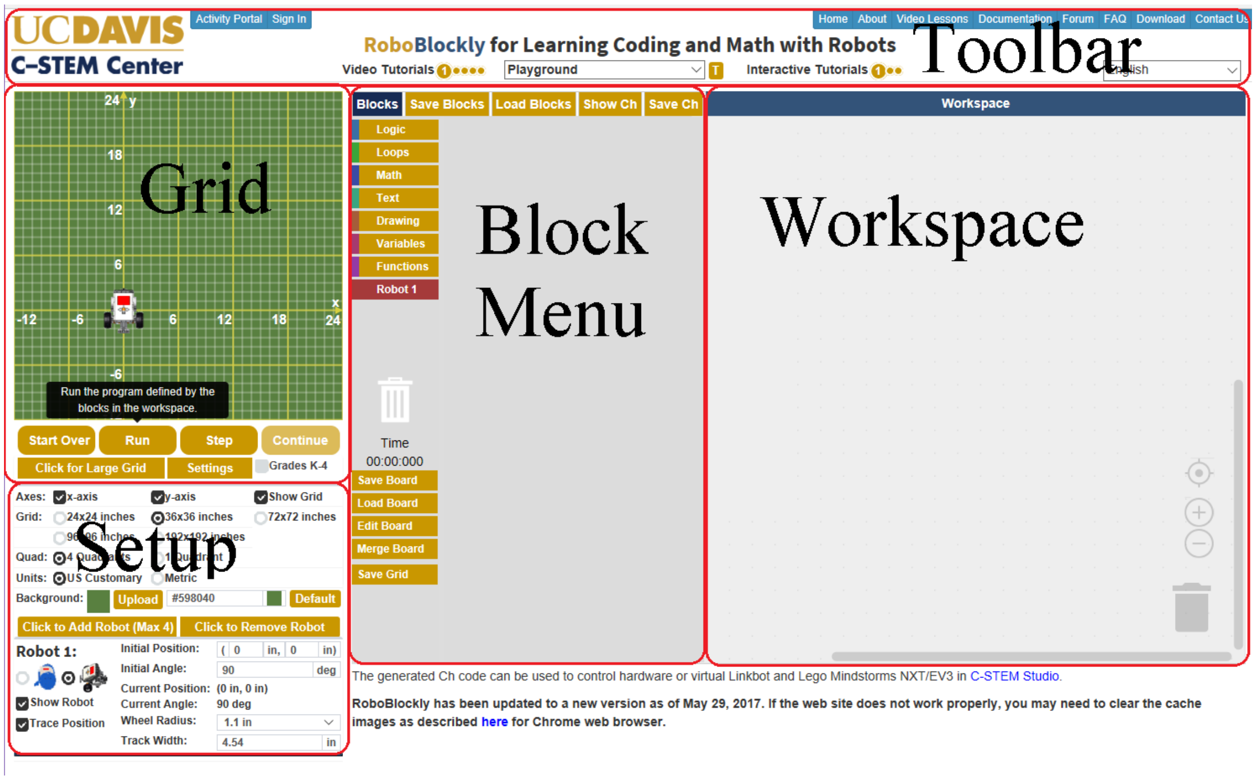
1. Write a program `turn2.ch` to drive a Mindstorms forward for 6 inches, turn left for 180 degrees, drive forward for 360 degrees, turn right for 180 degrees, and roll forward for 360 degrees.

2. Run the program `turn2.ch` developed in Exercise 1 in ChIDE in debug mode with the command `Next` on the debug bar.

5.3 Robot Simulation and Generating Robot Programs with RoboBlockly

Like RoboSim, RoboBlockly can be used to simulate the motion of Linkbot and Lego Mindstorms NXT or EV3. RoboBlockly can also generate Ch programs easily. The generated program can be downloaded to run in ChIDE to control either hardware robots or virtual robots in RoboSim. RoboBlockly can also be used to control hardware Linkbots and NXT/EV3 directly.

Figure 5.2 shows the graphical user interface (GUI) for the RoboBlockly. It consists of five parts: Toolbar, Grid, Setup, Block Menu, and Workspace.



Axes: [✓ x-axis] [✓ y-axis] [✓ Show Grid]
Grid: 24x24 inches [✓ 36x36 inches] 72x72 inches 96x96 inches 192x192 inches
Quad: [✓ 4 Quadrants] 1 Quadrant
Units: [✓ US Customary] Metric
Background: #598040
Robot 1: [✓ Show Robot] [✓ Trace Position]
Initial Position: (0 in, 0 in)
Initial Angle: 90 deg
Current Position: (0 in, 0 in)
Current Angle: 90 deg
Wheel Radius: 1.1 in
Track Width: 4.54 in

The generated Ch code can be used to control hardware or virtual Linkbot and Lego Mindstorms NXT/EV3 in C-STEM Studio.

RoboBlockly has been updated to a new version as of May 29, 2017. If the web site does not work properly, you may need to clear the cache images as described here for Chrome web browser.

Figure 5.2: The graphical user interface (GUI) in RoboBlockly.

The Toolbar contains the links to tutorials, documentations, robotics activities, coding activities, and math activities, etc.

The Grid will display the user's robot of choice (i.e. Linkbot or Mindstorms, or both) and its real-time simulated movement based on the blocks used. The grid and space on the Grid can be changed by in the Setup. Clicking the "Start Over" button will delete all blocks currently placed in the Workspace (which will be described in more detail next), while clicking "Run" will allow the user to run the code that is generated from the blocks that are placed in the Workspace. After "Run" has been clicked, it will change into the "Reset" button, which will allow the user to make changes to the blocks in the workspace and re-run the simulation with the new code.

On the Block Menu on the left, there are tabs that lead to various blocks. including Logic, Loops, Math, Text, Variables, Functions, Drawing, and Robots. These blocks can be dragged into the Workspace area for use. For example, Figure 5.3 has a driveDistance block in the Workspace. Extra or unnecessary blocks can be dragged to the trash can at the bottom right, or back to the left side for deletion. The tab "Show Ch" in the Block Menu can be clicked to show the generated Ch code for the blocks used in the workspace. The tab "Save Ch" in the Block Menu can be clicked to save the generated Ch code in the local machine.

Follow the instruction below to control hardware Linkbot and NXT/EV3 from RoboBlockly.

1. Launch Linkbot Labs from C-STEM Studio as usual

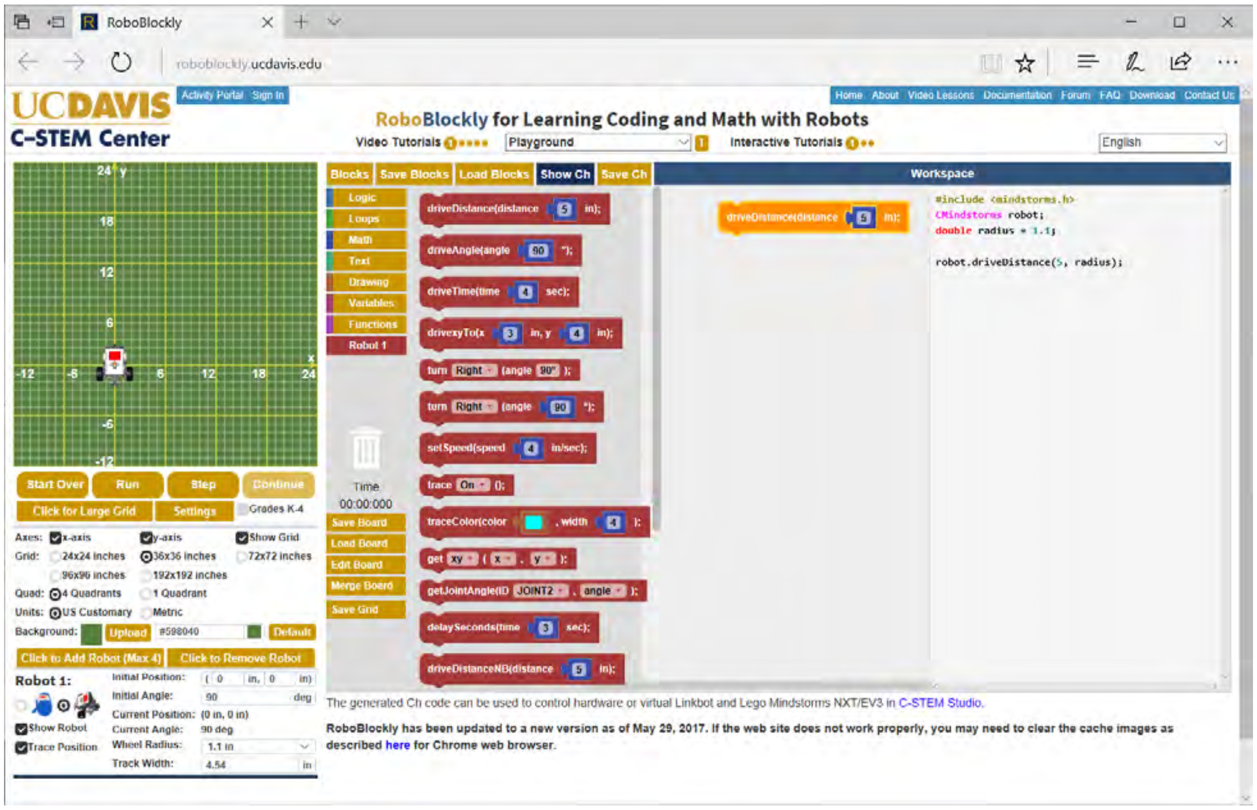
2. Add Linkbot ID inside Linkbot Labs as usual.

3. Launch RooboBlockly by clicking RoboBlcokly menu inside C-STEM Studio v5.0 or higher (Note, not from a regular browser).

4. Drag a Linkbot block such as **driveDistance**() to Workspace

5. Click “Run” to move both virtual on the grid and hardware Linkbot at the same time.

6. Click “Save Ch”, a Ch program, such as `roboblockly.ch`, will be saved in `C-STEM Studio->Student Home` folder and launched in ChIDE.



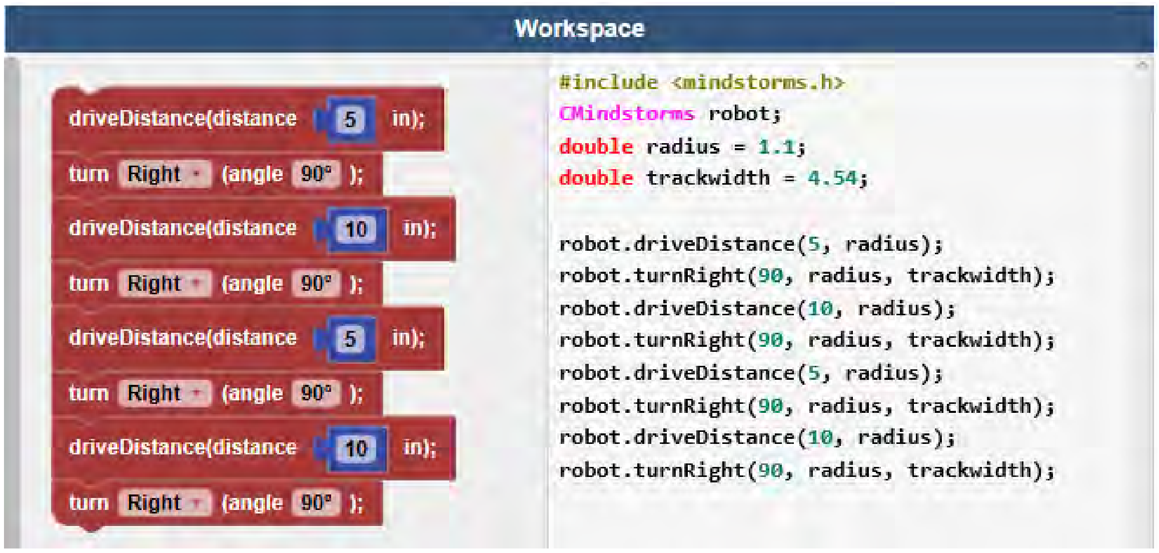
```
#include <mindstorms.h>
CMindstorms robot;
double radius = 1.1;

robot.driveDistance(5, radius);
```

The generated Ch code can be used to control hardware or virtual Linkbot and Lego Mindstorms NXT/EV3 in C-STEM Studio.

RoboBlockly has been updated to a new version as of May 29, 2017. If the web site does not work properly, you may need to clear the cache images as described here for Chrome web browser.

Figure 5.3: Drive a robot using **driveDistance**() in RoboBlockly.

As an example, the blocks and Ch code in Figure 5.4 will trace a 5x10 rectangle shown in Figure 5.5,


Workspace
driveDistance(distance 5 in);
turn Right (angle 90°);
driveDistance(distance 10 in);
turn Right (angle 90°);
driveDistance(distance 5 in);
turn Right (angle 90°);
driveDistance(distance 10 in);
turn Right (angle 90°);

```
#include <mindstorms.h>
CMindstorms robot;
double radius = 1.1;
double trackwidth = 4.54;

robot.driveDistance(5, radius);
robot.turnRight(90, radius, trackwidth);
robot.driveDistance(10, radius);
robot.turnRight(90, radius, trackwidth);
robot.driveDistance(5, radius);
robot.turnRight(90, radius, trackwidth);
robot.driveDistance(10, radius);
robot.turnRight(90, radius, trackwidth);
```

Figure 5.4: The blocks and Ch code for tracing a 5x10 rectangle using a Mindstorms in RoboBlockly.

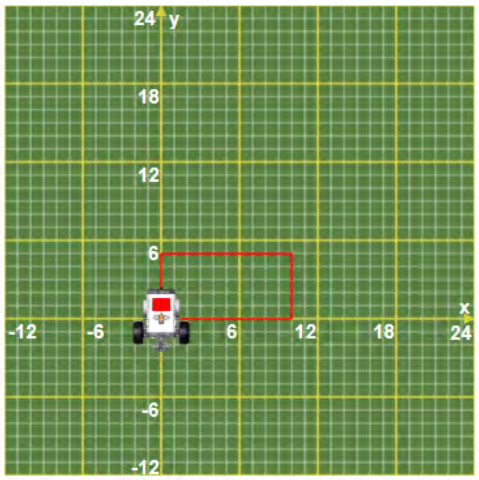


Figure 5.5: The 5x10 rectangle generated by the blocks shown in Figure 5.5.

E Do Exercises 1 and 3 on page 61.

5.3.1 Summary

1. Use blocks for **driveAngle**(), **driveDistance**(), **turnLeft**(), and **turnRight**() to program a robot in RoboBlockly.
2. Use RoboBlockly to generate Ch code for control a hardware robot or virtual robot in RoboSim using ChIDE.

5.3.2 Terminology

RoboBlockly, blocks, block menu, grid, setup, toolbar, workspace.

5.3.3 Exercises

1. Watch the following video tutorials for RoboBlockly in http://roboblockly.ucdavis.edu/videos/.

   (a) Interactive Tutorial.

   (b) T1. Introduction to RoboBlockly.

   (c) T2. RoboBlockly Toolbar.

   (d) T3. Grid and Setp.

   (e) T4. Workspace and Block Menu

   (f) R1. Drive a Distance: **driveDistance**()

   (g) R4. Drive a Distance by the Rotated Angle: **driveAngle**()

   (h) R5. Turn Left and Turn Right: **turnLeft**() and **turnRight**()

2. Use RoboBlockly to generate a Ch program `square.ch` to control a Linkbot-I.

   (a) Contruct a Linkbot program using blocks in RoboBlockly to trace a square of 8x8.

   (b) Save the generated Ch code in RoboBlockly and run the code in ChIDE to control a virtual Linkbot in RoboSim.

   (c) Save the generated Ch code in RoboBlockly and run the code in ChIDE to control a hardware Linkbot if you have the hardware.

3. Use RoboBlockly to generate a Ch program `square2.ch` to control a Mindstorms NXT or EV3.

   (a) Contruct a Lego Mindstorms NXT/EV3 program using blocks in RoboBlockly to trace a square of 8x8.

   (b) Save the generated Ch code in RoboBlockly and run the code in ChIDE to control a virtual NXT/EV3 in RoboSim.

   (c) Save the generated Ch code in RoboBlockly and run the code in ChIDE to control a hardware NXT/EV3 if you have the hardware.

CHAPTER 6

Interacting with a Mindstorms at Runtime through Variables and Input/Output Functions

6.1 The Output Function printf()

The ***input/output***, or ***I/O***, refers to the communication between a computer program and input/output devices. The inputs are the data received by the program. The outputs are the data produced by the program. In the previous section, we used the output function **printf**() to display the output from the program `hello.ch`, as shown in Figure 3.4, and the program `distance3.ch` in Program 6.1. In this section, we will learn more about the output function **printf**(). We will learn the input function **scanf**() in the next section.

The function **printf**() can be used to print text and data to the input/output pane. A general form of the function **printf**() and a sample application are shown in Figure 6.1. The `format` in the first argument is a string. This format string can contain an object called a conversion specifier, such as `"%lf"` and `"%d"`. A ***conversion specifier*** tells the program to replace that object with the value of the expression of specific type

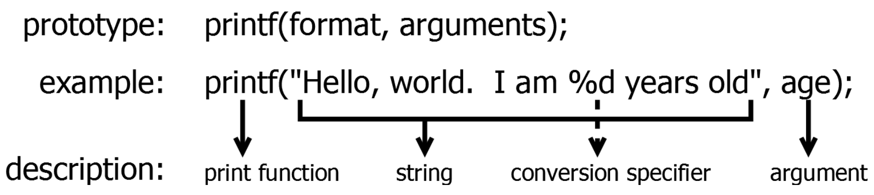


Figure 6.1: A format for using the function **printf**() and an example.

that follows the string. The expression can be a constant or variable. For instance, in Program 6.1 the line

```
    printf("The distance traveled by the robot is %lf inches.\n", distance);
```

contains the conversion specifier `"%lf"` which is replaced with the value of variable `distance` when the program is run. The printed output thus becomes:

```
    The distance traveled by the robot is 13.750000 inches.
```

Table 6.1 lists the conversion specifiers we will use along with the data type they represent. The conversion specifier `"%lf"` (letter 'l', not number '1') is used to print a decimal number or the value of a variable of **double** type. The conversion specifier `"%d"` is used to print an integer number or the value of a variable of **int** type.

Table 6.1: Conversion specifiers for the functions **printf**() and **scanf**().

Data Type	Format
double	`"%lf"`
int	`"%d"`

The conversion specifier `"%lf"` prints out a decimal with six digits after the decimal point. Using the wrong conversion specifier will give an incorrect result. For example, using `"%d"` for the decimal number 12.345 or variables of **double** type, and `"%lf"` for the integer 10 or variables of **int** type will give incorrect results. For example, the code below should use the conversion specifier `"%lf"` instead of `"%d"`.

```
    double speed = 10.5;
    printf("%d", 10.5);          // incorrect.
    printf("%d", speed);         // incorrect.
```

Application: Calculating the Distance of a Robot Traveled

Problem Statement:
A robot travels at the constant speed of 2.5 inches per second. The distance traveled by this robot can be expressed as follows:

$$distance = 2.5t$$

where $distance$ is measured in inches from the starting point and t is time in seconds. Therefore, if you want to know where the robot is at any time, you will take the number of seconds and multiply that by 2.5. Let's write a program to calculate the distance when the traveling time t for the robot is 5.5 seconds.

We will examine the source code for this program step-by-step. We will need some variables in the program to represent variables in the equation $distance = 2.5t$. Since the values for distance and time can be decimals we will declare them both as **double** types.

```
    double t;            // the traveling time in seconds
    double distance;     // distance traveled
```

We also know that we are looking at the problem when 5.5 seconds have passed, or $t = 5.5$, so we can initialize the variable t in the program.

```
    t = 5.5;            // 5.5 seconds for traveling time
```

To make our program actually do something, we need to tell it how to calculate the distance

```
    distance = 2.5 * t; // calculate the distance traveled
```

Note that the operators '+' and '-' in a program can be used for addition and subtraction operations in the same manner as in math. However, the program does not recognize proximity as multiplication. The multiplication operator '*' is needed for a multiplication operation. The division operator '/' is used for a division operation.

Finally, we want to show the answer to the user by calling the function **printf**() similar to how we used it in `hello.ch` in Program 3.1. There are some additional features needed for this **printf**() function that will be explained in section 6.1. For now, accept that `"%lf"` is replaced by the value of the variable `distance`.

```
    printf("The distance traveled by the robot is %lf inches.\n", distance);
```

Thus, the final source code will look like Program 6.1. When Program 6.1 is executed, the following output will be displayed in the input/output pane.

```
    The distance traveled by the robot is 13.750000 inches.
```

The computation result from Program 6.1 will be verified experimentally using a robot program presented in Program 8.14 on page 119.

```
/* File: distance3.ch
   Calculate the distance of a robot traveled at 2.5 inches pers second. */
double t;           // the traveling time in seconds
double distance;    // distance traveled

t = 5.5;            // 5.5 seconds for traveling time

distance = 2.5 * t; // calculate the distance traveled
printf("The distance traveled by the robot is %lf inches.\n", distance);
```

Program 6.1: Calculating and printing the distance of a robot traveled using **printf**().

In Program 6.1, for simplicity and consistency with the mathematical notations, the variables `t` and `distance` represent the time and distance, respectively. Comments for these variables are added in the declaration of these variables to make their intended use more clear. Using variables to represent mathematical notations make a program easier to modify and more readable. It is especially helpful for solving problems with complicated logic.

E Do Exercise 2 on page 66.

6.1.1 Precision of Decimal Numbers

By default, the conversion specifier `"%lf"` prints out a decimal number with six digits after the decimal point. When a decimal number is used to represent currency, we want to have two digits after the decimal point for cents. We can accomplish this by specifying the precision of the output. The ***precision*** of a decimal number specifies the number of digits after the decimal point. The precision typically takes the form of a period (.) followed by an integer. For example, the conversion specifier `"%.2lf"` specifies the precision with two digits after the decimal point. The number after the specified amount is rounded to the nearest value. For example, with the conversion specifier `"%.2lf"`, the decimal number 12.1234 is printed as 12.12 with the precision value of 2, whereas 12.5678 is printed as 12.57.

Application: Calculating the Cost for Buying the Ice Cream

Problem Statement:
The sale price of ice cream is $0.47 per ounce. Write a program `icecream.ch` to calculate the cost of buying 5.5 ounces of ice cream.

Program 6.2 can calculate the cost for buying ice cream. The math formula to calculate the cost of buying the ice cream at $0.47 per ounce is

$$\text{cost} = 0.47 * \text{weight}$$

```
/* File: icecream.ch
   Calculate the cost for 5.5 ounces of the ice cream.
   The ice cream is sold by weight. $0.47 per ounce. */

/* declare variables weight and cost */
double weight, cost;

/* initialize weight in lb */
weight = 5.5;

/* calculate the cost*/
cost = 0.47 * weight;

/* display the cost as output */
printf("The ice cream costs $%.2lf \n", cost);
```

Program 6.2: Calculating the cost for purchasing ice cream.

Program 6.2 declares two variables `weight` and `cost`, assigns the weight, and calculates the cost by the following statements.

```
    double weight, cost;  // weight and cost of the yogurt are declared
    weight = 5.5;         // weight is assigned the value 5.5
    cost = 0.47 * weight; // cost is calculated
```

printf

```
    printf("The ice cream costs $%.2lf \n", cost);
```

```
    The ice cream costs $2.58
```

E Do Exercises 1 and 3 on page 66.

6.1.2 Summary

1. Use the output function **printf**() with the conversion specifier `"%lf"` for decimal numbers with six digits after the decimal point and `"%d"` for integers. For example,

```
    printf("distance = %lf\n", f);
    printf("n = %d\n", n);
```

2. Use the precision of the output function **printf**() for decimal numbers. The conversion specifier `"%.#lf"` is used for precision of decimal numbers where `"#"` is replaced by a whole number for printing a decimal number with `"#"` number of digits after the decimal point.
3. Write programs with the output function **printf**() to solve applied problems.

6.1.3 Terminology

character string, conversion specifier, conversion specifier `"%lf"`, conversion specifier `"%d"`, copy program, format string, I/O, input, output, precision, **printf**()

6.1.4 Exercises

1. What's wrong with the following code and how do you correct it?

```
    int n = 10;
    printf("n is %lf\n", n);
```

2. A joint of a robot rotates at the constant speed of 90 degrees per second. The joint angle can be expressed as follows:

$$angle = 90t$$

   where $angle$ is measured in degrees from the starting point and t is time in seconds. Write a program `jointangle.ch` to calculate the joint angle of the robot 4.5 seconds after the robot starts its motion.

3. The sale price of frozen yogurt is \$0.39 per ounce. Write a program `yogurt.ch` to calculate the cost of buying 4.5 ounces of frozen yogurt.

6.2 Input into Programs Using Function scanf()

In the previous section, we learned how to produce the output from a program using the function **printf**(). In this section, we will learn how to write a program to accept the input values from the user. Therefore, the same program can be used to solve the same problems with different data. The function **scanf**() is used to input data to a program from the standard input, which is usually the keyboard. The input function **scanf**() can set a variable with the value from the user input. The conversion specifiers listed in Table 6.1 can also be used for the function **scanf**() in the form of

```
    scanf("%lf", &x);
```

for a variable `x` of **double** type and

```
    scanf("%d", &n);
```

for a variable `n` of **int** type, The address operator '`&`', preceding a variable name, obtains the address of the variable so that the value from the user input can be stored in the variable. For the function scanf(), you must use & before the variable name or you will get an error.

Application: An Ice Cream Shop

Problem Statement:
Write a program `icecream2.ch` for an Ice Cream Shop to process the sale of ice cream. The sale price for ice cream is $0.47 per ounce.

```
/* File: icecream2.ch
   Calculate the cost for the ice cream */

/* declare variables weight and cost */
double weight, cost;

/* get the user input for the value of the variable weight */
printf("Welcome to the Amazing Ice Cream Shop\n");
printf("We sell ice cream by weight, $0.47 per ounce.\n");
printf("Enter the weight in ounces.\n");
scanf("%lf", &weight);

/* calculate the cost */
cost = 0.47 * weight;

/* display the cost as output */
printf("The ice cream costs $%.2lf \n", cost);
printf("Thank you.\n");
```

Program 6.3: Using the function **scanf**() to input the weight of ice cream.

Program 6.3 can be used to process the purchase of ice cream. Like the program `icecream.ch` in Program 6.2, it declares two variables `weight` and `cost`. When the program is executed, the user enters the weight of the ice cream in ounces to be purchased to the variable `weight` through the function **scanf**(). After the program obtains the weight input from the user, the cost is calculated by multiplying the weight with 0.47. The conversion specifier `"%.2lf"` is used to print the cost to the nearest cent with two digits after the decimal point. An interactive execution of Program 6.3 through the input/output pane is displayed below:

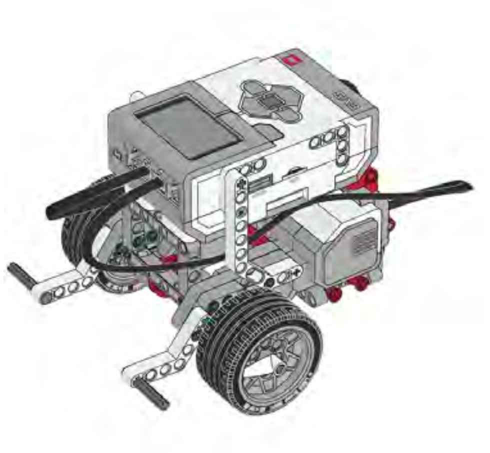
```
Welcome to the Amazing Ice Cream Shop
We sell ice cream by weight, $0.47 per ounce.
Enter the weight in ounces.
5
The ice cream costs $2.35
Thank you.
```

In this execution, the value `5` for the weight in ounces is entered after the prompt.

If a program is used interactively, it is important that a message be displayed before the function scanf() is called so that the user of the program is asked to input data accordingly, as shown by the statement `"Enter ..."`

E Do Exercise 1 on page 70.

Application: Controlling a Mindstorms with the User Input for Distance and Joint Angle



Problem Statement:
Write a program `distanceangle_p.ch` to control a Mindstorms with the distance travelled and angle to turn right by the user input.

```
/* File: distanceangle_p.ch
   User inputs the values for distance to drive and angle to turn right */
#include <mindstorms.h>
CMindstorms robot;
double distance; // declare variable 'distance' to drive
double angle;    // declare variable 'angle' for turnRight
double radius = 1.1;     // radius of the wheels
double trackwidth = 4.54; // track width inches

/* User inputs distance and angle */
printf("Enter the distance to drive\n");
scanf("%lf", &distance);
printf("Enter the angle to turn right \n");
scanf("%lf", &angle);

/* turn right by angle */
robot.driveDistance(distance, radius);
robot.turnRight(angle, radius, trackwidth);
```

Program 6.4: Using the function scanf() to input joint angles.

We can develop the program `distanceangle_p.ch` in Program 6.4. The `distance` for the function call

```
    robot.driveDistance();
```

and the `angle` for the function call

```
    robot.turnRight(angle, radius, trackwidth);
```

are obtained from the user at the runtime through the input function scanf(). An interactive execution of Program 6.4 is shown below.

```
   Enter the distance to drive
   8
   Enter the angle to turn right
   180
```

In this case, the robot drive forward to 8 inches first, then turn right 180 degrees.

E Do Exercises 2 and 3 on page 70.

6.2.1 Summary

1. Use the input function scanf() with the conversion specifier `"%lf"` for variables of **double** type and `"%d"` for variables of **int** type. The symbol '`&`' must precede a variable name. For example,

   ```
      scanf("%lf", &weight);
   ```

2. Write programs with the input/output functions to solve applied problems.

6.2.2 Terminology

address operator &, buffer, copy program, scanf()

6.2.3 Exercises

1. Write a program `yogurt2.ch` for a Yogurt Shop to process the sale of frozen yogurt. The sale price for frozen yogurt is $0.39 per ounce.

2. Modify Program 5.1 as a new program `driveangle4_p.ch` to control a Mindstorms. The program shall use a variable `angle` to get the joint angle from the user at runtime through the function **scanf**(). The program calls the function **driveAngle**() with the variable `angle` to drive the Mindstorms forward and backward. Test the program with the joint angle 360 degrees.

3. Write a program `drivedistance3_p.ch` to control a Mindstorms. The program shall use variables `distance` and `radius` to get the distance and radius of two-wheels from the user at runtime through the function **scanf**(). The program calls the function **driveDistance**() with the variables `distance` and `radius` to drive the Mindstorms forward. Test the program with the distance 6 inches and radius of 1.1 inches. You can also test the program with wheels of different radius.

6.3 Number Line for Distances

The member function **driveDistance**() can be called multiple times to drive a two-wheel robot forward or backward. In Program 6.5, the statements

```
    double distance1 = 12;      // distance1 in inches
    double distance2 = -5;      // distance2 in inches
    double distance3 = 3;       // distance3 in inches
```

declare variables `distance1`, `distance2`, and `distance3` and initialize them with the values of 12, -5, and 3 inches, respectively. The statements

```
    robot.driveDistance(distance1, radius);
    robot.driveDistance(distance2, radius);
    robot.driveDistance(distance3, radius);
```

move the robot from the initial zero position forward for 12 inches, then backward 5 inches, and forward 3 inches again. The robot will stop at 9 inches from the original position. Program 6.5 generates a plot of number line representing distances for these movements, as shown in Figure 6.2.

```
/* File: robotnumline.ch
   Plot robot distances in number line */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double radius = 1.1;      // radius of the wheel in inches
double distance1 = 12;     // distance1 in inches
double distance2 = -5;     // distance2 in inches
double distance3 = 3;      // distance3 in inches
CPlot plot;                // plotting class

robot.driveDistance(distance1, radius);
robot.driveDistance(distance2, radius);
robot.driveDistance(distance3, radius);

/* assume robot starts at 0 */
plot.numberLine(0, distance1, distance2, distance3);
plot.label(PLOT_AXIS_X, "Distance (inches)");
plot.plotting();
```

Program 6.5: Driving a robot to different positions and representing the distances on a number line.

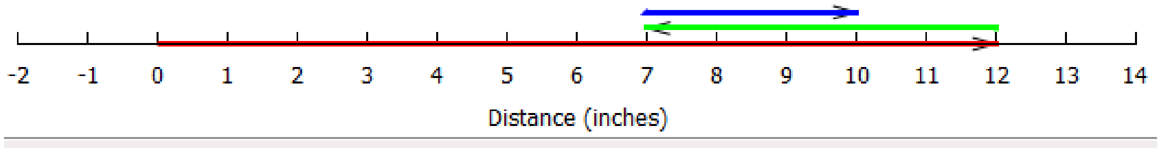


Figure 6.2: A number line for distances of a two-wheel robot, generated by Program 6.5.

The following statements in Program 6.5

```
    #include <chplot.h>
    CPlot plot;                // plotting class

    plot.numberLine(0, distance1, distance2, distance3);
    plot.label(PLOT_AXIS_X, "Distance (inches)");
    plot.plotting();
```

are responsible for generating the plot shown in Figure 6.2.

In Program 6.5, the line

```
    #include <chplot.h>
```

includes the header file **chplot.h**. The purpose of including the header file **chplot.h** is to use the class **CPlot** defined in this header file. As we have learned in previous chapters, a class is a user defined data type in Ch. The symbol **CPlot** can be used in the same manner as the symbol **int** or **double** to declare variables.

The line

```
    CPlot plot; //plotting class
```

declares the variable `plot` of type **CPlot** for plotting. A function associated with a class is called a ***member function***. For example, the function **plot.numberLine()** or **numberLine()** is a member function of the class **CPlot**. In Program 6.5, member functions of the class **CPlot** are called to process the data for the object `plot`.

The member function **plot.numberLine**() can have a variable number of arguments. The first argument is the initial position of the robot. Each subsequent argument represents a distance relative to its current position. The function call

```
    plot.numberLine(0, distance1, distance2, distance3);
```

is equivalent to

```
    plot.numberLine(0, 12, -5, 3);
```

It draws a direction line with an arrow from 0 to 12, then from 12 inches backward 5 inches, and forward 3 inches. To make it more clear, each direction line with a different color has a vertical offset from the number line. The robot ends at the position of 10 inches, which is the sum of the arguments of the member function **numberLine()**.

The subsequent function call

```
    plot.label(PLOT_AXIS_X, "Distance (inches)");
```

add a label to the number line. The macro **PLOT_AXIS_X** for the x axis is defined in the header file **chplot.h**.

Finally, after the plotting data are added, the program needs to call the function

```
    plot.plotting();
```

to generate a plot.

E Do Exercise 1 on page 74.

In some applications, the robot may not start at the zero position. In this case, the initial position of a robot can be treated as an offset in the first argument of the member function **plot.numberLine**() for distance on a number line.

```
/* File: robotnumlineoffset.ch
   Plot robot distances in number line */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double radius = 1.1;       // radius of the wheel in inches
double offset = 2;          // the offset for the initial distance
double distance1 = 12;      // distance1 in inches
double distance2 = -5;      // distance2 in inches
double distance3 = 3;       // distance3 in inches
CPlot plot;                 // plotting class

robot.driveDistance(distance1, radius);
robot.driveDistance(distance2, radius);
robot.driveDistance(distance3, radius);

/* assume robot starts at offset */
plot.numberLine(offset, distance1, distance2, distance3);
plot.label(PLOT_AXIS_X, "Distance (inches)");
plot.plotting();
```

Program 6.6: Driving a robot to different positions with an initial offset from the origin.

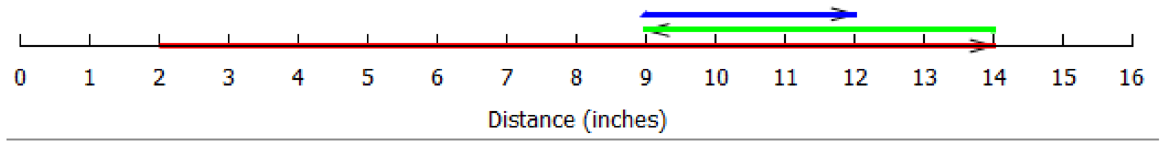


Figure 6.3: A number line for distances of a two-wheel robot with an initial offset from the origin, generated by Program 6.6.

Program 6.6 will generate the plot shown in Figure 6.3. In Program 6.6, a robot moves starting from the initial position of 2 inches from the origin. The initial position is represented by a variable `offset` as follows.

```
    double offset = 2;          // the offset for the initial distance
```

The function call

```
    plot.numberLine(offset, distance1, distance2, distance3);
```

equivalent to

```
    plot.numberLine(2, 12, -5, 3);
```

draws a direction line with an arrow from 2 to 14 for 12 inches, then backward 5 inches, and forward 3 inches. The robot ends at the position of 12 inches on a number line for distance, which is the sum of the arguments of the member function **numberLine()**.

E Do Exercise 2 on page 74.

6.3.1 Summary

1. Include the header file **chplot.h** and use the class **CPlot** to declare a variable `plot` by the following two statements

   ```
       #include<chplot.h>
       CPlot plot;
   ```

2. Call the **CPlot** member function

   ```
       plot.numberLine(offset, x1, ...);
   ```

   such as

   ```
       plot.numberLine(4.5, 3);
       plot.numberLine(4.5, 3, 8.5, -10, 20, -5);
   ```

3. Call the **CPlot** member function

   ```
       plot.label(PLOT_AXIS_X, "xlabel");
   ```

   to label a number line.

4. Call the **CPlot** member function

   ```
       plot.plotting();
   ```

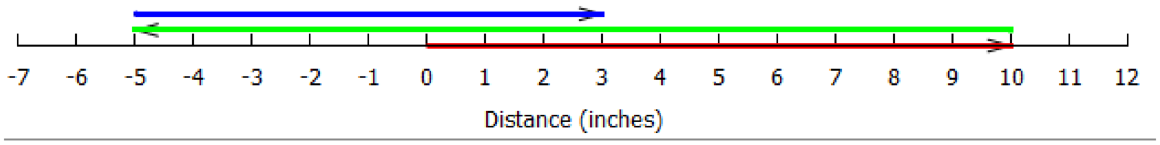
   to generate the final graph.

6.3.2 Terminology

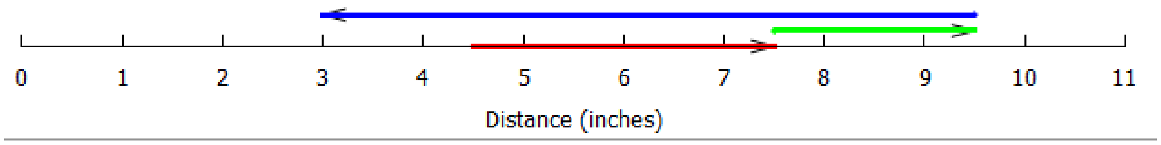
number line, **#include <chplot.h>**, **CPlot**, **plot.label()**, **plot.numberLine()**, **plot.plotting()**.

6.3.3 Exercises

1. Write programs to generate a number line shown below. (a) Write a program `numbeline2.ch` to just generate a number line without moving a robot. (b) Write a program `robotnumline2.ch` to control a Mindstorms configured as a two-wheel drive robot and generate a number line for the distance of the robot. Assume the radius of wheels is 1.1 inches.



2. Write programs to generate a number line shown below with an offset of 4.5 inches from the origin for the initial position of the robot. (a) Write a program `numbelineoffset.ch` to just generate a number line without moving a robot. (b) Write a program `robotnumlineoffset2.ch` to control a Mindstorms configured as a two-wheel drive robot and generate a number line for the distance of the robot. Assume the radius of wheels is 1.1 inches.



CHAPTER 7

Writing Programs to Control a Group of Mindstorms to Perform Identical Tasks

7.1 Control a Group of Mindstorms with Identical Movements

By using groups, Mindstorms can be synchronized easily using only a few lines of code.

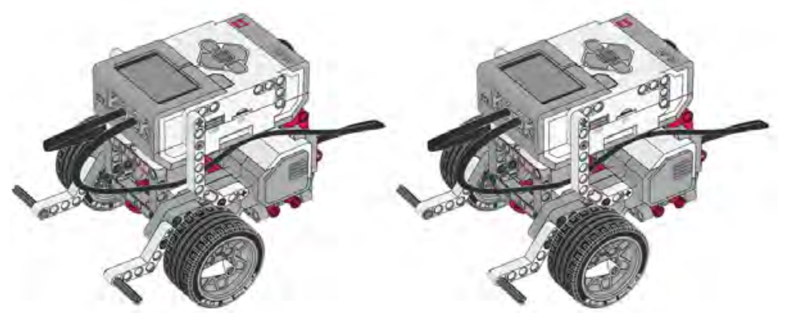


Figure 7.1: Control two Mindstorms to perform identical tasks simultaneously.

```
/* File: group.ch
   Control multiple robot modules simultaneously using the CMindstormsGroup class */
#include <mindstorms.h>
CMindstorms robot1, robot2;
CMindstormsGroup group;  // the robot group

/* add the two modules as members of the group */
group.addRobot(robot1);
group.addRobot(robot2);

group.driveAngle(360);  // drive robots forward
group.driveAngle(-360); // drive robots backward
```

Program 7.1: Controlling a group of Mindstorms with identical movements.

Program 3.2 on page 31 controls a single Mindstorms to roll forward by 360 degrees and then roll backward by 360 degrees. Program 7.1 peforms identical movements for two Mindstorms in the same manner as Program 3.2.

After declaring a separate variable and connecting it to a Mindstorms for each of the two Mindstorms, the line

```
    CMindstormsGroup group;
```

creates a variable group of class **CMindstormsGroup**. The class **CMindstormsGroup** is defined in the header file **mindstorms.h**, just as the class **CMindstorms** are. The general syntax of the **CMindstormsGroup** member function **addRobot**(), which is used to add a Mindstorms to a group, is as follows:

```
    group.addRobot(name);
```

The argument name represents the variable name of the Mindstorms that you want to add to the group. The next two lines

```
    group.addRobot(robot1);
    group.addRobot(robot2);
```

add each Mindstorms object to the group, one at a time. After these statements are executed, the variable group will be used to control both robot1 and robot2 at the same time. You can add any number of Mindstorms to a group.

The class **CMindstormsGroup** includes member functions that are similar to those included in the class **CMindstorms**. Examples of such member functions include **driveAngle**() and **driveDistance**(). In later sections we will see additional examples of such functions. The **CMindstormsGroup** versions of these member functions have one important difference: they move all the Mindstorms in a group identically, instead of only a single Mindstorms. It is more convenient to be able to write one line of code to move all the Mindstorms in a single group than it is to write a separate line of code for each Mindstorms.

The lines

```
    group.driveAngle(360);
    group.driveAngle(-360);
```

cause both robot1 and robot2 to roll forward at the same time by 360 degrees and then roll backward at the same time by 360 degrees, just as **robot.driveAngle(angle)** and **robot.driveAngle(−angle)** did for a single Mindstorms in Program 3.2.

E Do Exercises 1, 2, and 3 on page 77.

7.1.1 Summary

1. Include the header file **mindstorms.h** and use the class **CMindstormsGroup** to declare a variable group by the following two statements

   ```
   #include <mindstorms.h>
   CMindstormsGroup group;
   ```

   for controlling a group of Mindstorms.

2. Call the **CMindstormsGroup** member function

   ```
   group.addRobot(name);
   ```

   to add a single Mindstorms to a group.

3. Call the **CMindstormsGroup** member function

   ```
   group.driveAngle(angle);
   ```

   to drive all Mindstorms in a group forward or backward by the same specified angle, relative to their current positions.

7.1.2 Terminology

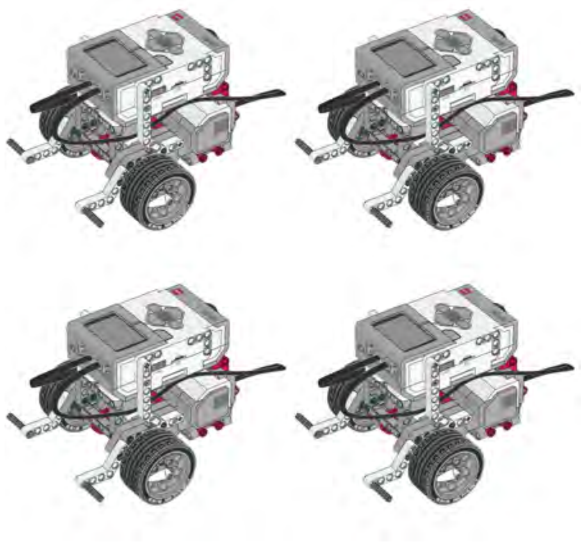
CMindstormsGroup, **group.addRobot()**, **group.driveAngle()**, groups, identical movements.

7.1.3 Exercises

1. Run the program `group.ch` in Program 7.1 to perform identical movements for two Mindstorms.

2. Write a program `group2.ch` to perform identical movements, as presented in Program 7.1, for three Mindstorms.

3. Write a program `group3.ch` to perform identical movements, as presented in Program 7.1, for four Mindstorms.



7.2 Control an Array of Mindstorms with Identical Movements

Another way to add multiple robots to a group is to use an array of robots. This is demonstrated in Program 7.2 below.

```
/* File: grouparray.ch
 * A group with 4 Mindstorms in synchronized motion using an array of 4 elements.
 *
 *         1       2
 *       |--|     |--|
 *       |__|     |__|
 *
 *        3       4
 *       |--|     |--|
 *       |__|     |__|
 *
 * Make sure that robots are attached with wheels. */
#include <mindstorms.h>
CMindstorms robot[4];      /* declare an array of 4 Mindstorms */
CMindstormsGroup group;    /* declare a Mindstorms group  */
double radius = 1.1;       // radius of 1.1 inches
double trackwidth = 4.54; // the track width, the distance between two wheels

group.addRobots(robot);    /* add 4 Mindstorms to the group */

group.driveAngle(360);
group.driveAngle(-360);
group.turnLeft(90, radius, trackwidth);
group.driveAngle(360);
group.driveAngle(-360);
group.turnLeft(45, radius, trackwidth);
group.driveAngle(360);
group.driveAngle(-360);
group.turnRight(135, radius, trackwidth);
group.driveDistance(5, radius);
group.driveDistance(-5, radius);
```

Program 7.2: Controlling a group of four Mindstorms with identical movements using an array of 4 elements.

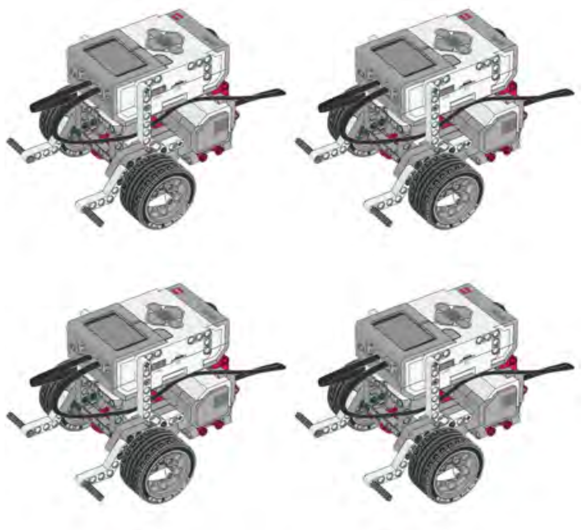


Figure 7.2: A group of four Mindstorms with identical movements.

Program 7.2 performs identical movements for four Mindstorms using an array with 4 elements. In this program, instead of declaring a separate variable for each Mindstorms, we declare one variable for multiple Mindstorms. The line

```
    CMindstorms robot[4];
```

declares an array of 4 Mindstorms. An ***array*** is a special kind of variable that stores a collection of individual values that are of the same data type. Each item in the collection can be accessed by using an index number. Arrays are useful because instead of having to separately store related information in different variables, you can store them as a collection in just one variable. The general syntax for declaring an array is as follows

```
    type name[num];
```

where `num` specifies the number of elements you want in the array. This number can be any positive integer value.

In particular, an array comes in handy in our program for use with the **CMindstormsGroup** member function **addRobots**(). The syntax for this new function is

```
    group.addRobots(name);
```

where name refers to an array of Mindstorms, instead of just one Mindstorms. With this member function, we can add all of our robots to a group at once, instead of one at a time like we did in Program 7.2. For instance, the line

```
    group.addRobots(robot);
```

adds the array of four Mindstorms to the group with just one line of code. This is easier than using four lines of code to add four Mindstorms to the group.

With Program 7.2 we discovered that the class **CMindstormsGroup** has its own version of the member function **driveAngle**(). Now we can add three more member functions to this list: **turnLeft**(), **turnRight**(). and **driveDistance**(). The line

```
    group.turnLeft(90, radius, trackwidth);
```

turns not just one Mindstorms but all four Mindstorms left by 90 degrees at the same time. Similarly, the line

```
    group.turnRight(135, radius, trackwidth);
```

turns all four Mindstorms 135 degrees right. The statements

```
    group.driveDistance(5,  radius);
    group.driveDistance(-5, radius);
```

drive all Mindstorms in the group forward by 5 inches first, then backward by 5 inches.
E Do Exercises 1 and 2 on page 81.

7.2.1 Summary

1. Declare an array variable to store a group of Mindstorms.

   ```
       CMindstorms robot[4];
   ```

   The number in square brackets is the number of Mindstorms in the group. This number can be any positive integer value.

2. Call the **CMindstormsGroup** member function

   ```
       group.addRobots(name);
   ```

to add an array of Mindstorms to a group. In this case `name` is the array name (identifier) instead of a single variable name.

3. Call the **CMindstormsGroup** member functions

   ```
       group.turnLeft(angle, radius, trackwidth);
       group.turnRight(angle, radius, trackwidth);
   ```

   to turn a group of Mindstorms left or right with the specified angle, radius for two wheels, and track width.

4. Call the **CMindstormsGroup** member function

   ```
       group.driveDistance(distance, radius);
   ```

   to drive a group of Mindstorms by the `distance` with the specified `radius` for two wheels.

7.2.2 Terminology

array, **group.addRobots()**, **group.turnLeft()**, **group.turnRight()**, **group.driveDistance()**.

7.2.3 Exercises

1. Run the program `grouparray.ch` in Program 7.2 to perform identical movements for 4 Mindstorms.

2. Write a program `grouparray2.ch` to control a group of 9 Mindstorms with the same motion as presented in the program `grouparray.ch` in Program 7.2.

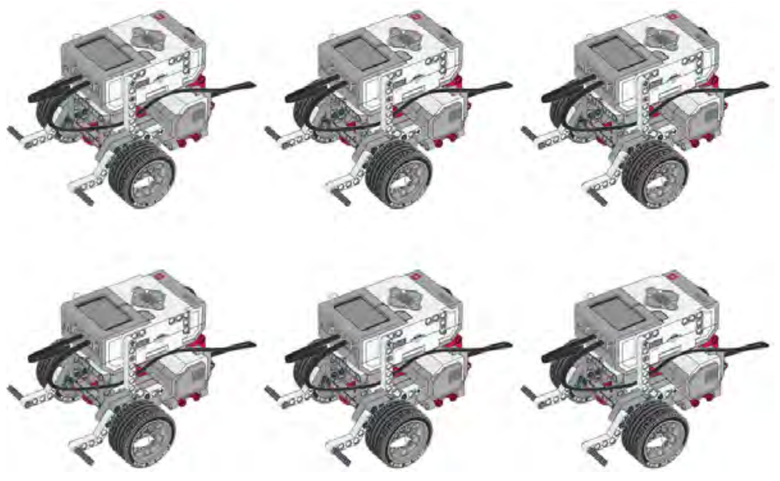


Figure 7.3: A group of six Mindstorms with identical movements.

CHAPTER 8

Controlling a Mindstorms as a Two-Wheel Robot

A Mindstorms can be configured as a two-wheel robot as shown in Figure 8.1. In this configuration, ports B and C of the brick are attached with two motors driving two wheels. In this Chapter, we will learn more programming features on how to control a Mindstorms as a two-wheel robot.

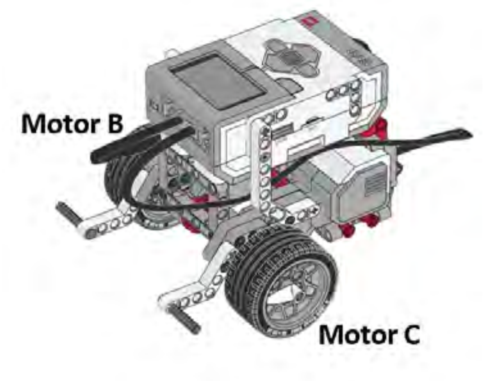


Figure 8.1: A two-wheel robot.

8.1 Move a Two-Wheel Robot with the Specified Distance

8.1.1 Move a Two-Wheel Robot with the Specified Speed, Joint Angles, and Distance

The **CMindstorms** class includes two additional member functions that can be used for the two-wheel Mindstorms configuration. The first of these two member functions is **setSpeed()**, which can be used to set both motors B and C of a Mindstorms to the desired speed with the specified wheel radius. The general syntax of this function is

```
    robot.setSpeed(speed, radius);
```

The argument `speed` is the desired vehicle speed in distance/second. The argument `radius` is the radius of the currently attached wheels, which should have the same units of distance as the argument `speed`. For instance, if `speed` is in inches/second, then `radius` should be in inches. If the speed is positive, the robot will drive forward. If the speed is zero, the robot will not move. If the speed is negative, the robot will drive backward.

The second member function is **driveDistance()**, which can be used to drive a Mindstorms a desired distance using a specific wheel radius. The general syntax of this function is

```
    robot.driveDistance(distance, radius);
```

where `distance` specifies how far you want the Mindstorms to move and `radius` specifies the radius of the currently attached wheels. The values of both `distance` and `radius` should have the same unit.

The member functions **setSpeed()** and **driveDistance()** can be used in combination to drive a Mindstorms with a specified speed and distance, as demonstrated solving the following problem.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `setspeed.ch` to drive forward 360 degrees using the member function **driveAngle**(), drive backward 360 degrees using the member function **driveAngle**() again, and then drive forward 5 inches using the member function **driveDistance**(). For these three motions, the robot drives at the speed of 2.5 inches per second.

```
/* File: setspeed.ch
   Set the speed of a two-wheel robot. */
#include <mindstorms.h>
CMindstorms robot;
double radius=1.1;  // the radius of the two wheels of the robot in inches
double speed=2.5;   // the speed in 2.5 inches per second for a two-wheel robot
double distance=5;  // the distance in 5 inches to drive forward

/* set the speed for a two-wheel robot to 3 inches per second */
robot.setSpeed(speed, radius);

/* drive the robot 360 degrees forward for joints 2 and 3 */
robot.driveAngle(360);

/* drive the robot 360 degrees backward for joints 2 and 3 */
robot.driveAngle(360);

/* drive forward for 'distance' inches */
robot.driveDistance(distance, radius);
```

Program 8.1: Setting the speed of a two-wheel robot using **setSpeed()**.

In Program 8.1 the variable `radius` is set to 1.1 inches, the variable `speed` is set to 2.5 inches per second, and the variable `distance` is set to 5 inches. The line

```
    robot.setSpeed(speed, radius);
```

sets the speeds for motors B and C of the Mindstorms to drive the robot at 2.5 inches per second with a wheel radius of 1.1 inches. Then when the next two lines

```
    robot.driveAngle(360);
    robot.driveAngle(-360);
```

are executed, the Mindstorms drives forward 360 degrees and then backward 360 degrees at a speed of 2.5 inches per second. The last line

```
    robot.driveDistance(distance, radius);
```

drives the Mindstorms forward 5 inches at the same speed of 2.5 inches per second.
E Do Exercises 1 and 2 on page 91.

8.1.2 Control a Mindstorms with the Speed and Distance Input from the User Using the Function scanf()

In the previous section we set the speed and distance of a Mindstorms by providing those values in the program specifically. In this section we will learn how to set the speed and distance of a Mindstorms with user input. This way the same program can be used to solve the same problem but with different data. Code reusability is an effective and convenient programming tool.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `drivedistance_p.ch` to accept the user input of speed and distance for moving the robot using the member function **driveDistance**().

```
/* File: drivedistance_p.ch
   Drive a two-wheel robot with the user specified radius of wheels, speed,
   and distance. */
#include <mindstorms.h>
CMindstorms robot;
double speed;        // the speed in inches per second for a two-wheel robot
double radius;       // the radius of the two wheels of the robot in inches
double distance;     // distance to drive

printf("Enter the radius of the two wheels in inches\n");
scanf("%lf", &radius);

printf("Enter the speed of the two-wheel robot in inches per second\n");
scanf("%lf", &speed);
/* set the speed for a two-wheel robot */
robot.setSpeed(speed, radius);

printf("Enter the distance in inches for the two-wheel robot to drive\n");
scanf("%lf", &distance);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);
```

Program 8.2: Using the input function **scanf**() to specify the speed and distance.

Program 8.2 uses the function **scanf**() to set the variables `speed`, `radius`, and `distance` to the values desired by the user. As we learned in Section 6.2, the input function **scanf**() is used to set the value of a variable from the user input at runtime. For instance, the line

```
    scanf("%lf", &radius);
```

is used to set the value of the variable `radius` from keyboard input at runtime. Recall that when using **scanf**() the address operator '&' must precede the variable name `radius` in order to obtain the address of that variable. Since all three variables are of **double** type, the conversion specifier "`%lf`" is used for **scanf**().

An interactive execution of Program 8.2 is shown below.

```
    Enter the radius of the wheels in inches
    1.1
    Enter the speed of the two-wheel robot in inches per second
    1
    Enter the distance in inches for the two-wheel robot to drive
    5
```

For a Mindstorms configured as a two-wheel robot with the radius of 1.1 inches for wheels, the above execution will drive the Mindstorms forward 5 inches at the speed of 1 inch per second.

E Do Exercise 3 on page 91.

8.1.3 Estimate the Error in Distance and Use the Member Function getDistance()

In some applications it is useful to compare the specified distance of a robot's motion with the actual distance that it has moved. Such a comparison can indicate whether the robot is working as intended. The member function **getDistance()** can be used to get the distance that a Mindstorms has moved using a specific wheel radius. The general syntax of this function is

```
    robot.getDistance(distance, radius);
```

where `distance` specifies how far the Mindstorms has moved and `radius` specifies the radius of the currently attached wheels. The values of both `distance` and `radius` have the same units.

The function call

```
 robot.resetToZero();
```

can be used to set the current joint position as the zero position for all joints.

Please note that, unlike Linkbot, Mindstorms does not have a specific zero position. In Linkbot, the member function **resetToZero()** moves all of its joints to the zero position. The member function **resetToZero()** is equivalent to the **zero** button on the Robot Control Panel in Linkbot Labs, or pressing both A and B buttons of the Linkbot.

Before getting a distance using the member function **getDistance**(), the function call

```
 robot.resetToZero();
```

should be used to set the current joint position as zero position for all joints.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `errorindistance.ch` to drive the robot for 12 inches at the speed of 2.5 inches per second. Estimate the error between the specified distance and driven distance.

```
/* File: errorindistance.ch
   Estimate the error in the distance between the specified and drived distances
   using robot.getDistance() */
#include <mindstorms.h>
CMindstorms robot;
double speed = 2.5;        // speed in inches/second
double radius = 1.1;      // radius of the wheel
double distance = 12;      // distance to drive in inches
double distance2;          // distance drived

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* get the drive distance */
robot.getDistance(distance2, radius);

printf("The distance to drive is %.2lf inches.\n", distance);
printf("The actual distance drived is %.2lf inches.\n", distance2);
printf("The error is %lf inches.\n", distance-distance2);
printf("The error is %lf percent.\n", (distance-distance2)/distance *100);
```

Program 8.3: Estimate the error between the specified distance and driven distance using **getDistance**().

The member function **getDistance()** is used to get the actual distance in inches that was traveled by the Mindstorms. The error of the distance that a robot has moved is defined as

$$error = (distance\ to\ be\ moved) - (distance\ moved)$$

The percentage of the error of the distance that a robot has moved is defined as

$$percentage\ of\ error = \frac{(distance\ to\ be\ moved) - (distance\ moved)}{(distance\ to\ be\ moved)} \times 100$$

When Program 8.3 is executed, the following output will be displayed in the input/output pane

```
The distance to drive is 12.00 inches.
The actual distance driven is 11.98 inches.
The error is 0.020000 inches.
The error is 0.016667 percent.
```

The output indicates that the robot actually drives about 9.98 inches, with the error of 0.02 inches and 0.16667 percent of the distance to drive.

In the following four scenarios, the member function **resetToZero()** should be called to set its joints to zero positions first.

1. The program will call the member function moveTo() or moveJointTo().
2. The program will getJointAngle(), getJointAngles(), or getDistance().
3. The program will call a recording member function starting with the prefix **record**.
4. The program will control a system connected with multiple Mindstorms.

As Program 8.3 uses the member function **getDistance**(), the member function **resetToZero**() needs to be called first. We will learn other cases in next section and later chapters.

E Do Exercise 4 on page 91.

8.1.4 ‡ Use the Functions **distance2angle**() and **angle2distance**()

In the previous section, we use the member function **getDistance**() to get the distance that a Mindstorms has moved. In this section, we will an alternative method to get the distance. As we learned in Chapter 10, we can get the joint angle after a Mindstorms has stopped moving using the member function **getJointAngle()**. We can convert this joint angle value into the actual distance that the Mindstorms has moved using the function **angle2distance**(). The general syntax of this function is

```
distance = angle2distance(radius, angle);
```

where `radius` is the radius of the wheels attached to the Mindstorms and `angle` is the joint angle in degrees. The distance returned will be in the same units as `radius`. The function **angle2distance**() is implemented in Ch with the code

```
double angle2distance(double radius, double angle) {
    return radius*(angle * M_PI/180);
}
```

It is also possible to convert a distance value into a joint angle value using the counterpart function **distance2angle**(), which has the following syntax

```
angle = distance2angle(radius, distance);
```

where `radius` is the radius of the wheels attached to the Mindstorms and `distance` is the distance a Mindstorms has traveled. Both `radius` and `distance` should have the same units. The value `angle` returned from this function is in degrees. The function **distance2angle**() is implemented in Ch as follows

```
    double distance2angle(double radius, double distance) {
        return (distance/radius)*180/M_PI;
    }
```

We will solve the same problem presented in the previous section using the member function **getJointAngle()** and function **angle2distance**().

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `errorindistance.ch` to drive the robot for 12 inches at the speed of 2.5 inches per second. Estimate the error between the specified distance and driven distance.

```
/* File: errorindistance2.ch
   Estimate the error in the distance between the specified and drive distances
   usig robot.getJointAngle() and angle2distance() */
#include <mindstorms.h>
CMindstorms robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.1;       // radius of the wheel
double distance = 12;       // distance to drive in inches
double angle;               // angle corresponding to the drived distance in degrees
double distance2;           // distance drived based on the angle

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* obtain the angle for joint 2 */
robot.getJointAngle(JOINT2, angle);

/* calculate the distance based on the joint angle */
distance2 = angle2distance(radius, angle);

printf("The distance to drive is %.2lf inches.\n", distance);
printf("The actual distance drived is %.2lf inches.\n", distance2);
printf("The error is %lf inches.\n", distance-distance2);
printf("The error is %lf percent.\n", (distance-distance2)/distance *100);
```

Program 8.4: Estimate the error between the specified distance and driven distance using **angle2distance**().

The motion statement

```
    robot.driveDistance(distance, radius);
```

is equivalent to the statements

```
    angle = distance2angle(radius, distance);
    robot.driveAngle(angle);
```

The motion statement

```
    robot.getDistance(distance, radius);
```

is equivalent to the statements

```
    robot.getJointAngle(JOINT1, angle);
    distance = angle2distance(radius, angle);
```

The member function **getJointAngle()** is used to get the value of the joint angle after the Mindstorms has stopped moving, then the statement

```
    distance2 = angle2distance(radius, angle);
```

gives the actual distance in inches that were traveled by the Mindstorms.

When Program 8.4 is executed, the output will be similar to that of Program 8.3.

E Do Exercise 5 on page 91.

8.1.5 Measure the Clock Time Using the Member Function systemTime()

You can time the motion of a Mindstorms using the member function **systemTime**(). The syntax of the member function **systemTime**() is as follows.

```
    robot.systemTime(time);
```

This member function passes the time in seconds since 00:00:00 January 1, 1970, on Mac OS X and Linux systems. In Windows, this function returns the time in seconds since the system last started.

The member function **systemTime**() can be used in many other applications. One example is measuring the time it takes for a robot to complete its movement with a specified speed and distance.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `gettime.ch` to drive the robot forward 10 inches at the speed of 2.5 inches per second. The program shall measure how long it takes for the robot to complete the motion.

```
/* File: gettime.ch
   Get the time to drive the vehicle with a specified speed and distance */
#include <mindstorms.h>

CMindstorms robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.1;       // radius of the wheel
double distance = 10;       // distance in inches
double time1, time2, elapsedtime;    // system time and elapsed time

/* set the robot speed */
robot.setSpeed(speed, radius);

robot.systemTime(time1);       // get the system time since the system starts
/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);
robot.systemTime(time2);       // get the system time since the system starts
elapsedtime = time2 - time1;   // Calculate the time for the motion.

printf("The motion for the robot took %.2lf seconds\n", elapsedtime);
printf("The motion should take %.2lf seconds in theory.\n", distance/speed);
```

Program 8.5: Get the time for a two-wheel robot to complete a motion with a specified speed and distance using the member function **systemTime**().

Similar to what was done in Program 11.5, the statement

```
    robot.systemTime(time1);    //the system time since the system starts
```

is called before the Mindstorms starts moving in order to record the current system time in seconds. Then the statement

```
    robot.driveDistance(distance, radius);
```

drives the Mindstorms the specified distance of 10 inches at the set speed of 2.5 inches/second. Then the statement

```
    robot.systemTime(time2);    //the system time since the system starts
```

is called to record the system time in seconds after the Mindstorms stops moving. The following statement

```
    elapsedtime = time2 - time1;        //calculate the time for the motion.
```

calculates the difference between the end time and the start time, to give the total time the Mindstorms was in motion. This time is then displayed, along with the theoretical time. In the case of Mindstorms movement with a specified speed and distance, the theoretical time is defined as

$$theoretical\ time = \frac{(distance\ in\ inches)}{(speed\ in\ inches\ per\ second)}$$

When Program 8.5 is executed, the following output will be displayed in the input/output pane

```
    The motion for the Mindstorms took 4.12 seconds.
    The motion should take 4.00 seconds in theory.
```

E Do Exercise 6 on page 91.

8.1.6 Summary

1. Call the **CMindstorms** member function

   ```
       robot.setSpeed(speed, radius);
   ```

   to set both ports B and C of a Mindstorms to the desired speed and wheel radius.

2. Call the member function

   ```
       robot.systemTime(time);
   ```

   to get the system time in seconds since the system was last started.

3. Call the function

   ```
       distance = angle2distance(radius, angle);
   ```

   to convert a joint angle value to a distance.

4. Call the function

   ```
       angle = distance2angle(radius, distance);
   ```

   to convert a distance to a joint angle value.

8.1.7 Terminology

angle2distance(), **distance2angle**(), actual time, theoretical time, **robot.setSpeed()**. **plot.systemTime()**,

8.1.8 Exercises

1. Watch the video tutorial "R2. Set the Speed: setSpeed()" in http://roboblockly.ucdavis.edu/videos.

2. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `setspeed2.ch` to drive forward 360 degrees using the member function **driveAngle**() at the speed of 2.5 inches per second, and then drive backward 3 inches using the member function **driveDistance**().

3. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Use the program `drivedistance_p.ch` in Program 8.2 to make the Mindstorms move 12 inches in 5 seconds (Note that the speed is defined as the distance divided by the time).

4. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `errorindistance3.ch` to accept the user input of the radius of wheels, speed, and distance for moving the robot using the member function **driveDistance**(). The program shall estimate the distance that the robot has moved using the member function **getDistance**(). It shall also calculate the error and percentage of error of the distance moved. (a) Test your program with the input speed of 2.5 inches per second and distance of 12 inches. (b) Test your program with the input speed of 2.5 inches per second and distance of 2 inches.

5. Solve the same problem described in Exercise 4 without using the member function **getDistance**().

6. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `gettime2.ch` to drive the robot for 12 inches at the speed of 3.2 inches per second. The program shall measure how long it takes for the robot to complete the motion using the member function **systemTime**(). Also try to measure this motion using a stopwatch. Is the time measured by the program `gettime2.ch` the same as that on a stopwatch? Why?

8.2 Plot a Curve Using the Plotting Member Functions scattern() and data2DCurve()

A picture is worth a thousand words. Graphical plotting is useful for visualization and understanding many problems in engineering and science. Graphical plotting can be conveniently accomplished in Ch. The member function **scattern**() of the plotting class CPlot can be used to plot data in arrays in a scatter plot

```
    plot.scattern(x, y, n);
```

The array x stores data for the x-axis, whereas the array y stores data for the y-axis. Both arrays x and y will have the same number of elements, which is specified by the third argument `n` of integral value. The member function **data2DCurve**() of the plotting class CPlot can be used to plot data in arrays in a line plot

```
    plot.data2DCurve(x, y, n);
```

The arguments of **data2DCurve**() are the same as those of **scattern**(). The plot generated by the member function **data2DCurve**() will connect each two adjacent points by a line.

Problem Statement:
The time and corresponding positions of a robot are recorded in Table 8.1 from an experiment.

Write a program to plot the trajectory of the robot based on this table (a) in a scatter plot and (b) in a line plot.

Table 8.1: Positions of a robot versus time.

time (seconds)	0.00	2.00	4.00	6.00	8.00	10.00
position (meters)	1.25	1.1	2.25	2.75	3.25	3.75

```
/* File: scattern.ch
   Plot the positions of a robot versus time for 6 points of data in arrays */
#include <chplot.h>    /* for the function plotxy() */

CPlot plot;
/* declare two sets of arrays with 6 points for plottting for p versus t */
double t[6] = {0.00, 2.00, 4.00, 6.00, 8.00, 10.00};
double p[6] = {1.25, 1.75, 2.25, 2.75, 3.25, 3.75};

plot.title("Position Plot");
plot.label(PLOT_AXIS_X, "time (seconds)");
plot.label(PLOT_AXIS_Y, "position (meters)");
plot.scattern(t, p, 6);
plot.plotting();
```

Program 8.6: Plotting positions versus time in a scatter plot for motion of a robot using arrays.

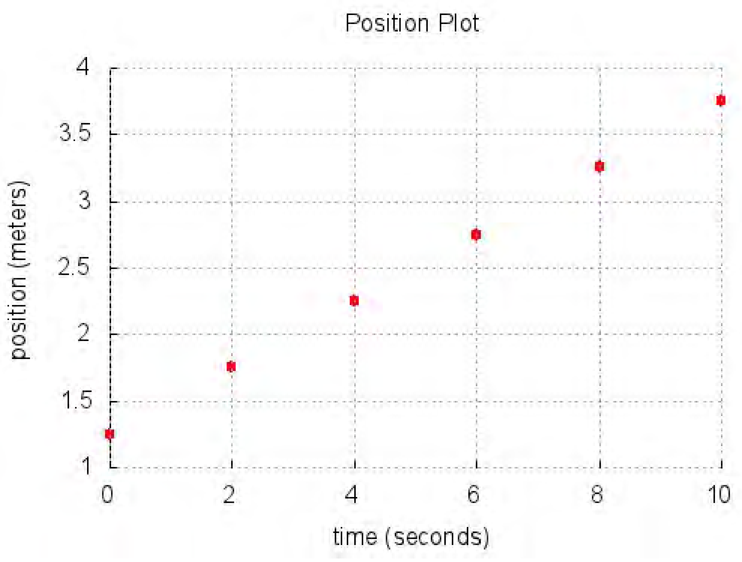


Figure 8.2: The scatter plot for the position versus time from Program 8.6.

As seen in Program 8.6, a program generating a plot typically contains the following statements:

```
#include <chplot.h>
CPlot plot;
plot.title("title");
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
/* add plotting data here */
plot.plotting();
```

The line

```
#include <chplot.h>
```

includes the header file **chplot.h**. The purpose of including the header file **chplot.h** is to use the class **CPlot** defined in this header file. As we have learned in previous chapters, a class is a user defined data type in Ch. The symbol **CPlot** can be used in the same manner as the symbol **int** or **double** to declare variables.

The following lines

```
double t[6] = {0.00, 2.00, 4.00, 6.00, 8.00, 10.00};
double p[6] = {1.25, 1.1, 2.25, 2.75, 3.25, 3.75};
```

consist of the data that will be plotted. The array with the variable name `t` holds six time values, and the array with the variable name `p` holds the six corresponding distance values. Both are of type **double** since we are plotting time and distance, which are typically written as decimal numbers. For generating the plot, `t` will be the x-axis values and `p` will be the y-axis values.

The next line

```
CPlot plot; //plotting class
```

declares the variable `plot` of type **CPlot** for plotting. A function associated with a class is called a ***member function***. For example, the function **plot.title()** or **title()** is a member function of the class **CPlot**. In Program 8.6, member functions of the class **CPlot** are called to process the data for the object `plot`.

The function call

```
plot.title("title");
```

adds a title to the plot. The argument for this member function is a string. If this member function is not called, the generated plot will have no title. In Program 8.6, the line

```
plot.title("Position Plot");
```

adds the appropriate label for our intended purpose.

The subsequent two function calls

```
plot.label(PLOT_AXIS_X, "xlabel");
plot.label(PLOT_AXIS_Y, "ylabel");
```

add labels to the x and y coordinates. The macros **PLOT_AXIS_X** and **PLOT_AXIS_Y** for the x and y axes, respectively, are defined in the header file **chplot.h**. The second arguments of the above two member functions are also strings for labels. If these two functions are not called, by default, the label for the x-axis will be "x" whereas the label for the y-axis will be "y". Thus the lines in Program 8.6

```
plot.label(PLOT_AXIS_X, "time (seconds)");
plot.label(PLOT_AXIS_Y, "position (meters)");
```

add the correct labels for position versus time.

The next line

```
plot.data2DCurve(t, p, 6);
```

plots the trajectory of the Mindstorms based on the six data points from Table 8.1.

Finally, after the plotting data are added, the program needs to call the function

```
 plot.plotting();
```

to generate a plot. The generated graph is shown in Figure 8.2.

To generate a line plot with the same data listed in Table 8.1, we can just replace the function call in Program 8.6

```
   plot.scattern(t, p, 6);
```

by the statement

```
   plot.data2DCurve(t, p, 6);
```

as shown in Program 8.7. The generated line plot by Program 8.7. is shown in Figure 8.3.

```
/* File: posplot.ch
   Plot the positions of a robot versus time for 6 points of data in arrays */
#include <chplot.h>    /* for the function plotxy() */
CPlot plot;
/* declare two sets of arrays with 6 points for plottting for p versus t */
double t[6] = {0.00, 2.00, 4.00, 6.00, 8.00, 10.00};
double p[6] = {1.25, 1.75, 2.25, 2.75, 3.25, 3.75};

plot.title("Position Plot");
plot.label(PLOT_AXIS_X, "time (seconds)");
plot.label(PLOT_AXIS_Y, "position (meters)");
plot.data2DCurve(t, p, 6);
plot.plotting();
```

Program 8.7: Plotting positions versus time in a line plot for motion of a robot using arrays.

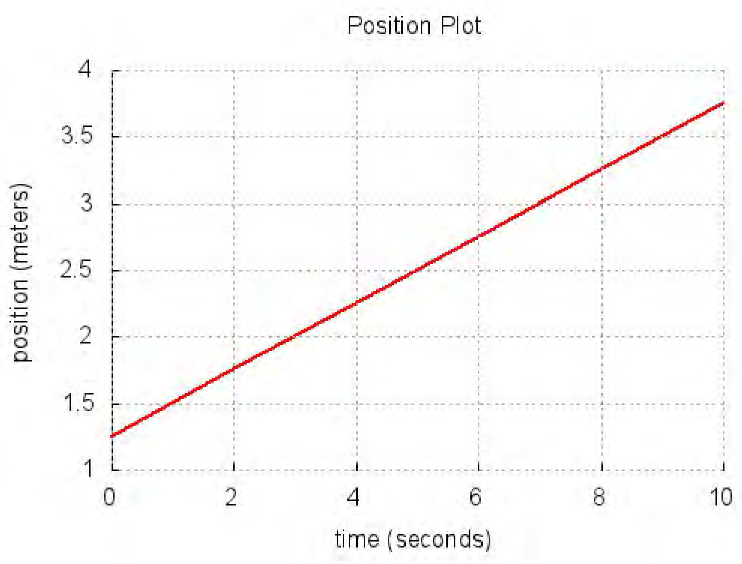


Figure 8.3: The line plot for the position versus time from Program 8.7.

E Do Exercise 1 on page 96.

You may read section 9.1 in Appendix 9 to learn how to plot points and lines using member functions of the plotting class **CPlot**.

8.2.1 Summary

1. Include the header file **chplot.h** and use the class **CPlot** to declare a variable `plot` by the following two statements

```
    #include<chplot.h>
    CPlot plot;
```

2. Call the **CPlot** member function

```
    plot.title("title");
```

to add a title to the plot.

3. Call the **CPlot** member functions

```
    plot.label(PLOT_AXIS_X, "xlabel");
    plot.label(PLOT_AXIS_Y, "ylabel");
```

to label the x and y axes of the graph.

4. Call the **CPlot** member function

```
    plot.scattern(x, y, n);
```

to plot n data points stored in arrays x and y in a scatter plot.

5. Call the **CPlot** member function

   ```
   plot.data2DCurve(x, y, n);
   ```

   to plot n data points stored in arrays x and y in a line plot.

6. Call the **CPlot** member function

   ```
   plot.plotting();
   ```

   to generate the final graph.

8.2.2 Terminology

#include <chplot.h>, **CPlot**, **plot.title()**, **plot.label()**, **plot.scattern()**, **plot.data2DCurve()**, **plot.plotting()**.

8.2.3 Exercises

1. When a soccer ball is kicked on the ground, the time and corresponding positions of the soccer ball are recorded in Table 8.2. Based on the data in the table, plot the trajectory of the soccer ball.
   - Write a program `scattern2.ch` using the member function **scattern()** to plot the trajectory in a scatter plot.
   - Write a program `posplot2.ch` using the member function **data2DCurve()** to plot the trajectory in a line plot.

Table 8.2: Positions of a soccer ball versus time.

time (seconds)	0	0.2	0.4	0.6	0.8	1	1.2	1.4	1.6	1.8	2	2.2	2.4	2.6	2.8	3
position (meters)	0	2.8	5.2	7.2	8.9	10.1	10.9	11.4	11.5	11.1	10.4	9.3	7.8	5.9	3.6	0.9

8.3 Plot Distances versus Time

In the previous section data is provided to the program specifically. In this section we will learn how to write a program to receive data from a Mindstorms in real time. The ability to record and process real time data is a powerful tool with multiple applications in robotics. It aids in the understanding of how a robot is functioning so that we can improve upon that robot's design. Real time data acquisition also allows a robot to interact with its environment through sensors or user input. An example would be a robot that can receive and respond to voice commands using an audio sensor. Another example would be a robot that can recognize and react to its surroundings using a visual sensor.

In this section, we will learn how to plot the distance values of robot travelled versus time with real-time data.

8.3.1 Plot Distances versus Time for a Two-Wheel Mindstorms with the Specified Speed and Distance

We can record and plot the distance versus time for a Mindstorms. To record distance data, we use the two **CMindstorms** member functions **recordDistanceBegin()** and **recordDistanceEnd()**.

The general syntax for the function **recordDistanceBegin()** is

```
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

The first and second arguments `timedata` and `distances` are variables of type **robotRecordData_t**. The reason for this is the same as why the `timedata` and `angledata` arguments are of type **robotRecordData_t** in Program 11.1. Recall that the arguments `time` and `distance` are special arrays that grow to the size needed during the execution of the program. The third argument `radius` is the radius of the wheels attached to the Mindstorms. The final argument `timeInterval` is the time interval between angle readings. The minimum possible value for `timeInteval` is 0.05 seconds.

The general syntax for the function **recordDistanceEnd()** is

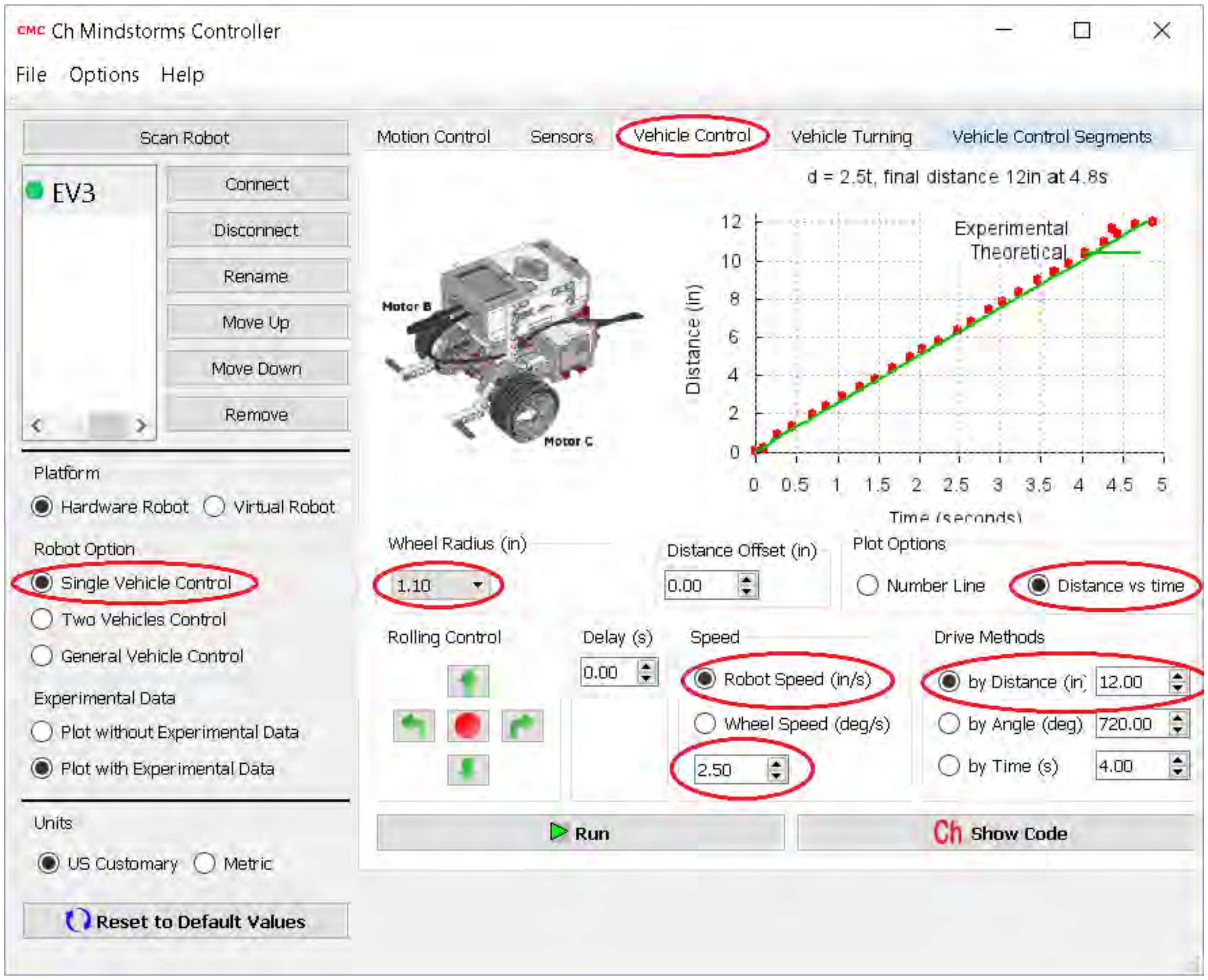
```
robot.recordDistanceEnd(num);
```

The argument `num` is the total number of data points that were recorded while the Mindstorms was moving.

We can use the functions **recordDistanceBegin()** and **recordDistanceEnd()** in combination with the member functions of the **CPlot** class we learned in Section 11.1. This will enable the recording and plotting of distance versus time for a two-wheel Mindstorms.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to port B and C. The radius of each wheel is 1.1 inches. Write a program `recorddistancescattern.ch` to drive the Mindstorms for 12 inches at the speed of 2.5 inches per second. Record the distance as the Mindstorms moves with a time interval of 0.1 second. Plot the distance versus time in a scatter plot.

Before we write a program to solve this problem. We can use Ch Mindstorms Controller (CMC) in C-STEM Studio to create the motion and generate the scatter plot for the distance versus time with real-time data, and line plot for the distance versus time with theoretical data. You can double click "Ch Mindstorms Controller" on the C-STEM Studio shown in Figure 1.3 to launch CMC. With the setup for CMC as shown in Figure 8.4, click the tab "Run", the robot will move accordingly. Once the robot finishes its motion, a picture will be displayed. The equation of the motion and final distance at the end of the time, such as ``` ``d = 2.5t, final distance 12in at 4.8s" ```, will be displayed above the picture as shown in Figure 8.4. The motion for the robot and picture in CMC is actually generated by Ch program. You can click the tab "Show Code" to see the Ch code with different options and launch the code in ChIDE directly to control the robot. We will learn how the Ch code works in the remaining section.


Ch Mindstorms Controller

File Options Help

Scan Robot

EV3

Connect
Disconnect
Rename
Move Up
Move Down
Remove

Motion Control | Sensors | Vehicle Control | Vehicle Turning | Vehicle Control Segments

d = 2.5t, final distance 12in at 4.8s

Platform: [✓ Hardware Robot] Virtual Robot

Robot Option: [✓ Single Vehicle Control] Two Vehicles Control General Vehicle Control

Experimental Data: Plot without Experimental Data [✓ Plot with Experimental Data]

Units: [✓ US Customary] Metric

Wheel Radius (in): 1.10

Distance Offset (in): 0.00

Plot Options: Number Line [✓ Distance vs time]

Delay (s): 0.00

Speed: [✓ Robot Speed (in/s)] Wheel Speed (deg/s) — 2.50

Drive Methods: [✓ by Distance (in)] 12.00; by Angle (deg) 720.00; by Time (s) 4.00

Run

Show Code

Reset to Default Values

Figure 8.4: The setup of Ch Mindstorms Controller for solving the problem.

E Do Exercise 1 on page 113.

```
/* File: recorddistancescattern.ch
   Record time and distances, plot the acquired data */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;        // speed in 2.5 inches/seconds
double radius = 1.1;      // radius of the wheel in inches
double distance = 12;      // distance in inches
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;         // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* begin recording time and distance based on joint 2 */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

/* plot the data */
plot.title("Distance versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (inches)");
plot.scattern(timedata, distances, numDataPoints);
plot.plotting();
```

Program 8.8: Plotting the distance of a Mindstorms versus time for a two-wheel robot with a specified distance in a scatter plot.

As mentioned in section 8.1.3, for recording data using member function starting with the prefix **record**, the function call

```
    robot.resetToZero();
```

should be used to set all joints to their zero positions.

Program 8.8 begin the recording of time and distance before the Mindstorms starts moving. The statement

```
    robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

starts recording data of the Mindstorms every 0.1 second. After the Mindstorms stops moving, the statement

```
    robot.recordDistanceEnd(numDataPoints);
```

stops recording data from the Mindstorms.

The aquired data are graphed as a scatter plot as shown in Figure 8.5 by the function call

```
    plot.scattern(timedata, distances, numDataPoints);
```

Program 8.9 changes the above statement in Program 8.8 to

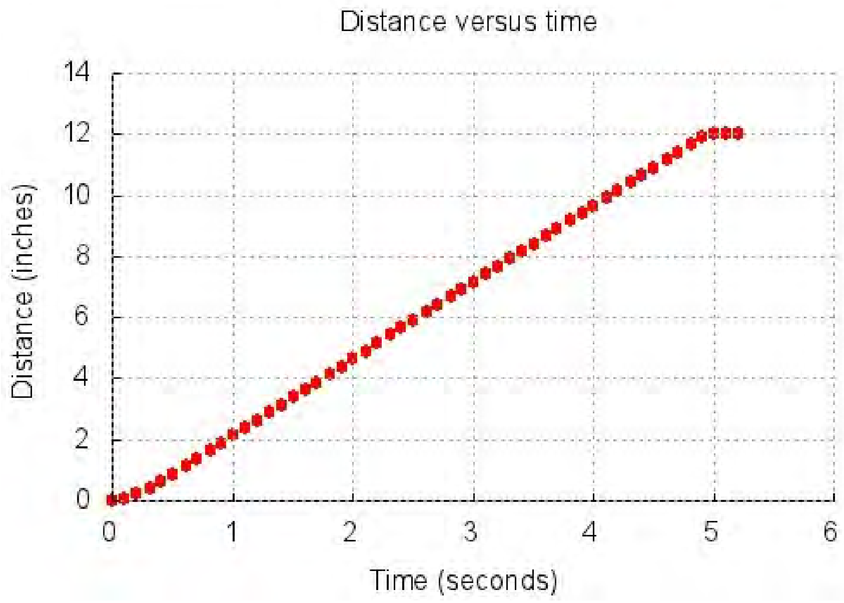


Figure 8.5: The scatter plot for the distance versus time from Program 8.8.

```
    plot.data2DCurve(timedata, distances, numDataPoints);
```

to generate a line plot. The plot generated by Program 8.9 is shown in Figure 8.6.

```
/* File: recorddistance.ch
   Record time and distances, plot the acquired data */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.1;       // radius of the wheel in inches
double distance = 12;       // distance in inches
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                 // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* begin recording time and distance based on joint 2 */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

/* plot the data */
plot.title("Distance versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (inches)");
plot.data2DCurve(timedata, distances, numDataPoints);
plot.plotting();
```

Program 8.9: Plotting the distance of a Mindstorms versus time for a two-wheel robot with a specified distance in a line plot.

The program `recorddistancescatternline.ch` uses the following two statements

```
 plot.scatter(timedata, distances, numDataPoints);
 plot.data2DCurve(timedata, distances, numDataPoints);
```

to overlay the scatter plot and line plot in a single plot as shown in Figure 8.7.

When a two-wheel robot moves at the speed of 2.5 inches per second, the relation between the distance (d) and time (t) in Figure 8.6 can be formulated by the following linear equation.

$$d = 2.5t \tag{8.1}$$

E Do Exercise 3 on page 113.

If we change the statement

```
 robot.driveDistance(distance, radius);
```

in Program 8.9 to

```
 robot.driveDistance(-distance, radius);
```

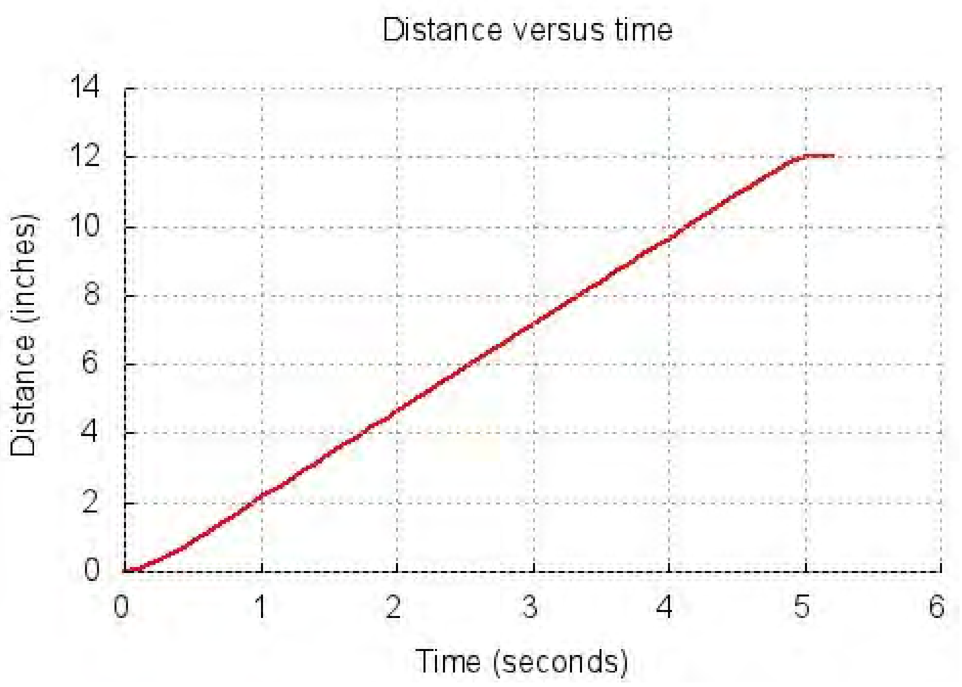


Figure 8.6: The line plot for the distance versus time from the program `recorddistance.ch`.

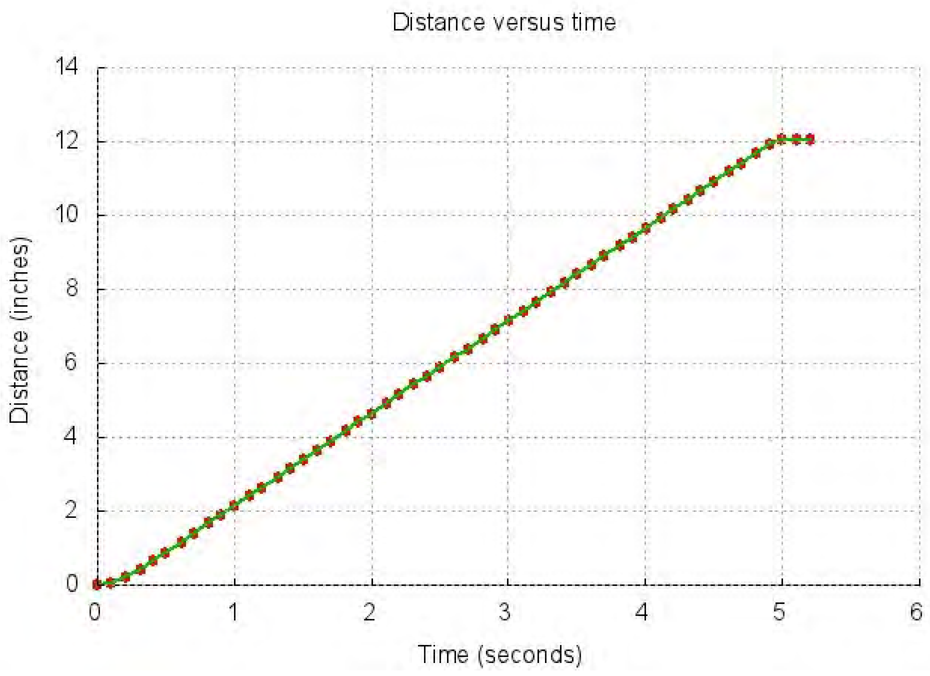


Figure 8.7: The scatter and line plots for the distance versus time from the program `recorddistancescatternline.ch`.

to drive the Mindstorms in the opposite direction. The plot generated by such a program `recorddistanceneg.ch` is shown in Figure 8.8.

If we change the statement

```
robot.setSpeed(speed, radius);
```

in Program 8.9 to

```
robot.setSpeed(-speed, radius);
```

to drive the Mindstorms in the opposite direction. The program will also generate Figure 8.8.

The relation between the distance (d) and time (t) in Figure 8.8 can be formulated by the following equation.

$$d = -2.5t \tag{8.2}$$

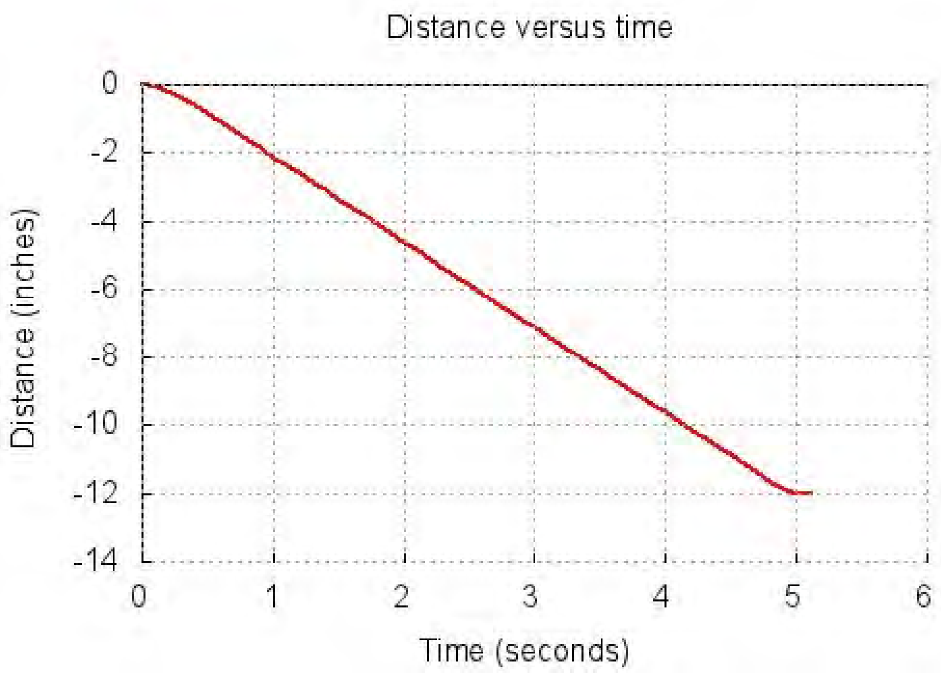


Figure 8.8: The plot for the distance versus time for a Mindstorms moving in the negative direction from the program `recorddistanceneg.ch` with a specified distance.

E Do Exercise 4 on page 115.

8.3.2 Plot Robot Distance in Number Line

As we learned in Section 6.3, we can plot robot distance on a number line to show where the robot is located after each movement. We can also plot the recorded distances on a number line with a scatter plot using the member function **plot.numberLineScattern()**. The general syntax for the function **numberLineScattern()** is

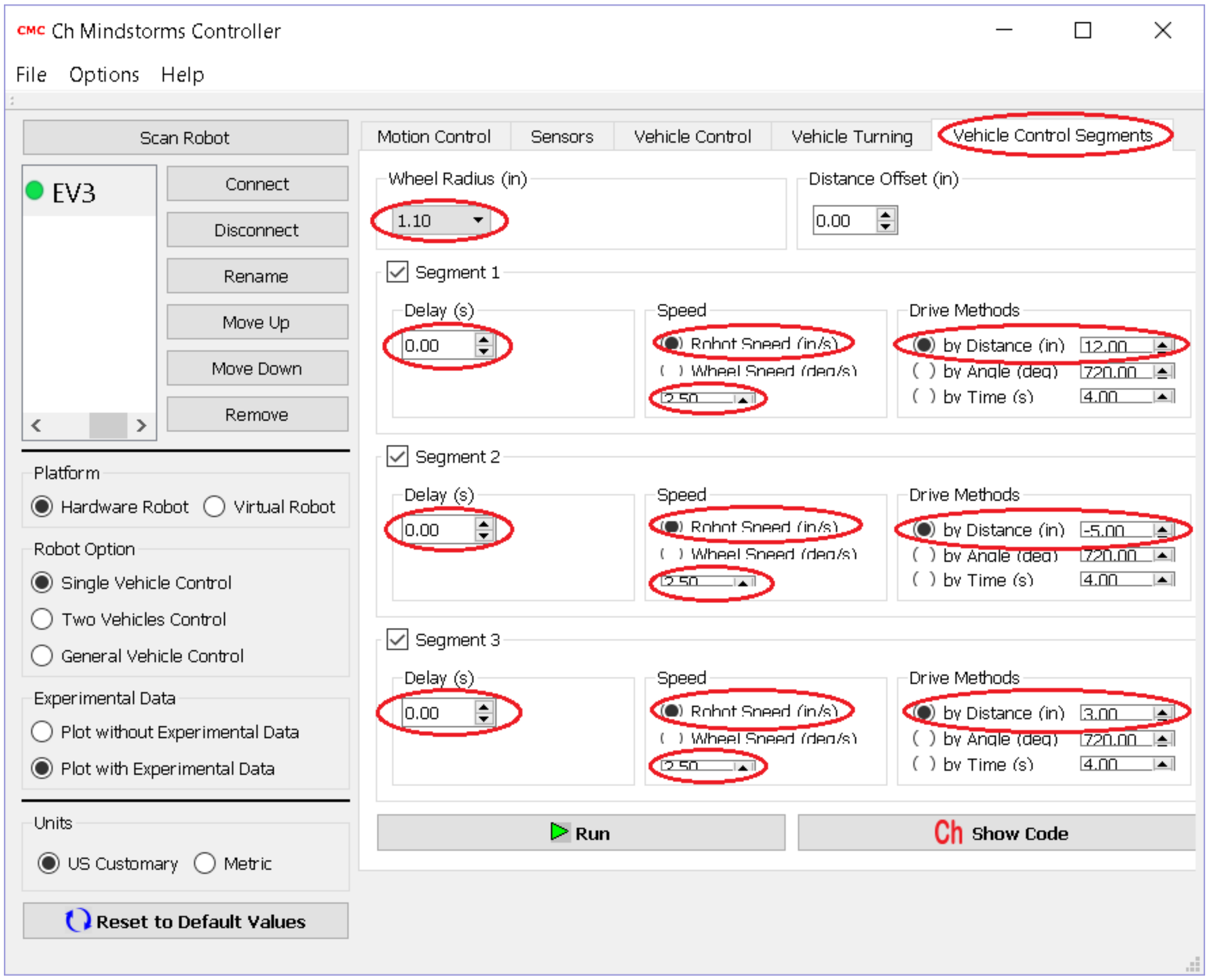
```
plot.numberLineScattern(distances, num);
```

The distance data for the first argument `distances` is acquired through the member function **robot.recordDistanceBegin()**. The second argument `num` is the total number of data points that were recorded while the robot was moving. It is passed from the member function **robot.recordDistanceEnd()**.

We can use the member functions **plot.numberLine()** and **plot.numberLineScattern()** to plot both theoretical and experimental data for distance on a number line in the same graph.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `recorddistancenumline.ch` to drive the Mindstorms from the origin at the speed of 2.5 inches per second for forward 12 inches, then backward 5 inches, and forward 3 inches again. Record the distance as the Mindstorms moves with a time interval of 0.2 second. Plot the experimental distance on a number line a scatter plot and theoretical distance on a number line in direction lines.

We can solve this problem conveniently using Ch Mindstorms Controller (CMC) with the setup shown in Figure 8.9. To control a robot with multiple movements, we need to use "Vehicle Control Segments" under Single Vehicle Control in CMC.



Wheel Radius (in): 1.10
Distance Offset (in): 0.00

☑ Segment 1
Delay (s): 0.00
Speed: (●) Robot Speed (in/s) ( ) Wheel Speed (deg/s): 2.50
Drive Methods: (●) by Distance (in) 12.00; ( ) by Angle (deg) 720.00; ( ) by Time (s) 4.00

☑ Segment 2
Delay (s): 0.00
Speed: (●) Robot Speed (in/s) ( ) Wheel Speed (deg/s): 2.50
Drive Methods: (●) by Distance (in) -5.00; ( ) by Angle (deg) 720.00; ( ) by Time (s) 4.00

☑ Segment 3
Delay (s): 0.00
Speed: (●) Robot Speed (in/s) ( ) Wheel Speed (deg/s): 2.50
Drive Methods: (●) by Distance (in) 3.00; ( ) by Angle (deg) 720.00; ( ) by Time (s) 4.00

Platform: (●) Hardware Robot ( ) Virtual Robot
Robot Option: (●) Single Vehicle Control ( ) Two Vehicles Control ( ) General Vehicle Control
Experimental Data: ( ) Plot without Experimental Data (●) Plot with Experimental Data
Units: (●) US Customary ( ) Metric

Figure 8.9: The setup of Ch Mindstorms Controller for solving the problem.

E Do Exercise 2 on page 113.

```
/* File: recorddistancenumline.ch
   Plot robot distances in number line */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;        // speed in 2.5 inches/seconds
double radius = 1.1;      // radius of the wheel in inches
double distance1 = 12;     // distance1 in inches
double distance2 = -5;     // distance2 in inches
double distance3 = 3;      // distance3 in inches
double timeInterval = 0.2; // time interval in 0.2 second
int numDataPoints;         // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* begin recording time and distance */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

robot.driveDistance(distance1, radius);
robot.driveDistance(distance2, radius);
robot.driveDistance(distance3, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

plot.numberLine(0, distance1, distance2, distance3); // 3 lines
plot.legend("Theoretical, 1st line");
plot.legend("Theoretical, 2nd line");
plot.legend("Theoretical, 3rd line");
plot.numberLineScattern(distances, numDataPoints);
plot.legend("Experimental");
plot.label(PLOT_AXIS_X, "Distance (inches)");
plot.plotting();
```

Program 8.10: Plotting the robot distance on a number line.

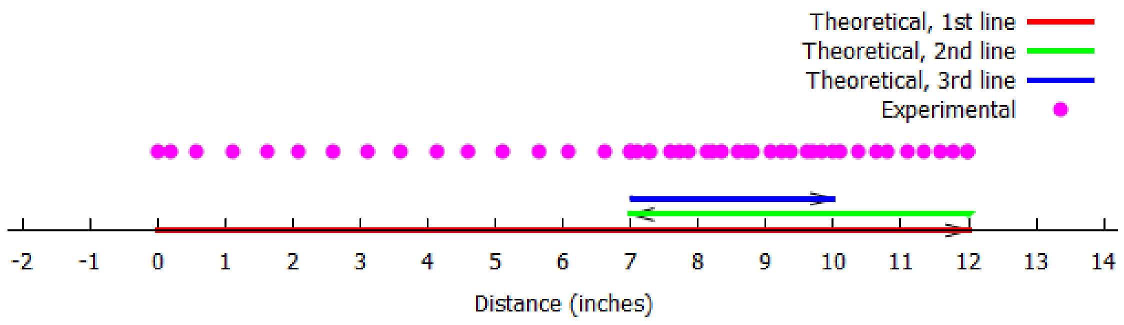


Figure 8.10: The number line for the robot distance, generated by Program 8.10.

Similar to other programs for recording distance, Program 8.10 begins the recording of time and distance before the robot starts moving. The statement

```
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

starts recording data of the Mindstorms every 0.2 second. After the Mindstorms stops moving, the statement

```
robot.recordDistanceEnd(numDataPoints);
```

stops recording data from the Mindstorms.

The theoretical distances are graphed as direction lines by the function call

```
plot.numberLine(0, distance1, distance2, distance3);
```

The statements

```
plot.legend("Theoretical, 1st line");
plot.legend("Theoretical, 2nd line");
plot.legend("Theoretical, 3rd line");
```

create legend entries for each line segment using the **CPlot** member function **legend()**. This function has the syntax

```
plot.legend(legend);
```

The argument `legend` is a string with the label you want for a particular data set.

Although the program records both distances and time, we only use distances for plotting a number line graph. The acquired distance data are graphed as a scatter plot with a legend as shown in Figure 8.10 by the function calls

```
plot.numberLineScatter(distances, numDataPoints);
plot.legend("Experimental");
```

E Do Exercise 5 on page 115.

8.3.3 Plot Distances versus Time with an Offset for the Initial Position

For plotting distances versus time with the specified speed and distance in the previous section, it is assumed that the robot is placed in the origin. We have learned in Chapter 4 that a robot can be placed at different locations in a coordinate system. In this section, we will learn how to plot the distances versus time with an initial offset for the distance. The ideas presented in this section can be used to handle the distance offset in other situations which will be described in the next section as well.

The offset for the distance of a robot can be added by the member function **recordDistanceOffset()**. The general syntax for the function **recordDistanceOffset()** is

```
robot.recordDistanceOffset(offset)
```

The argument `offset` of type **double** is the offset of the distance.

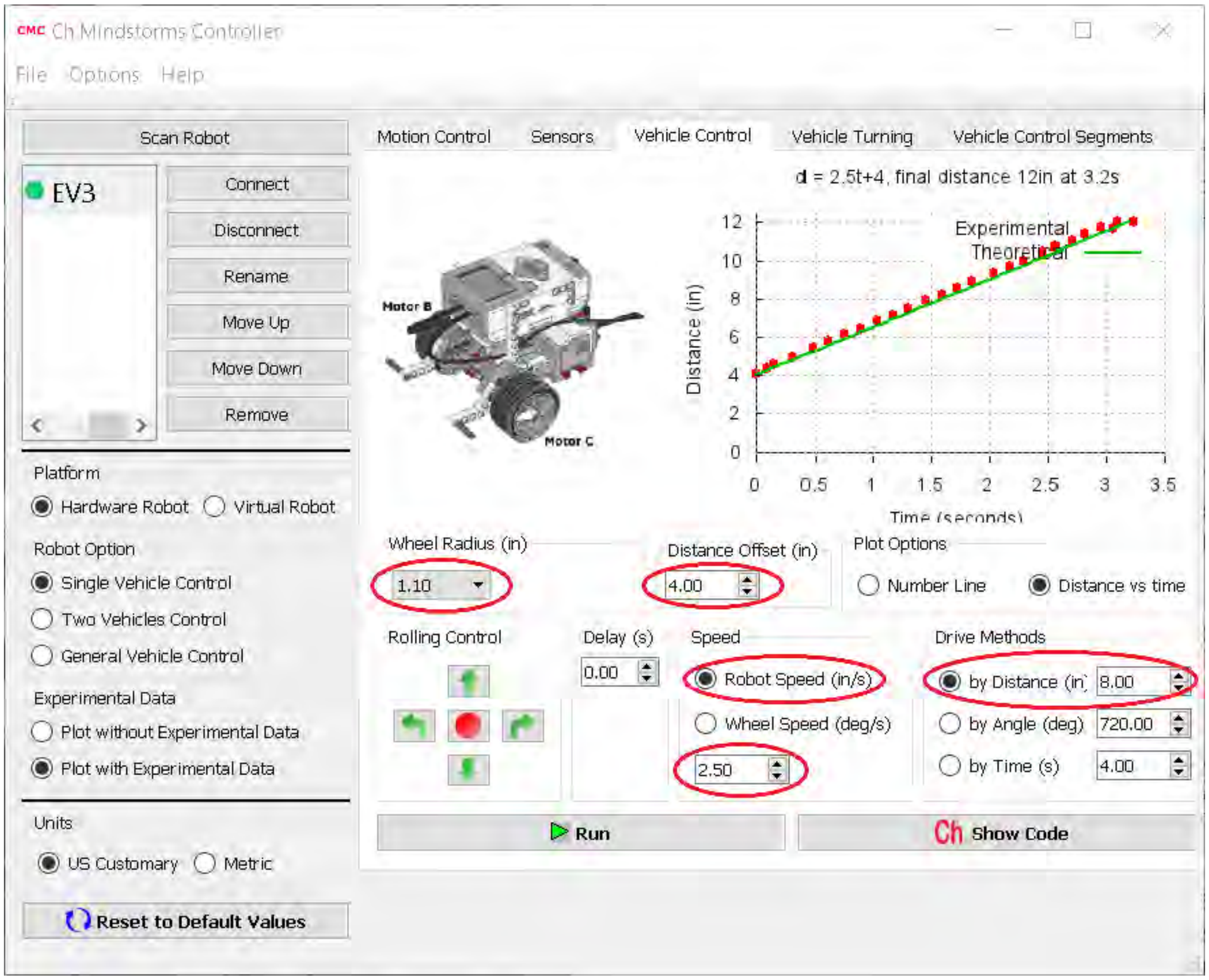
When the member function **recordDistanceBegin()** is called by

```
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

the time and distances will be recorded in the variables `timedata` and `distances`. The offset set by the member function **recordDistanceOffset**() will be added to the `distances` for each sampling point. Therefore, The member function **recordDistanceOffset**() should be called before the member function **recordDistanceBegin**() is called.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. The robot is placed in an X-Y coordinate system at the coordinate (0, 4). Write a program `recorddistanceoffset.ch` to drive the Mindstorms for 8 inches at the speed of 2.5 inches per second. Record the distance as the Mindstorms moves with a time interval of 0.1 second. Plot the distance versus time.

We can solve this problem conveniently using Ch Mindstorms Controller (CMC) with the setup shown in Figure 8.11.



Platform: [✓ Hardware Robot] Virtual Robot
Robot Option: [✓ Single Vehicle Control] Two Vehicles Control General Vehicle Control
Experimental Data: Plot without Experimental Data [✓ Plot with Experimental Data]
Units: [✓ US Customary] Metric
Wheel Radius (in): 1.10
Distance Offset (in): 4.00
Plot Options: Number Line [✓ Distance vs time]
Delay (s): 0.00
Speed: [✓ Robot Speed (in/s)] Wheel Speed (deg/s)
2.50
Drive Methods: [✓ by Distance (in) 8.00] by Angle (deg) 720.00 by Time (s) 4.00

Figure 8.11: The setup of Ch Mindstorms Controller for solving the problem.

```
/* File: recorddistanceoffset.ch
   Record time and distances with an initial offset distance by
       robot.recordDistanceOffset(offset);
   plot the acquired data */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.1;       // radius of the wheel in inches
double distance = 8;        // distance in inches
double offset = 4;          // the offset for the initial distance
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                 // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* set the offset of the distance */
robot.recordDistanceOffset(offset);

/* begin recording time and distance based on joint 2 */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

/* plot the data */
plot.title("Distance versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (inches)");
plot.axisRange(PLOT_AXIS_Y, 0, 14);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.data2DCurve(timedata, distances, numDataPoints);
plot.plotting();
```

Program 8.11: Plotting the distance of a Mindstorms versus time for a two-wheel robot with a specified distance with an initial offset for the distance.

Comparing Program 8.11 with Program 8.9, we changed the line

```
    double distance = 12;       // distance in inches
```

in Program 8.9 to

```
    double distance = 8;        // distance in inches
    double offset = 4;          // the offset for the initial distance
```

in Program 8.11 with the new distance of 8 inches and the offset of 4 inches. The offset for the distance is added to each recorded distance by the statement

```
    robot.recordDistanceOffset(offset);
```

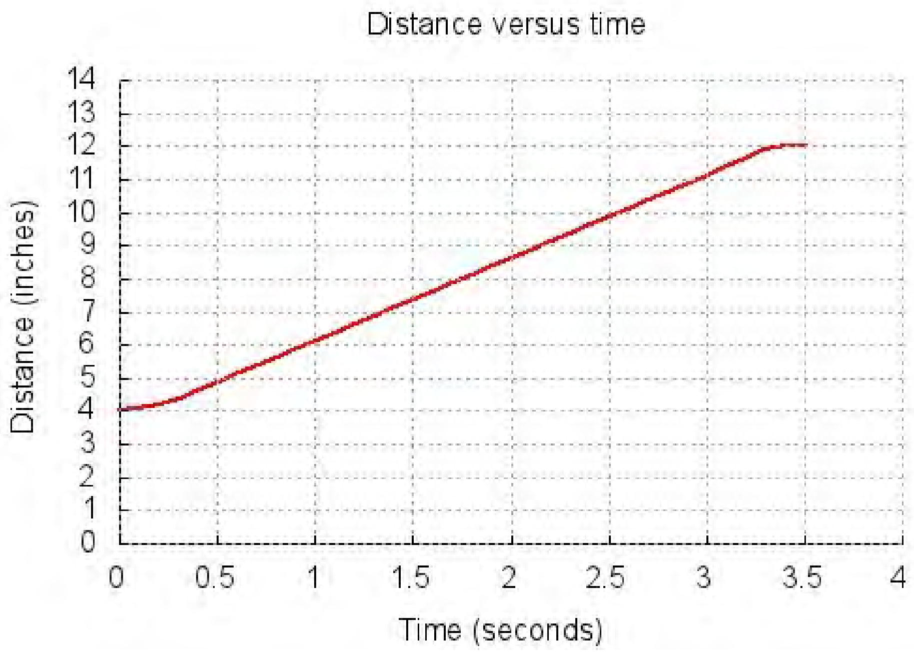


Figure 8.12: The plot for the distance versus time from Program 8.11.

When Program 8.11 is executed, the plot shown in Figure 8.12 will be displayed. The linear relation shown in Figure 8.12 can be formulated by the equation

$$d = 2.5t + 4 \tag{8.3}$$

When t is 3.2 seconds, the distance is 12 inches.

The offset is the y-intercept of this linear equation. To show the y-intercept, Program 8.11 calls the member function **axisRange** of the plotting class **CPlot** to set the range of the y axis . The general syntax of the member function **axisRange** is as follows.

```
plot.axisRange(axis, minimum, maximum);
```

The first argument `axis` specifies the axis. The macros **PLOT_AXIS_X** and **PLOT_AXIS_Y** can be used for `axis` to specify the x and y axes, respectively. The second argument `minimum` is for the minimum value on the axis. The third argument `maximum` is for the maximum value on the axis.

The tick marks on an axis can be set by the member function **ticsRange** as follows.

```
plot.ticsRange(axis, incr);
```

Like the member function **plot.axisRange()**, the first argument `axis` specifies the axis. The second argument `incr` gives the increment between tick marks.

The two statements

```
plot.axisRange(PLOT_AXIS_Y, 0, 14);
plot.ticsRange(PLOT_AXIS_Y, 1);
```

in Program 8.11 set the range for the y axis from 0 to 14 with the increment value of 1 between two tick marks.

E Do Exercise 6 on page 115.

The offset can also be created by a separate motion statement before the data recording starts. In the program `recordistanceoffset2.ch` distributed along with the other programs in this book, the statement

```
    robot.recordDistanceOffset(offset);
```

is replaced by the statement

```
    robot.driveDistance(offset, radius);
```

The program `recordistanceoffset2.ch` will generate the same plot as shown in Figure 8.12.

To run Program 8.11 in RoboSim, the robot needs to be placed at the coordinate (0, 4). However, to run the program `recordistanceoffset2.ch`, the robot should be placed at the origin (0, 0).

E Do Exercise 7 on page 116.

8.3.4 Plot Robot Distances in Number Line with an Offset for the Initial Position

For plotting distances on a number line with the specified speed and distance in section 8.3.2, it is assumed that the robot is placed in the origin. We have learned in the previous section that a robot can be placed at an offset from the origin. In this section, we will learn how to plot the distances on a number line with an offset for the initial position.

> **Problem Statement:**
> A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. The robot is placed at 2 inches in the positive direction from the origin. Write a program `recorddistancenumlineoffset.ch` to drive the robot at the speed of 2.5 inches per second for forward 12 inches, then backward 5 inches, and forward 3 inches again. Record the distance as the Mindstorms moves with a time interval of 0.2 second. Plot the experimental distance on a number line a scatter plot and theoretical distance on a number line in direction lines.

```
/* File: recorddistancenumlineoffset2.ch
   Plot robot distances in number line */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;        // speed in 2.5 inches/seconds
double radius = 1.1;      // radius of the wheel in inches
double offset = 2;         // the offset for the initial distance
double distance1 = 12;     // distance1 in inches
double distance2 = -5;     // distance2 in inches
double distance3 = 3;      // distance3 in inches
double timeInterval = 0.2; // time interval in 0.2 second
int numDataPoints;         // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

timeInterval = 0.2;

/* set the offset of the distance */
robot.recordDistanceOffset(offset);

/* begin recording time and distance */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

robot.driveDistance(distance1, radius);
robot.driveDistance(distance2, radius);
robot.driveDistance(distance3, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

plot.numberLine(0, distance1, distance2, distance3); // 3 lines
plot.legend("Theoretical, 1st line");
plot.legend("Theoretical, 2nd line");
plot.legend("Theoretical, 3rd line");
plot.numberLineScattern(distances, numDataPoints);
plot.legend("Experimental");
plot.label(PLOT_AXIS_X, "Distance (inches)");
plot.plotting();
```

Program 8.12: Plotting the robot distance on a number line with an offset for the initial position.

As we learned in the previous section, the offset for the distance of a robot can be added by the function call

```
    robot.recordDistanceOffset(offset);
```

Program 8.12 then begins the recording of time and distance before the robot starts moving. The statement

```
    robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

starts recording data of the Mindstorms every 0.2 second. After the Mindstorms stops moving, the statement

```
    robot.recordDistanceEnd(numDataPoints);
```

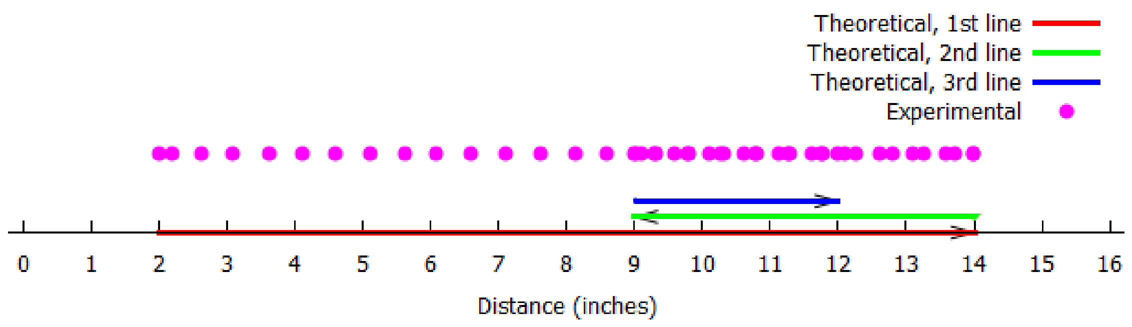


Figure 8.13: The number line for the robot distance with an offset, generated by Program 8.12.

stops recording data from the Mindstorms.

The acquired distance data are graphed as a scatter plot as shown in Figure 8.13 by the function call

```
plot.numberLineScattern(distances, numDataPoints);
```

The theoretical distances are graphed as direction lines by the function call

```
plot.numberLine(offset, distance1, distance2, distance3);
```

with an offset value 2 for the initial position of the robot.

E Do Exercise 8 on page 116.

8.3.5 Summary

1. Call the **CMindstorms** member function

   ```
   robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
   ```

   to start recording time and distance values of a Mindstorms, for a specified wheel radius and a specified interval between distance readings.

2. Call the **CMindstorms** member function

   ```
   robot.recordDistanceEnd(num);
   ```

   to stop recording time and distance values for a Mindstorms.

3. Call the **CMindstorms** member function

   ```
   robot.recordDistanceOffset(offset);
   ```

   to add an offset to the recorded distance by the member function `recordDistanceBegin()`.

4. Call the **CPlot** member function

   ```
   plot.numberLineScatern(x, n);
   ```

   to plot n data points stored in array x in a scatter plot.

5. Call the **CPlot** member function

   ```
   plot.axisRange(axis, minimum, maximum);
   ```

   to set the range of an axis.

6. Call the **CPlot** member function

```
    plot.ticsRange(axis, incr);
```

   to set the increment between tick marks for an axis.

7. Call the **CPlot** member function

```
    plot.legend(legend);
```

   to create a legend entry for an individual Mindstorms when plotting data for multiple Mindstorms.

8.3.6 Terminology

plot.axisRange(), **plot.legend()**, **plot.numbeLineScattern()**, **plot.ticsRange()**, **robot.recordDistanceBegin()**, **robot.recordDistanceEnd()**, **robot.recordDistanceOffset()**. scatter plot for distance on a number line. legend entry,

8.3.7 Exercises

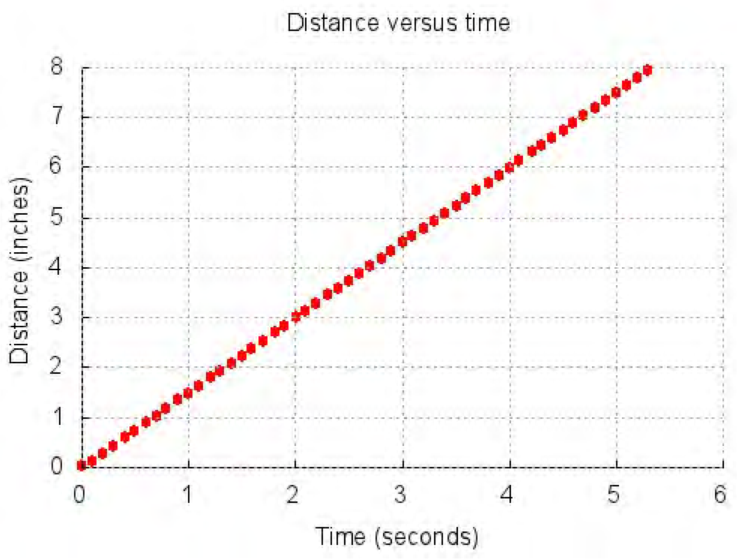
1. Watch the following video tutorials for Ch Mindstorms Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

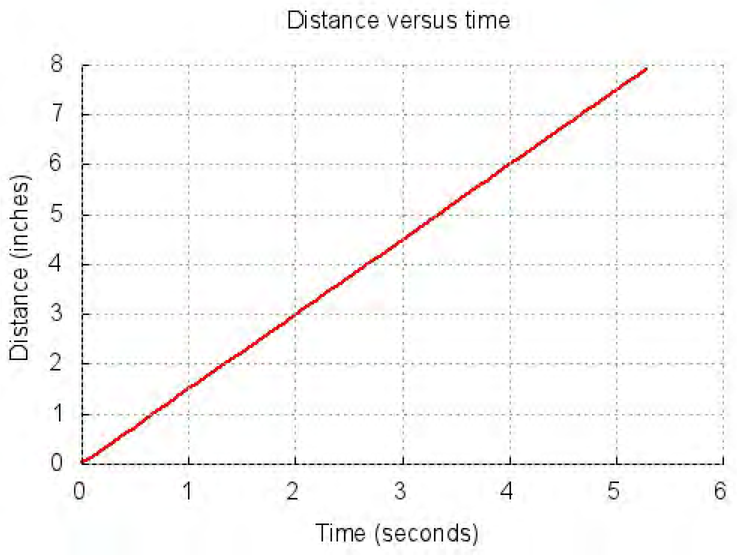
   (a) Setup for Ch Mindstorms Controller
   (b) Control a Robot with One Movement
   (c) Generating Ch Code from Ch Mindstorms Controller
   (d) Control a Robot to Turn

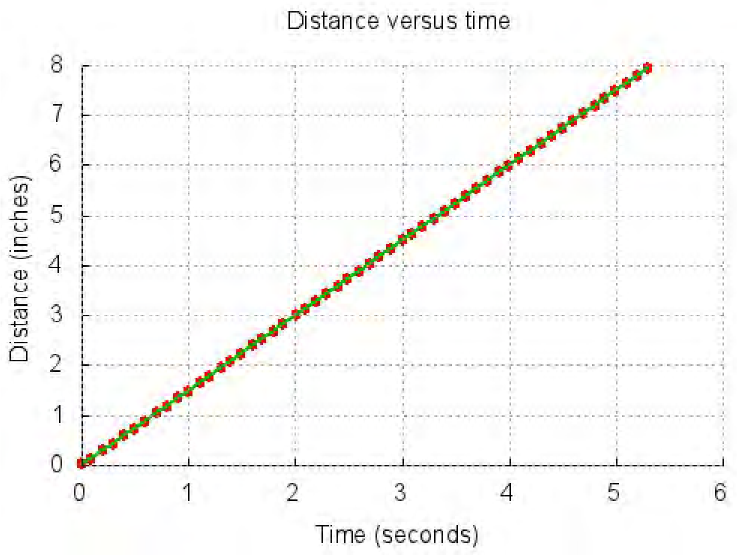
2. Watch the following video tutorial for Ch Mindstorms Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Control a Robot with Multiple Movements

3. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `recorddistancescattern2.ch` to drive the Mindstorms drives 8 inches forward at the speed of 1.5 inches per second. Record the distance as the Mindstorms travels with a time interval of 0.1 second. Plot the distance versus time as shown in the figure below in a scatter plot. Write a program `recorddistance2.ch` to generate a line plot. Write a program `recorddistancescatternline2.ch` to generate a plot with both scatter points and line as shown below. What is the equation of motion with the linear relation shown in the figure?

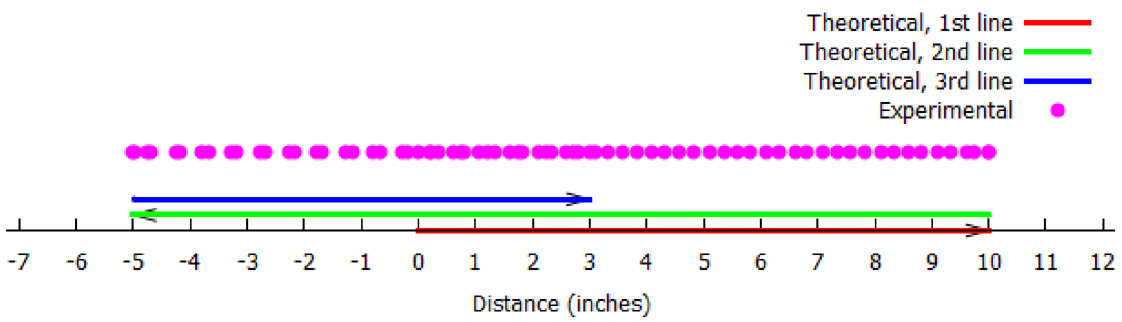

Distance versus time
Distance (inches)
Time (seconds)
0
1
2
3
4
5
6
7
8


Distance versus time
Distance (inches)
Time (seconds)
0
1
2
3
4
5
6
7
8

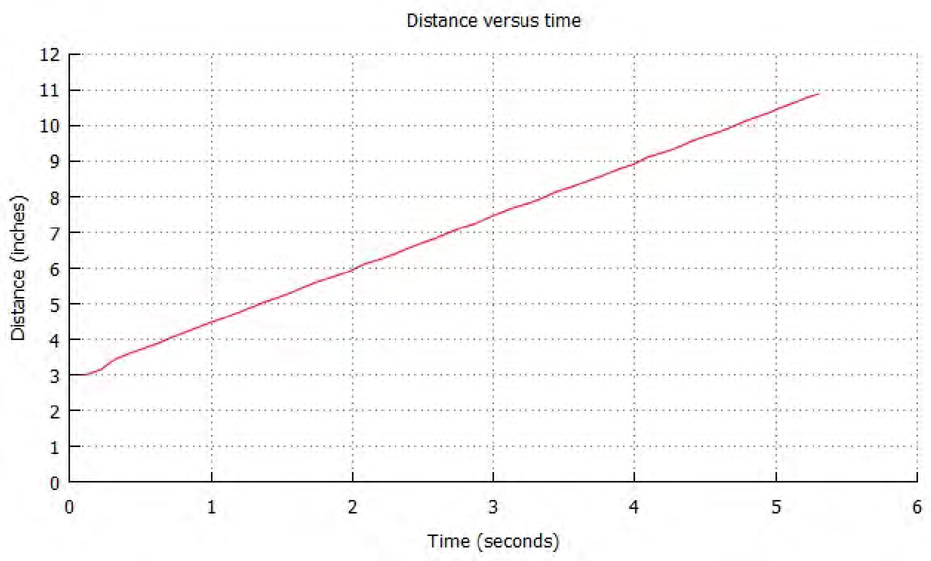


4. Modify the program `recorddistance2.ch` developed in Exercise 3 as the program `recorddistanceneg2.ch` to drive the Mindstorms 8 inches *backward*, instead of forward, at the speed of 1.5 inches per second. Plot the distance versus time. What is the equation of motion for the robot?

5. Write a program `recorddistancenumline2.ch` to control a Mindstorms configured as a two-wheel drive robot and generate a number line for the distance of the robot shown below. Assume the radius of wheels is 1.1 inches.



6. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. The robot is placed in an X-Y coordinate system at the coordinate (0, 3). Write a program `recorddistanceoffset3.ch` to drive the Mindstorms 8 inches forward at the speed of 1.5 inches per second. Record the distance as the Mindstorms travels with a time interval of 0.1 second. Plot the distance versus time as shown in the figure below. What is the equation of motion with the linear relation shown in the figure?



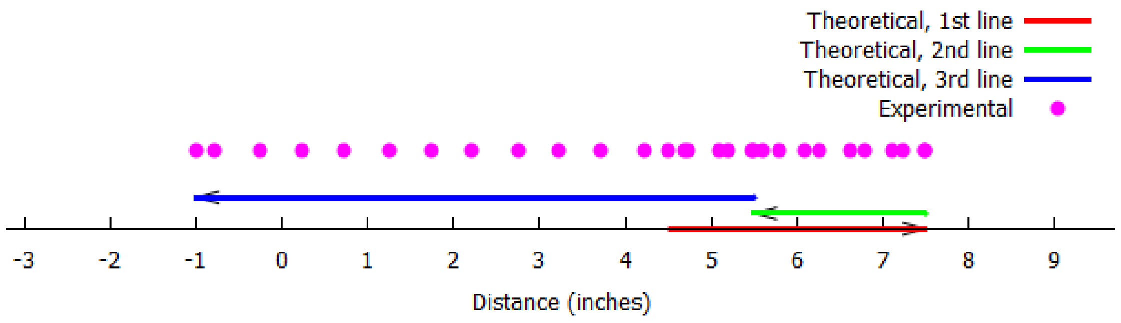
7. Modify the program `recorddistanceoffset3.ch` developed in Exercise 6 as the program `recorddistanceoffset4.ch` by replacing the statement

   ```
   robot.recordDistanceOffset(offset);
   ```

   with the statement

   ```
   robot.driveDistance(offset, radius);
   ```

8. Write a program `recorddistancenumlineoffset2.ch` to control a Mindstorms configured as a two-wheel drive robot and generate a number line for the distance of the robot shown below. The offset for the initial position of the robot is 4.5 inches from the origin. Assume the radius of wheels is 1.1 inches.



8.4 Move a Two-Wheel Robot with the Specified Time

8.4.1 Control a Mindstorms with the Speed and Time Input from the User Using the Function scanf()

In Section 8.1.2 we learned how to set the speed and distance of a Mindstorms with user input. In this section, we will see that we can also set the speed and time of a Mindstorms's motion with user input.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `drivetime_p.ch` to accept the user input of speed and time for moving the robot using the member function **driveTime**().

```
/* File: drivetime_p.ch
   Move a two-wheel robot with the user specified radius of wheels, speed, and time. */
#include <mindstorms.h>
CMindstorms robot;
double radius;      // the radius of the two wheels of the robot in inches
double speed;       // the speed in inches per second for a two-wheel robot
double time1;       // time for the movement

printf("Enter the radius of the two wheels in inches\n");
scanf("%lf", &radius);

printf("Enter the speed of the two-wheel robot in inches per second\n");
scanf("%lf", &speed);
/* set the speed for a two-wheel robot */
robot.setSpeed(speed, radius);

printf("Enter the time in seconds for the two-wheel robot to drive\n");
scanf("%lf", &time1);
/* rotate joints 1 and 3 for the specified 'time1' */
robot.driveTime(time1);
```

Program 8.13: Using the input function **scanf**() to specify the speed and time.

In Program 11.2, the member function **moveTime()** is used to move joints of a robot for a user defined amount of time. The direction of the motion depending on the sign of the speed specified for each joint. Program 8.13 uses the member function **driveTime**() to drive a Mindstorms forward or backward depending on the joint speed for joint 1 or the speed specified by the member function **setSpeed**(). The general syntax of the function **driveTime**() is

```
    robot.driveTime(seconds);
```

The argument, `seconds`, defines how long each joint will be moved in seconds. The robot moves forward if the speed for joint 1 is positive. The robot moves backward if the speed for joint 1 is negative.

The function **scanf**() is used to obtain the values for the variables `radius`, `speed`, and `time`. For a review on how to use **scanf**(), see Section 6.2.

An interactive execution of Program 8.13 is shown below.

```
    Enter the radius of the wheels in inches
    1.1
    Enter the speed of the two-wheel robot in inches per second
    1
    Enter the time in seconds for the two-wheel robot to drive
    10
```

For a Mindstorms configured as a two-wheel robot with the radius of 1.1 inches for wheels, the above execution will move the Mindstorms at the speed of 1 inch per second for 10 seconds.
E Do Exercises 1 and 2 on page 123.

8.4.2 Get the Moved Distance Based on the Specified Speed and Time

In Section 8.1.3 we learned how to get the actual distance traveled by a Mindstorms using **getDistance**(). This distance was compared with the distance specifically given to the program to determine the error in distance. In this section the function **getDistance**() will be used to determine the distance traveled by a Mindstorms when a distance is not specifically given in the program.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `getdistance.ch` to drive the robot forward 5.5 seconds at the speed of 2.5 inches per second. The program shall measure the distance that the robot has driven.

The distance to drive can be calculated using the formula

$$d = speed * t$$

with the specified speed and time `t`.

```
/* File: getdistance.ch
   Get the distance driven in the specified speed and time using robot.getDistance() */
#include <mindstorms.h>
CMindstorms robot;
double speed = 2.5;        // speed in inches/second
double radius = 1.1;      // radius of the wheel
double time1 = 5.5;        // 5.5 seconds
double distance;           // distance traveled

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* rotate joints 1 and 3 for the specified 'time' */
robot.driveTime(time1);

/* get the distance driven */
robot.getDistance(distance, radius);

printf("The distance driven is %.2lf inches.\n", distance);
printf("The distance to drive is %.2lf inches in theory.\n", speed*time1);
```

Program 8.14: Get the distance of a two-wheel robot based on the specified speed and time using **angle2distance**().

Program 8.3 used the member function **driveDistance()** to drive the Mindstorms for a specified distance. Program 8.14 is very similar to Program 8.3. The only difference in Program 8.14 is that the member function **driveTime()** is used to drive the Mindstorms for a specified period of time. Because a distance was not specifically given to the program, the actual distance traveled is obtained using the function **getDistance**(). A theoretical distance is calculated using the variables `speed` and `time`, for comparison to the actual distance moved.

When Program 8.14 is executed, the following output will be displayed in the input/output pane

```
    The distance driven is 13.78 inches.
    The distance to drive is 13.75 inches in theory.
```

E Do Exercise 3 on page 123.

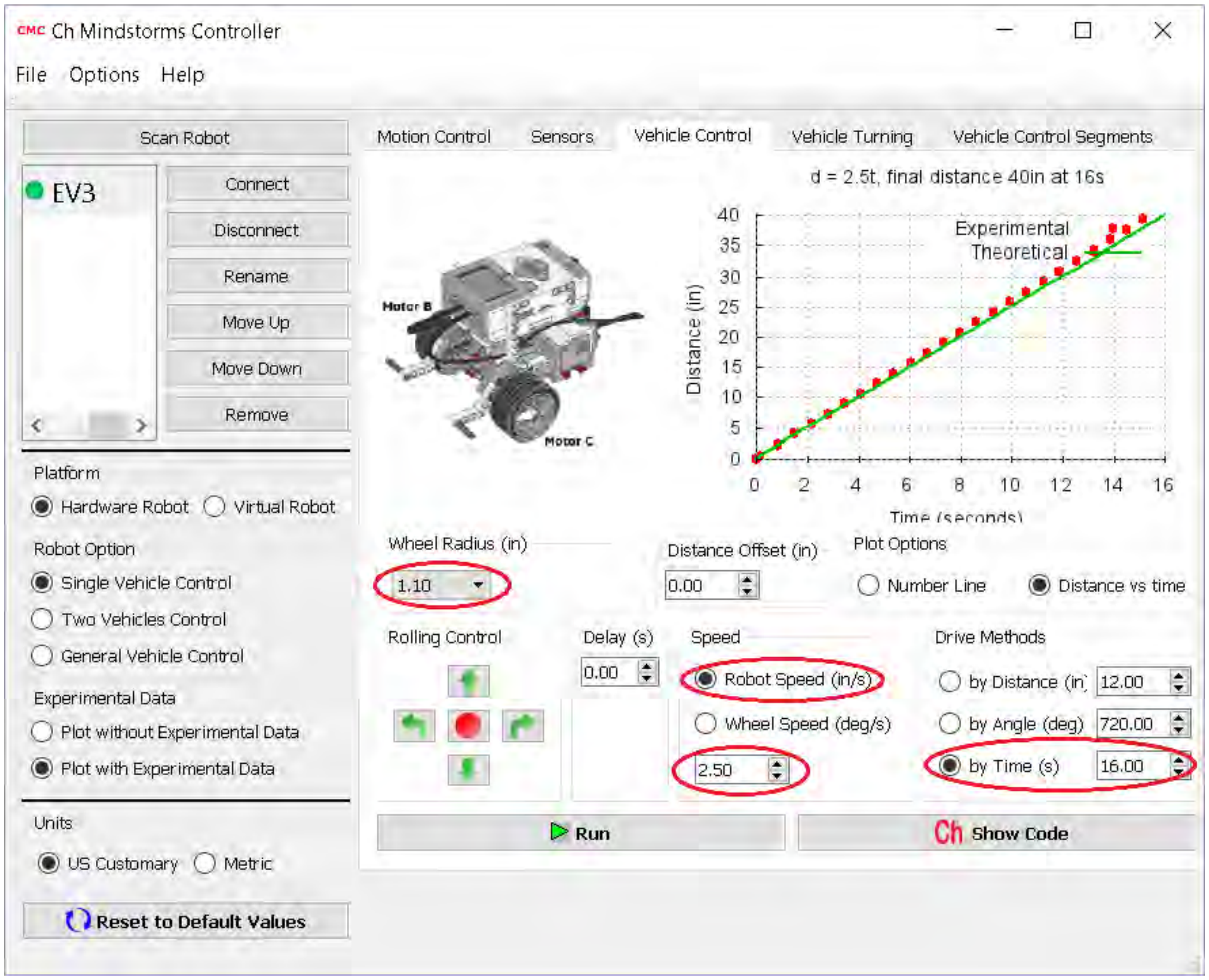
8.4.3 Plot Distances versus Time for a Two-Wheel Mindstorms with the Specified Speed and Time

In Section 8.3.1 we recorded and plotted distance versus time for a specified distance. In this section we will record and plot distance versus time for a specified speed. To do this we can use the member functions **recordDistanceBegin()** and **recordDistanceEnd()** that were introduced in Section 8.3.1. These functions are versatile and can be used to record distance data for a Mindstorms performing many different kinds of motions.

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 1.1 inches. Write a program `recordspecifytime.ch` to drive the

Mindstorms for 16 seconds at the speed of 2.5 inches per second. Record the distance as the Mindstorms moves with a time interval of 0.1 second. Plot the distance versus time.

We can solve this problem using Ch Mindstorms Controller (CMC) with the setup shown in Figure 8.14 to solve this problem conveniently.



Platform: [✓ Hardware Robot] Virtual Robot

Robot Option: [✓ Single Vehicle Control] Two Vehicles Control General Vehicle Control

Experimental Data: Plot without Experimental Data [✓ Plot with Experimental Data]

Units: [✓ US Customary] Metric

Wheel Radius (in): 1.10

Distance Offset (in): 0.00

Plot Options: Number Line [✓ Distance vs time]

Delay (s): 0.00

Speed: [✓ Robot Speed (in/s)] Wheel Speed (deg/s) 2.50

Drive Methods: by Distance (in) 12.00 by Angle (deg) 720.00 [✓ by Time (s)] 16.00

Figure 8.14: The setup of Ch Mindstorms Controller for solving the problem.

```
/* File: recordspecifytime.ch
   Record time and distances, plot the acquired data */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 2.5;         // speed in 1.5 inches/seconds
double radius = 1.1;       // radius of the wheel in inches
double timeInterval = 0.1; // time interval in 0.1 second
double time1 = 16;          // total travel time
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                 // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed to 'speed' */
robot.setSpeed(speed, radius);

/* begin recording time and distance */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

/* drive the robot for the specified 'time1' */
robot.driveTime(time1);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

/* plot the data */
plot.title("Distance versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (inches)");
plot.data2DCurve(timedata, distances, numDataPoints);
plot.plotting();
```

Program 8.15: Plotting the distance of a Mindstorms versus time for a two-wheel robot with a specified time.

In Program 8.9 we recorded data for a Mindstorms moving forward for 16 seconds using **driveDistance()**. Program 8.15 is very similar to Program 8.9. The difference is that **driveTime()** now we use to drive a Mindstorms forward for 16 seconds. Even though a different movement function was used, **recordDistanceBegin()** and **recordDistanceEnd()** can still be used to collect data on this movement.

When a two-wheel robot moves at the speed of 2.5 inches per second, the relation between the distance (d) and time (t) in Figure 8.15 can be formulated by the following linear equation.

$$d = 2.5t \tag{8.4}$$

Similarly, if we change the statement

```
 robot.setSpeed(speed, radius);
```

in Program 8.15 to

```
 robot.setSpeed(-speed, radius);
```

to drive the Mindstorms in the opposite direction. The plot generated by such a program `recordspecifytimeneg.ch` is shown in Figure 8.16. The relation between the distance (d) and time

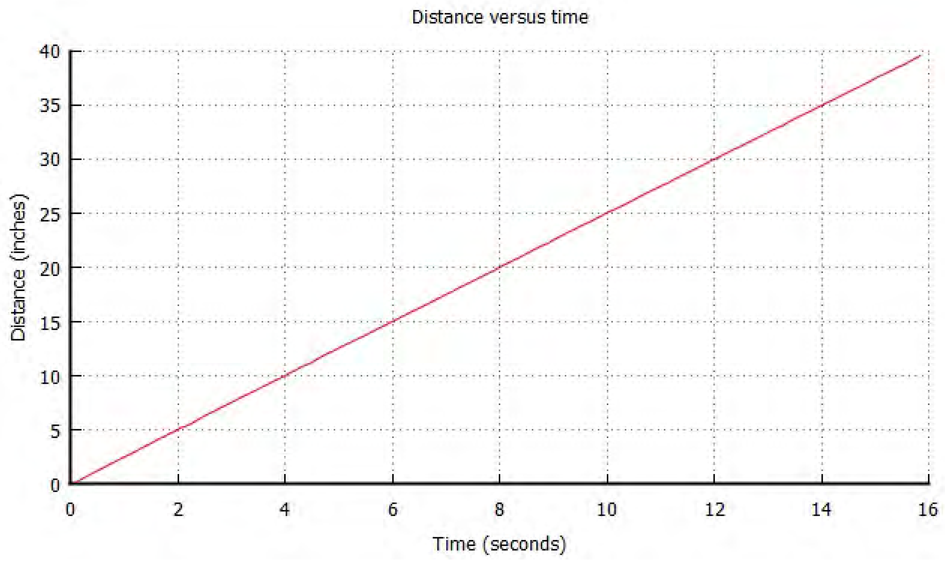


Figure 8.15: The plot for the distance versus time from Program 8.15.

(t) in Figure 8.16 can be formulated by the following equation.

$$d = -2.5t \tag{8.5}$$

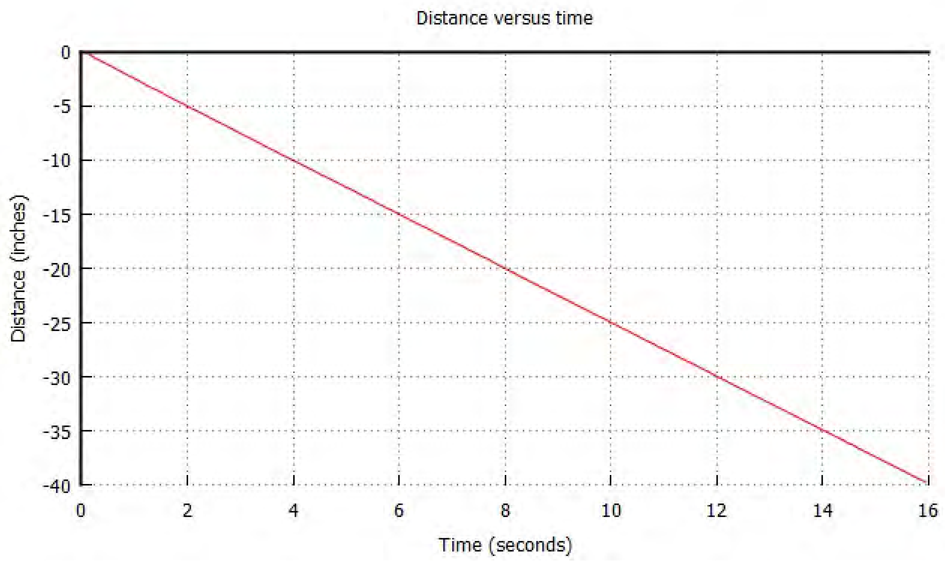


Figure 8.16: The plot for the distance versus time for a Mindstorms moving in the negative direction from the program `recordspecifytimeneg.ch` with a specified time .

E Do Exercise 4 on page 123.

8.4.4 Summary

1. Call the **CMindstorms** member function

```
robot.driveTime(time1);
```

to drive a Mindstorms for a specified time ‘time1‘ rather than a specified distance.

8.4.5 Terminology

robot.driveTime().

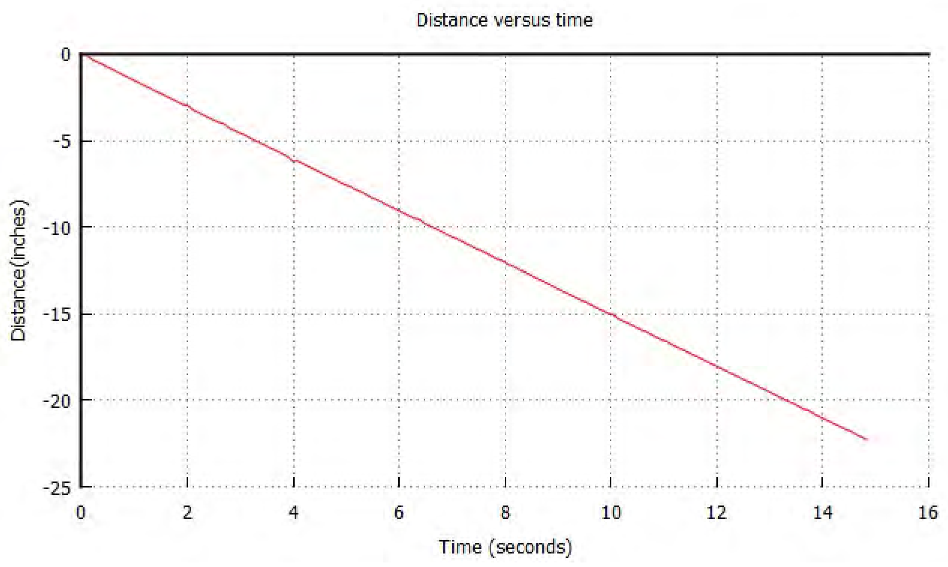
8.4.6 Exercises

1. Watch the video tutorial “R3. Drive for a Certain Time: driverTime()” in http://roboblockly.ucdavis.edu/videos.

2. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Use the program `drivetime_p.ch` in Program 8.13 to make the Mindstorms move 12 inches in 5 seconds (Note that the speed is defined as the distance divided by the time).

3. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `getdistance3.ch` to drive the robot for 4.5 seconds at the speed of 3.2 inches per second. The program shall measure the distance that the robot has driven.

4. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 1.1 inches. Write a program `recordspecifytime2.ch` to drive the Mindstorms backward for 15 seconds at the speed of 1.5 inches per second by the member function **driveTime**(). Record the distance as the Mindstorms travels with a time interval of 0.1 second. Plot the distance versus time as shown in the figure below. What is the equation for the linear relation shown in the figure?



8.5 Use Different Units for Speed, Radius, and Distance

The arguments of member functions **setSpeed**() and **driveDistance**() of the **CMindstorms** class as well as functions **distance2angle**() and **angle2distance**() involve speed, radius, and distance. They are typically used in programs in the following format.

```
    robot.setSpeed(speed, radius);
    robot.driveDistance(distance, radius);
    distance = angle2distance(radius, angle);
    angle = distance2angle(radius, distance);
```

We can use different units for speed, radius, and distance conveniently to control a Mindstorms so long as they are consistent using the same unit for length. Samples of consistent units for speed, radius, and distance are shown in Table 8.3.

Table 8.3: The commonly used units for speed, radius, and distance.

Speed	Radius	Distance	Description
cm/s	cm	cm	centimeters
m/s	m	m	meters
inch/s	inch	inch	inches
foot/s	foot	foot	feet

Problem Statement:
A Mindstorms is configured as a two-wheel robot with wheels attached to ports B and C. The radius of each wheel is 4.445 centimeters. Write a program `recorddistancecm.ch` to drive the Mindstorms for 20 centimeters at the speed of 6.5 centimeters per second. Record the distance as the Mindstorms moves with a time interval of 0.1 second. Plot the distance versus time.

```
/* File: recorddistancecm.ch
   Record time and distances, plot the acquired data, using centimeters */
#include <mindstorms.h>
#include <chplot.h>
CMindstorms robot;
double speed = 6.5;         // speed in 6.5 centimeters/seconds
double radius = 4.445;      // radius of the wheel in centimeters
double distance = 20;       // distance in centimeters
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, distances; // recorded time and distances
CPlot plot;                 // plotting class

/* move to the zero position */
robot.resetToZero();

/* set the robot speed */
robot.setSpeed(speed, radius);

/* begin recording time and distance */
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);

/* drive the specified distance based on the radius of the wheels */
robot.driveDistance(distance, radius);

/* end recording time and distance */
robot.recordDistanceEnd(numDataPoints);

/* plot the data */
plot.title("Distance versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distance (centimeters)");
plot.data2DCurve(timedata, distances, numDataPoints);
plot.plotting();
```

Program 8.16: Plotting the distance of a Mindstorms versus time for a two-wheel robot with the specified distance in centimeters.

Program 8.16 is nearly identical to Program 8.9. The only difference is the values for `speed`, `radius`, and `distance` are all in centimeters instead of inches. Because all these values have consistent length units, they can be used as arguments of the functions **setSpeed()**, **driveDistance()**, **distance2angle**(), and **angle2distance**() without error.

When a two-wheel robot moves at the speed of 6.5 centimeters per second, the relation between the distance (d) and time (t) in Figure 8.17 can be formulated by the following linear equation.

$$d = 6.5t \qquad (8.6)$$

E Do Exercises 1, 2, and 3 on page 126.

8.5.1 Summary

1. The values for speed, radius, and distance can be expressed in centimeters, meters, inches, or feet as long as the units used for length are consistent.

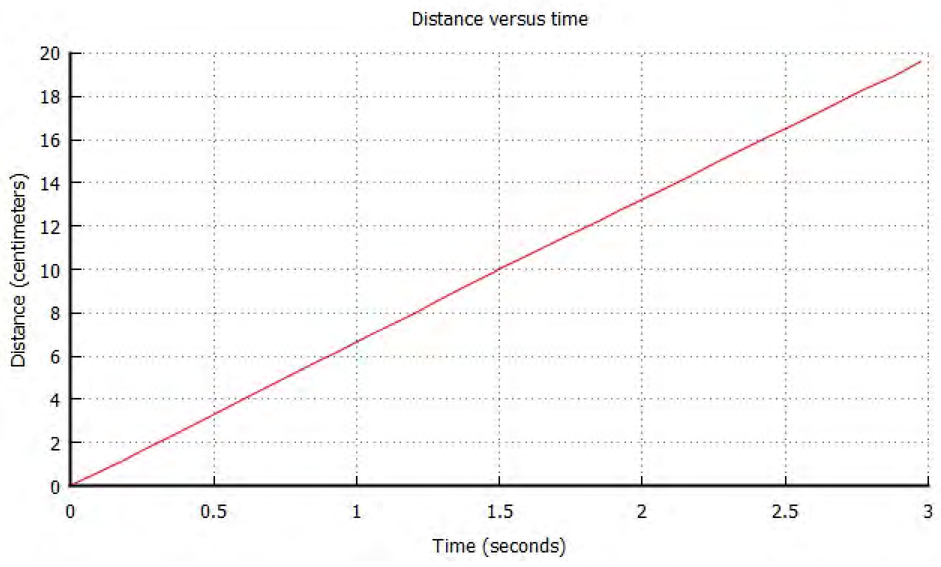


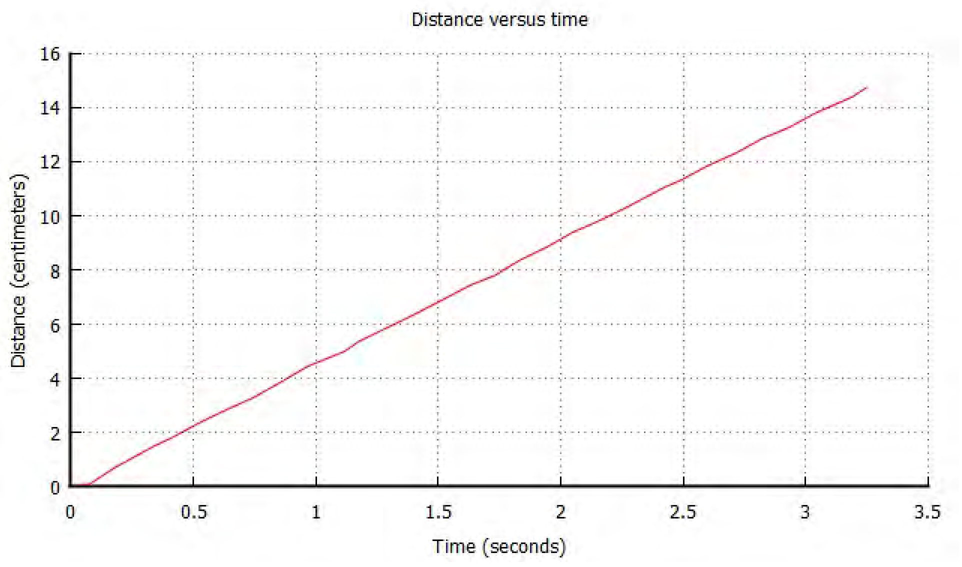
Figure 8.17: The plot for the distance versus time from Program 8.16.

8.5.2 Terminology

different length units, consistent length units.

8.5.3 Exercises

1. A Mindstorms is configured as a two-wheel robot with wheels attached to joints 2 and 3. The radius of each wheel is 4.445 centimeters. Write a program `recorddistancecm2.ch` to drive the Mindstorms 15 centimeters at the speed of 4.5 centimeters per second. Record the distance as the Mindstorms travels with a time interval of 0.1 second. Plot the distance versus time as shown in the figure below. What is the equation for the linear relation shown in the figure?



2. Modify the program `drivedistance_p.ch` in Program 8.2 as the program `drivedistancecm_p.ch` so that the user can enter the radius of wheels in centimeters, speed in centimeters per second, and distance in centimeters to drive a Mindstorms configured as a two-wheel robot. Test your program by entering the radius of the wheel of your Mindstorms, speed of 5 centimeters per second, and distance of 12 centimeters.

3. Modify the program `drivetime_p.ch` in Program 8.13 as the program `drivetimecm_p.ch` so that the user can enter the radius of wheels, speed in centimeters per second, and time in seconds to drive a Mindstorms configured as a two-wheel robot. Test your program by entering the radius of the wheel of your Mindstorms, speed of 5 centimeters per second, and time of 6 seconds.