Learning Robot Programming with Linkbot for the Absolute Beginner

7th Edition

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

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 Linkbot. 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 Linkbot 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 Lego Mindstorms 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 the Linkbot 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 Linkbot.

Chapter 2 uses a user friendly graphical user interface called Linkbot Labs to control the Linkbot.

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 Linkbots to perform identical tasks such as dancing.

Chapter 8 describes how to control a Linkbot 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 Linkbot with different motion characteristics.

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

Chapter 12 describes how to write programs to process the sensory information for Linkbots.

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

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 Linkbots.

Chapter 16 describes how to control multiple connected Linkbots.

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

Appendix C lists the color names and corresponding RGB values available for the Linkbot.

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

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

Appendix F contains quick references to member functions of the Linkbot classes.

Appendix G 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  indicates the location to pause for students to solve problems in the exercise section.

The symbol  indicates the Linkbot-L is used in the section or exercise.

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

The latest version of this documentation is available from **(http://c-stem.ucdavis.edu)**.

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

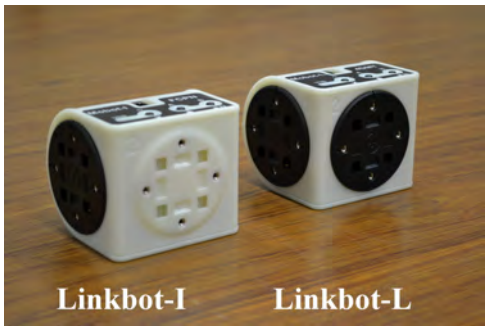
Introduction

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 **Mobot**, as shown in Figure 1.1, is designed as a building block. However, a single Mobot module is a fully functional four-degrees-of-freedom modular robot. It has four motors inside that allow it to perform a multitude of novel robot locomotion, including inch-worming, rolling, arched rolling, turning, tumbling, and standing up. Multiple Mobot modules can be interconnected into various geometries for different applications, such as a space explorer, snake, gorilla, dog, humanoid, etc.

Figure 1.1: A Mobot.

The ***Linkbot*** is a new version of low-cost modular robot. It can more conveniently interact with other devices and sensors. In comparison with the Mobot, the Linkbot has only two, instead of four, degrees-of-freedom. Namely, there are only two motors inside a Linkbot. There are two versions of Linkbot, called Linkbot-I and Linkbot-L. The shapes for both Linkbot-I and Linkbot-L are the same as shown in Figure 1.2. There are three locations for joints. For Linkbot-I, two joints are in the opposite of the housing in the I-shape. For Linkbot-L, two joints are adjacent in the L-shape. Other than the difference in the location of joints, the interface and programming for both Linkbot-I and Linkbot-L are the same. The term Linkbot in this manuscript refers to both Linkbot-I and Linkbot-L. Like Mobots,



(a) Left view of Linkbot-I and Linkbot-L

(b) Right view of Linkbot-I and Linkbot-L

Figure 1.2: Linkbot-I and Linkbot-L.

multiple Linkbot modules can be interconnected into various geometries for different applications, such as a space explorer, snake, four-wheel drive, omnidrive, etc. as shown in Figure 1.3. Linkbots can also be connected with Mobots.

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 Linkbots and Mobots.

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

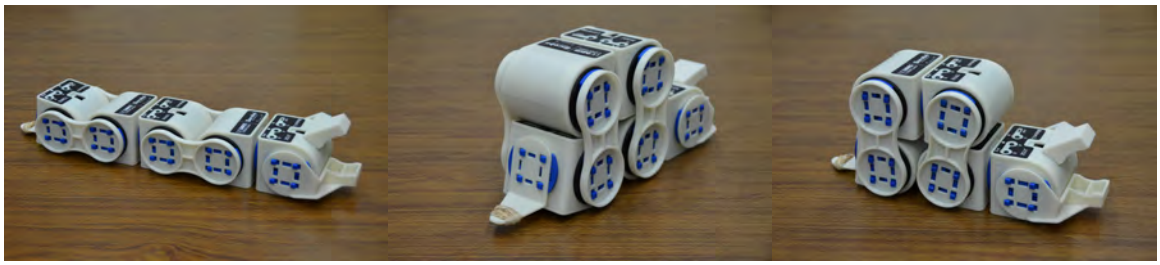
1.1.1 Summary

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

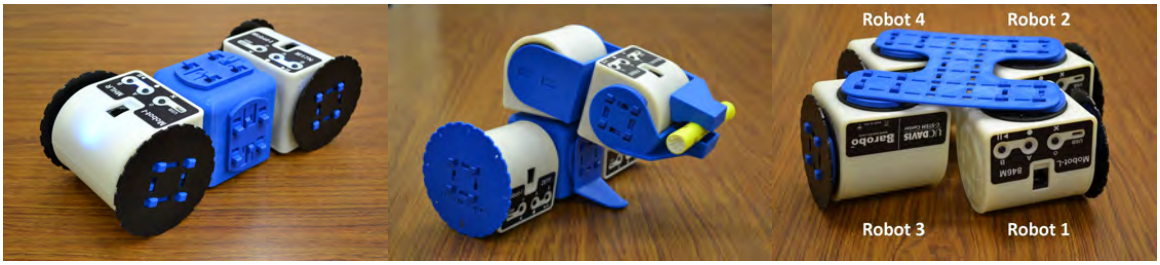
1. A robot is a re-programmable machine that moves, senses, and reacts to its environment.
   (a) Robots can only perform tasks they have been programmed to accomplish
   (b) Robotics is a branch of technology that deals with design, construction, operation and application of robots and the related computing systems for the control, sensing, and information processing for robots.
2. A Mobot is a modular robot with 4 degrees of freedom.
3. A Linkbot is the new version of modular robot that interfaces with other devices and sensors. It has 2 degrees of freedom.
   (a) Linkbot-I: two joints are in the opposite of the housing in the I-shape.
   (b) Linkbot-L: two joints are adjacent in the L-shape.
   (c) Programming the Linkbot-I and Linkbot-L is the same.

1.1.2 Terminology

Linkbot, Linkbot-I, Linkbot-L, Mobot, degrees of freedom, robot, robotics.



(a) Snake position 1 (b) Snake position 2 (c) Snake position 3



(d) four-wheel drive (e) Explorer (f) Omnidrive

Figure 1.3: Robotic systems built with multiple Linkbots.

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 Linkbot-I, Linkbot-L, and Mobot?
5. List at least 3 combinations that the Linkbots can be put into for various applications.

1.2 C-STEM Studio

Figure 1.4: 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 Robot Controller for Lego Mindstorms NXT/EV3, RoboSim, RoboBlockly, Ch Arduino, and Ch Raspberry Pi, as shown in Figure 1.4.

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 Linkbots. 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 1.4, in Windows, click on the icon labeled "C-STEM Studio" on your desktop, as shown in Figure 1.5. On Mac OS X systems, C-STEM Studio application is located inside the "Applications" folder in Finder.

Figure 1.5: The icon for C-STEM Studio.

E Do Exercise 1 on page 5.

1.2.1 Summary

C-STEM Studio

1.2.2 Terminology

C-STEM Studio.

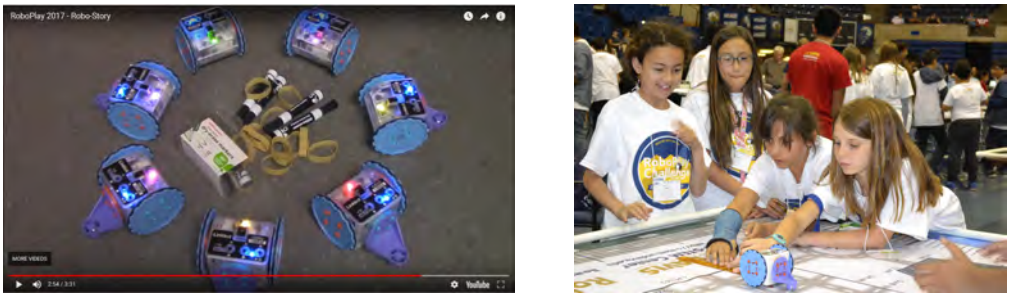
1.2.3 Exercises

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 1.6: 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.

E Do Exercises 2 and 3 on page 6.

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 Linkbot

A wide range of technologies have been integrated into the Linkbot. A CAD (computer-aided design) model along with its features for Linkbot is shown in Figure 1.7.

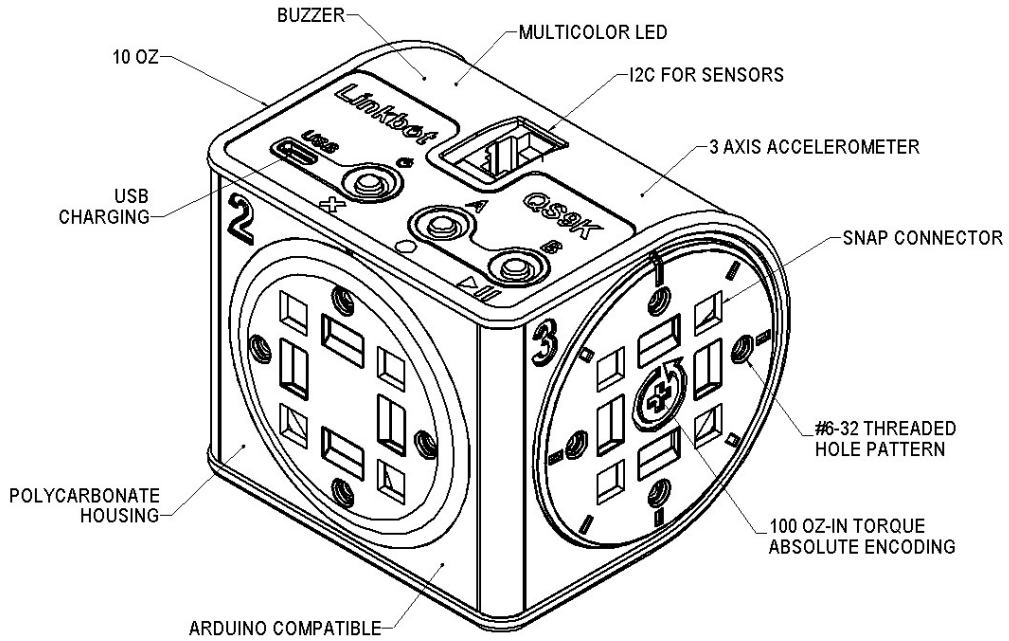


Figure 1.7: A CAD model of Linkbot.

The CPU (central processing unit) inside the Linkbot is the ATmega128RFA1 microcontroller by Atmel, running at 16 MHz. It integrates a ZigBee radio transceiver for wireless communication.

The Linkbot is powered by an internal, rechargeable lithium-ion battery that can drive motors for over 3 hours with typical use. Charging is done through the USB port with a cable and takes about 4 hours when connected to a computer. But, if it is plugged into a cell phone charger, the charging time drops to less than 2 hours. The user can use the Linkbot while it is plugged in for continual operation.

A multi-color LED can be used to personalize a Linkbot by selecting from a wide spectrum of colors. The Linkbot has a buzzer built-in to play a tune. The Linkbot also has a 3-axis accelerometer to detect free-falls, bumps, and tilted angles. Each Linkbot has three buttons for user interface.

The Linkbot weighs 10 oz. It has 100 oz-in (7.2 Kg-cm) of torque for each joint. Each hub has absolute encoding for precise control and measurement of speeds and angles accurate to ± 0.5 degree. By default, the maximum joint speed is 200 degrees per second for constant velocity control.

The Linkbot is expandable with SnapConnectors, which allow modules to be snapped together without needing tools. This allows the user to quickly try out new robot creations. The Linkbot also has standard #6-32 threaded holes available in its faceplate for custom-made accessories. The robot is made out of durable poly-carbonate plastic. A wide variety of accessories in CAD files are available for the Linkbot. The user can make accessories on their own using a 3D printer. Users can also customize these CAD files to their unique application and print using a 3D printer, laser cut, or even CNC (computer numerical control) machine depending on the part design.

E Do Exercises 1 and 2 on page 8.

1.4.1 Summary

1. Linkbot is powered by a lithium-ion battery that lasts approximately 3 hours. It is charged through USB with a cable and takes approximately 4 hours.
2. The Linkbot has multi-color LED for personalization.

3. It has a 3-axis accelerometer to detect free-falls, bumps, and tilted angles.
4. The maximum joint speed is 200 degrees per second for velocity control.
5. SnapConnectors allow modules to attach to the Linkbot no tools needed. Customizable CAD files are available for customers to make accessories using a 3D printer or laser cutter.

1.4.2 Terminology

3-axis accelerometer, 3D printer CAD, CAD files, CNC machine, CPU, Laser cut, SnapConnectors, Torque, ZigBee.

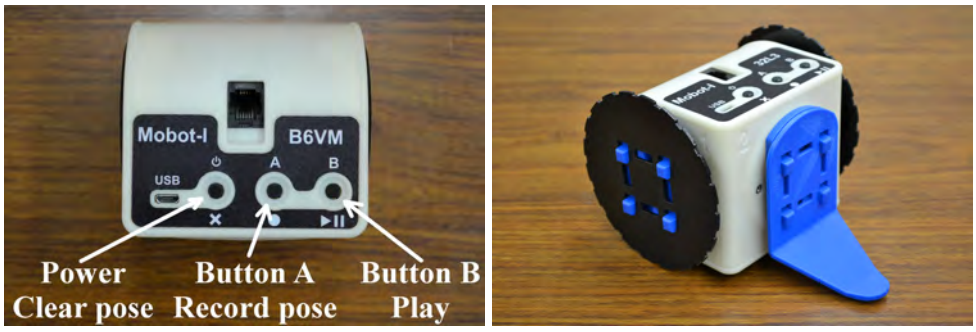
1.4.3 Exercises

1. What does the Linkbot use for wireless communication?
2. How long does the battery last in a Linkbot? How long does it take to charge the Linkbot from a computer? How long using a cell phone charger?

1.5 Run the On-Board Demo Program on the Linkbot

The Linkbot comes with a demo program in the on-board embedded computer allowing it to work out-of-the box. The program is designed to show the motion and make sure everything is working properly. The demo program will move the Linkbot continuously until the Linkbot is shut off or the batteries die.

To run the demo program, turn on the Linkbot by pressing the power button on the Linkbot as shown in Figure 1.8. Then simply hold the button 'A' for three seconds. After three seconds have passed, the LED will be changed to different colors and the demo motion will begin. You may want to run the demo for a Linkbot-I with two wheels attached, as shown in Figure 1.8b. You can turn off the demo program by pressing and holding the 'A' button for three seconds while the robot is moving.



(a) Buttons on the Linkbot (b) Running the on-board demo program

Figure 1.8: Running the on-board demo program on the Linkbot.

E Do Exercises 1 and 2 on page 9.

1.5.1 Summary

1. The Linkbot comes with a demo program in the on-board embedded computer.
   (a) The Demo Program shows the motion to make sure everything works properly.
   (b) The Demo Program will continuously move Linkbot until it is shut off or batteries die.
2. To run demo program: turn on the Linkbot by pressing the power button. Then, hold the A button for three seconds. After three seconds, blue LED blinks three times and demo motion begins.

1.5.2 Terminology

On-board embedded computer, demo program.

1.5.3 Exercises

1. Run the demo program inside a Linkbot by pressing the power button on the Linkbot and hold the button 'A' for three seconds. You can stop the demo program by holding the button 'A' for three seconds.
2. What is the purpose of the Linkbot demo program?

1.6 Play with Multiple Linkbots

A Linkbot can be used to control one or multiple other Linkbots. Multiple Linkbots can also be controlled using PoseTeaching. In such applications, the multiple Linkbot modules must be paired together using a process called BumpConnect. The process can also be used to add additional follower robots to an existing group. All BumpConnected groups of Linkbots contain a single leader robot which can control one or more follower robots.

BumpConnect two unpaired Linkbots:

1. Press and hold the 'B' button on the Linkbot you want to be the leader.
2. Press and hold the 'B' button on a second Linkbot.
3. Gently bump the two modules together.
4. Release the 'B' buttons.
5. After a second or so, the LED colors on both robots should change. The leader robot will blink, cycling between Blue and a randomly chosen group color. The follower should display the randomly chosen group color.
6. If nothing happens after 5 seconds, try again starting from step 1.

BumpConnect a paired Linkbot with an unpaired Linkbot:

1. Press and hold the 'B' button on the leader robot and the unpaired robot.
2. While holding the 'B' buttons, gently bump the two robots together.
3. The unpaired robot should take on the color of the paired robot. If this does not happen after 5 seconds, try the process again starting from step 1.
4. At this point, the robots are in PoseTeaching mode, you may press the 'B' button on the leader robot to switch into TiltDrive mode. The PoseTeaching and TiltDrive modes are described in next sections.

1.6.1 PoseTeaching Mode

After robots have been paired with each other to form a group, they enter the default "PoseTeaching" mode. Initially, there is no recorded pose. In this mode, the buttons have the following functions:

- The Power or 'X' button: If the robots are not playing poses, this button deletes all recorded poses.
- The 'A' button: If the robots are not playing poses, this button records a new pose. You can manual more joints, then press the 'A' button. Each time the 'A' button is pressed, a new pose is recorded. Therefore, a sequence of poses can be recorded.
- The 'B' button: If there are no recorded poses, the 'B' button makes the robot switch into "TiltDrive" mode. If there are recorded poses, the 'B' button begins playing the poses. If the robot is currently playing poses, the 'B' button stops playing poses.

1.6.2 TiltDrive Mode

While the group is in TiltDrive mode, the leader robot will flash its LED colors between green and the group color. In TiltDrive mode, the leader module can be tilted forward, backward, and side-to-side to drive the follower modules. In this mode, the buttons have the following functions:

- The 'B' button: Changes the current mode from "TiltDrive" mode to "CopyCat" mode.

1.6.3 CopyCat Mode

While the group is in CopyCat mode, the leader robot will flash its LED colors between light-blue and the group color. In CopyCat mode, all of the follower modules will move their joints to match the position of the joints on the leader module. If there are intermixed -I and -L Linkbot modules, joint 2 will match the angle on the leader's joint 3, and vice versa. In this mode, the buttons have the following functions:

- The 'B' button: Changes the current mode from "CopyCat" mode to "PoseTeaching" mode.

1.6.4 Unpair the Connected Linkbots

You can either press and hold the 'A' button on the leader module to unpair all modules at once, or press and hold the 'A' button on a follower module to just unpair that one follower.


Do Exercises 1, 2, and 3 on page 11.

1.6.5 Summary

1. Use BumpConnect to pair Linkbots by pushing the B buttons on both Linkbots and bumping them together.
2. You can control paired robots using PoseTeaching by pushing the A button and recording new poses that can then be saved as a ch program.
3. You can control paired robots using TiltDrive by pushing the B button directly after pairing them. The TiltDrive allows the user to control the follower module by tilting the leader module to drive the follower module.
4. CopyCat.mode allows the user to manipulate the leader module and have the follower module match that motion. You can control paired robots using CopyCat mode by pushing the B button while in TiltDrive.

1.6.6 Terminology

BumpConnect, CopyCat, PoseTeaching, and TiltDrive.

1.6.7 Exercises

1. Watch the video tutorial "Bump Connect and Tilt-Drive for Linkbot" in http://c-stem.ucdavis.edu/studio/tutorial/.
2. BumpConnect two Linkbots into a group. Play with two Linkbots with PoseTeaching, TiltDrive, and CopyCat modes.
3. BumpConnect three Linkbots into a group. Play with three Linkbots with PoseTeaching, TiltDrive, and CopyCat modes.

CHAPTER 2

Controlling a Linkbot Using the Robot Control Panel

A Linkbot can be conveniently controlled without writing a computer program. This chapter presents detailed steps to control a Linkbot through a user friendly graphical user interface called Linkbot Labs.

2.1 The Zero Positions of the Linkbot

2.1.1 The Schematic Diagram of the Linkbot

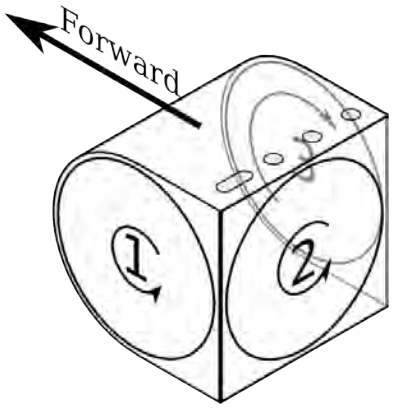


Figure 2.1: A schematic diagram of a Linkbot module.

Figure 2.1 shows a schematic diagram of the Linkbot displaying the locations and positive directions of

three joints of a Linkbot. For Linkbot-I, only joints 1 and 3 are fully rotational. The joint 2 for Linkbot-I cannot move. For Linkbot-L, only joints 1 and 2 are fully rotational. The joint 3 for Linkbot-L cannot move.

The direction of motion for each joint follows the right hand rule shown in Figure 2.2. Figure 2.1 also shows the forward direction for a Linkbot-I.

Figure 2.2: The right hand rule for the direction of motion for a joint of Linkbot.

2.1.2 The Zero Position for a Joint of the Linkbot

There are tiny notches on the top of the housing and disk for each joint. When these two notches for a joint are aligned as shown in Figure 2.3, the joint is in its zero position. For an illustrative purpose, these two notches for a joint are displayed with different colors as zero position marks in Figure 2.3.

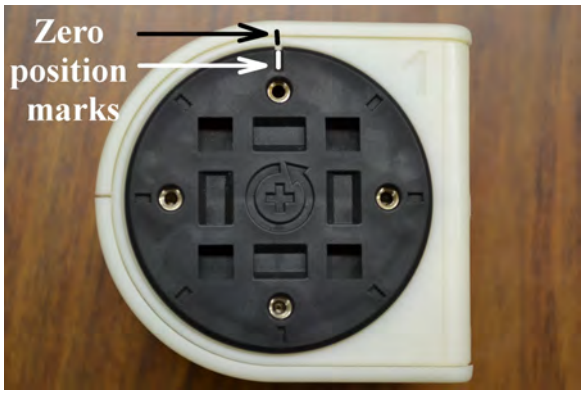


Figure 2.3: The zero position of the Linkbot.

2.1.3 Testing the Zero Positions and Relaxing Motors of a Linkbot

Buttons 'A' and 'B' pressed together may be used to toggle the Linkbot between holding its zero position or relaxing all of its joints.

If the robot is currently relaxed, pressing buttons 'A' and 'B' at the same time will move both joints to zero position and hold its position. If the 'A' and 'B' buttons are pressed again, the robot will relax its motors, allowing the user to reposition the robot.

If the robot is currently holding a position, pressing buttons 'A' and 'B' at the same time will relax the joints of the robot. If the 'A' and 'B' buttons are pressed again, the robot will move to zero position.

This feature is typically used to check the robot's zero position to make sure it is properly calibrated. The button may also be used to relax a robot that is currently holding any of its joints at a position.

The feature can also be used to reset the robot to the zero position before a program controlling it is run. We will learn how to write a robot program in the next chapter.

2.1.4 Recalibrating the Zero Positions of the Linkbot

When the Linkbot is used for a long period of time, its zero positions for joints may be off. A user may recalibrate the robot's zero positions if necessary. To recalibrate the robot, perform the following steps:

- Position each joint of the Linkbot into its zero position. If the robot is currently actuating any of its joints, or if any joints feel "stiff", you may power the robot off and on again, or you may press both the 'A' and 'B' buttons to relax the joints.
- While the robot is powered on, press and hold both the 'A' and 'B' buttons at the same until the motors begin to move. The motors will move for 5 seconds as the robot recalibrates itself.
- Test the zero position as described in section 2.1.3.

E Do Exercises 1 and 2 on page 14.

2.1.5 Summary

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

1. Reset the two joint angles of a Linkbot to the zero position by pressing the A and B buttons simultaneously.
2. The direction of a joint motion follows the right hand rule.
3. For Linkbot-I, only joints 1 and 3 are fully rotational joint 2 cannot move.
4. For Linkbot-L, only joints 1 and 2 are fully rotational joint 3 cannot move.
5. Recalibrate the Linkbot if used for a long period of time because the joints zero positions may be off. Put the joints in the zero position, and then push the A and B buttons simultaneously for three seconds until the LED flashes quickly three times.

2.1.6 Terminology

Right hand rule, zero position.

2.1.7 Exercises

1. Recalibrate the zero positions of a Linkbot and test the zero postions.
2. Explain how the right hand rule works and what it tells you about the direction of motion for a joint of the Linkbot.

2.2 Connect Linkbots from a Computer

A Linkbot-I can be configured as a two-wheel robot as shown in Figure 2.4. In this configuration, the faceplates for joints 1 and 3 are attached with two wheels. Joints 2 and 3 are locked by a caster to serve as passive contacting point while the two wheels rotating. We will control a Linkbot-I as a two-wheel vehicle to get started for robot programming.

Figure 2.4: A two-wheel robot.

Linkbot modules should be configured the first time they are used with a new computer. The process informs the computer to which Linkbots are allowed to connect. The configuration is performed through Linkbot Labs program. This section contains step-by-step instructions on how to configure Linkbots.

2.2.1 The Linkbot Device Driver

Linkbot Labs in a computer can control one or multiple Linkbots. One Linkbot needs to be connected to the computer through the USB cable. For controlling multiple Linkbots, the Linkbot connected to the computer through the USB cable will serve as a dongle for Linkbot Labs to connect and control other Linkbots through the ZigBee wireless communication, as shown in Figure 2.5. The other Linkbots in the network can be located as far as 100 meters.

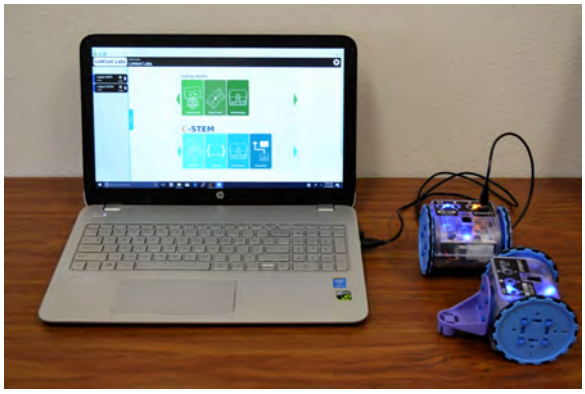


Figure 2.5: The configuration for Linkbot Labs to control multiple Linkbots, with one Linkbot connected to a computer through a USB cable.

For Linkbot Labs to control a Linkbot, a device driver for the Linkbot needs to be installed. A ***device driver*** is a computer program that operates or controls a particular type of device that is attached to a computer. Linkbot Labs invokes a Linkbot device driver which communicates with the Linkbot through the USB cable.

2.2.2 Adding Linkbot IDs in Linkbot Labs

First, start Linkbot Labs application program. You can double click "Linkbot Labs" on the C-STEM Studio shown in Figure 1.4 to launch Linkbot Labs. The Robot Manager as shown in Figure 2.6 should pop up.

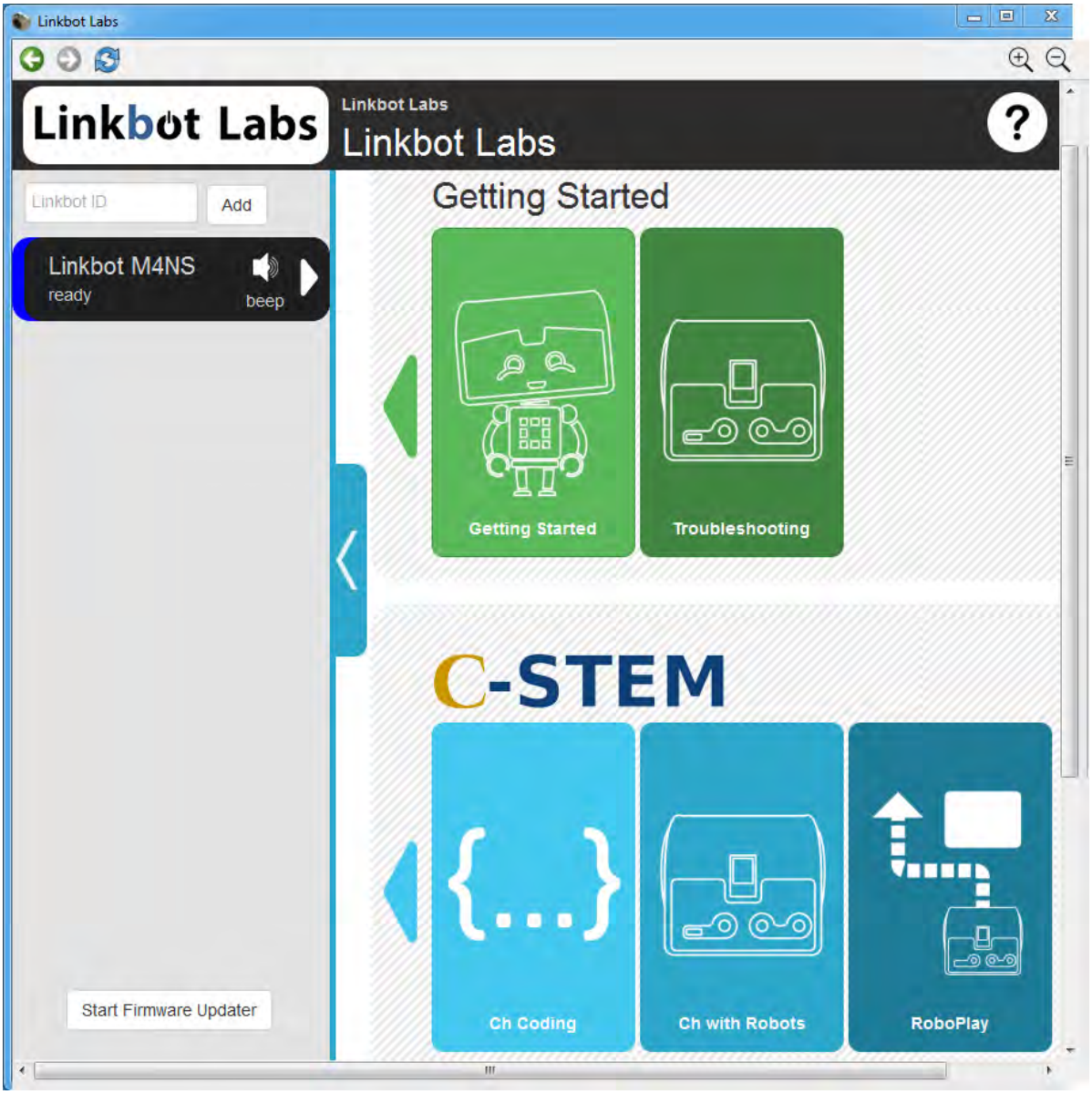


Figure 2.6: The opening screen with the Robot Manager for Linkbot Labs.

To connect a computer to a Linkbot, first, add your Linkbot ID into the Robot Manager by typing in the ID in the box marked "<Linkbot ID>" and clicking on the "Add" button. The Linkbot should appear in the Robot Manager as shown in Figure 2.7.

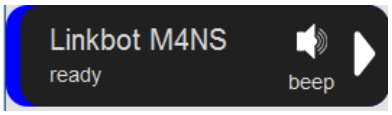


Figure 2.7: The Linkbot "M4NS" in the Robot Manager.

Only one Linkbot needs to be connected to the computer using the USB cable. Additional Linkbots can be connected to the computer through the ZigBee wireless communication, as shown in Figure 2.5. Each additional Linkbot added occupies its own row in the dialog. You can move a Linkbot up and down on the list by dragging its label, such as "Linkbot M4NS", on the Robot Manager.

Once new IDs are added, Linkbot Labs will remember the IDs in the future.

Trouble Shooting the Connection of the Linkbot from a Computer

Should the connection from a computer to the Linkbot fail, you may check the following options to fix the connection problem and try again.

1. The Linkbot ID has been entered correctly.
2. The Linkbot is turned on.
3. Turn the Linkbot off and on again.
4. Restart Linkbot Labs.
5. Plug the USB cable firmly.
6. Try a different USB cable.

Once the connection succeeds, You can click the "beep" button as shown in Figure 2.7 to hear a beep from the Linkbot. Please note that there is no limit on the number of Linkbot that can be connected to Linkbot Labs.

E Do Exercises 1 and 1 on page 18.

2.2.3 Summary

1. Connect to a Linkbot from a computer using Linkbot Labs.
2. Linkbot Labs can control one or multiple Linkbots.
3. To add your Linkbot, type the Linkbot ID into the Robot Manager box marked <Linkbot ID> and click "Add".

2.2.4 Terminology

Linkbot Labs, device driver, Zigbee communication.

2.2.5 Exercises

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

   Connect the computer to your Linkbot by typing in the address and adding the robot and then clicking connect. Click "Beep" to make your Linkbot beep.

2.3 Control a Linkbot Using the Robot Control Panel

2.3.1 The Robot Control Panel

Figure 2.8: The Robot Control Panel of Linkbot Labs.

Once a robot is connected to Linkbot Labs, you can open the Robot Control Panel by pressing the white expansion arrow next to the "beep" button as shown in Figure 2.7. The Robot Control Panel shown in Figure 2.8 displays information about the Linkbot's joint positions, and can also control the speeds and positions of the Linkbot's joints. You can click the text "trash" on the Robot Manager to remove the Linkbot from Linkbot Labs.

2.3.2 Individual Joint and Speed Control

The first section, located at the top-middle section in Figure 2.9, is for "individual joint and speed control.". The buttons in this section command the Linkbot to move individual joints. When the up or down arrows are clicked, the Linkbot begins to move the corresponding joint in either the positive, or negative direction. The joint will continue to move until the stop button, located between the up and down arrows, is clicked.

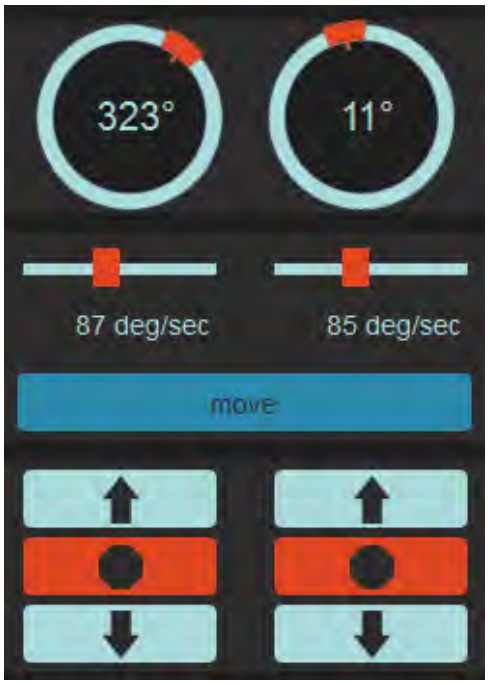


Figure 2.9: The individual joint and speed control.

The positions of joints are displayed inside two cricles. You can drag the red box on the circle to move a joint of the Linkbot.

The speeds for two joints can be set by two vertical bars. It displays and controls the current joint speeds of the Linkbot. The joint speeds are in units of degrees per second. To set a specific desired joint speed for a particular joint, you can move the slider to change the joint speed.

If the joint encounters any obstacle that prevents it from moving, the joint will automatically disengage power to the joint. This may happen, for example, if it collides with the other object.

E Do Exercise 1 on page 22.

2.3.3 Rolling Control for Linkbot-I

The Rolling Control section contains buttons for controlling a Linkbot-I as a two-wheel robot as shown in Figure 2.10. The up and down buttons cause the Linkbot to roll forward or backward, as shown in Figure 2.10. The left and right buttons cause the Linkbot to turn to the left, or to the right. The stop button in the middle causes the Linkbot to stop where it is.

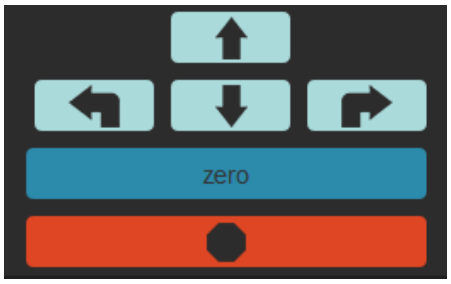


Figure 2.10: The rolling control for a Linkbot-I.

There is a button with the text “zero” on the Robot Control Panel. When clicked, this button will command the connected Linkbot to rotate each joint to the zero angle position.

You can also move all joints of a Linkbot into zero positions by pressing both A and B buttons of the Linkbot.

E Do Exercises 2, 3, and 4 on page 22.

2.3.4 Buzzer Control

A buzzer control bar as shown in Figure 2.11. You can set the buzzer at various frequency.

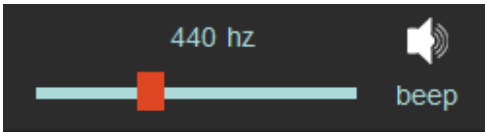


Figure 2.11: The buzzer control on the Robot Control Panel.

2.3.5 Monitoring Accelerometer Data

The acceleromter data for a Linkbot are displayed on the Robot Control Panel as shown in Figure 2.12. When you shake the Linkbot connected to the Linkbot Labs, the accelerometer sliders will move accordingly. It displays the X, Y, and Z components as well as the magnitude of the accelerometer data.

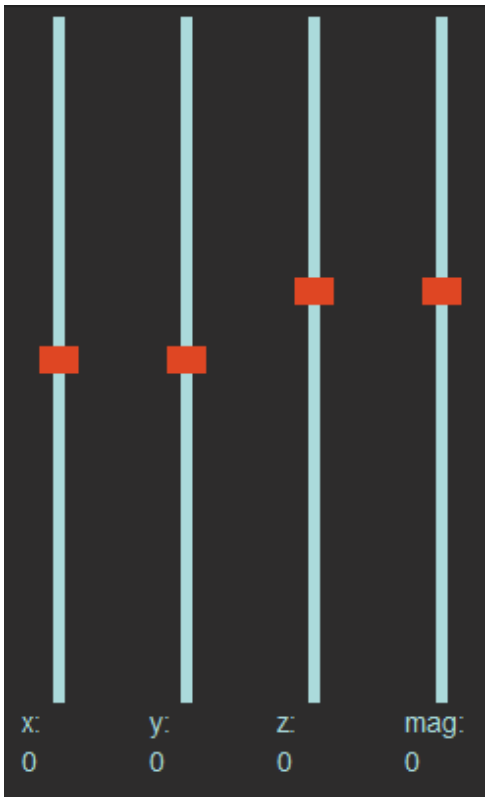


Figure 2.12: Monitoring the accelerometer data on the Robot Control Panel.

E Do Exercise 5 on page 23.

2.3.6 Control Multiple Linkbots

A computer can connect to multiple Linkbots as described in section 2.2. The Robot Control Panel can be used to control multiple Linkbots, one at a time. If a computer is connected to multiple Linkbots as shown in Figure 2.13, an individual Linkbot can be selected to be controlled by clicking the Linkbot ID on the Robot Manager.

Figure 2.13: Two Linkbots on the Robot Control Panel.

E Do Exercise 6 on page 23.

2.3.7 Summary

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

1. Control the motion of joints of a Linkbot individually through the "Robot Control Panel".
2. Control the motion of a Linkbot as a two-wheel vehicle.
3. Set the joint speeds of a Linkbot. Joint speeds are specified in degrees per second.
4. The motion for the Linkbot's joints has no limit.
5. The Robot Control Panel can control the frequency of the buzzer on a Linkbot.
6. The Robot Control Panel can display the X, Y, and Z components and the magnitude of the accelerometer data.
7. Connect to multiple Linkbots from a computer through Linkbot Labs and control one at a time through the Robot Control Panel.

2.3.8 Terminology

Joint angle, joint speed, individual joint control, joint speed control, and joint position control.

2.3.9 Exercises

1. Connect your computer to a Linkbot-I through Linkbot Labs. Move joint 1 in the positive direction through the Individual Joint Control. Move joint 3 in the positive direction. Stop the motion of joints 1 and 3. Move joints 1 and 3 in the negative direction. then stop both joints.

2. Click "zero" to reset two joint angles of a Linkbot-I to the zero position on the Robot Control Panel. First, roll the joints 1 and 3 forward through the Rolling Control. Then, turn the robot left. Next, roll the Linkbot forward. Turn the robot right. Roll the Linkbot backward. Finally, stop the Linkbot.

3. Click “zero” to reset two joint angles to the zero position on the Robot Control Panel. First, roll the joints 1 and 3 forward through the Rolling Control. Change the speed of joints 1 and 3 to 30 degrees per second through the Joint Speed Control while the robot is moving.
4. A Linkbot, 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 1 and 3 to the specified position while timing the motion using a stop watch. (b) If the radius of the wheel is 1.75 inches, what is the distance that the robot has moved forward?
5. While your linkbot is connected, rotate and move the robot around and observe ho w the sliders for accelerometer aata move. Can you cause only two sliders to move while the third remains roughly still?
6. Work with your partner to connect a computer to two Linkbots. Control the connected two Linkbots using the Robot Control Panel of Linkbot Labs on the computer.

2.4 The Color Panel

By default, when a Linkbot is turned on, its LED displays the light blue color. Once it is connected, the color will be changed to the royal blue. You can click the color button for a connected Linkbot on the Robot Manager as shown in Figure 2.6 to launch the Select Color Panel to change the color of the LED on the Linkbot. The Select Color Panel in Figure 2.14 displays the LED color, its RGB (Red, Green, Blue) values and HSV (Hue, Saturation, and Value) values. The RGB and HSV are color models. The color of the LED can be changed through the color dial, RGB, or HSV values.

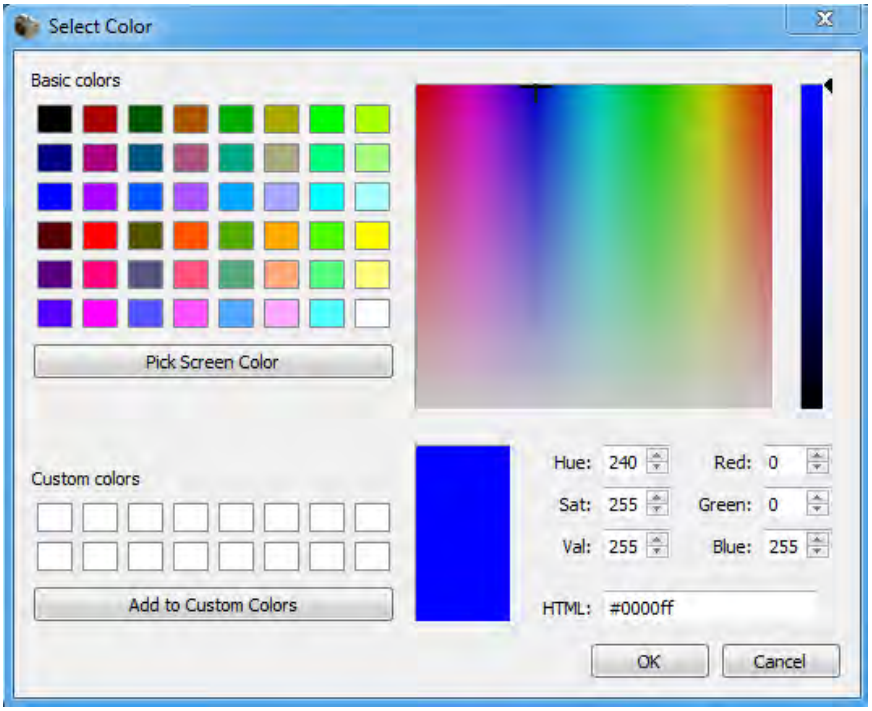


Figure 2.14: The Select Color Panel showing the color of the LED.

 Do Exercise 1 and on page 24.

2.4.1 Summary

1. The default LED color is light blue to show that the robot is turned on.
2. The LED color turns royal blue when the robot is connected.
3. Click on the “color” tab on the Robot Manager. The color of the LED can be changed through the color dial, RGB, or HSV values.

2.4.2 Terminology

LED color, Select Color Panel.

2.4.3 Exercises

1. Click on the “color” tab on the Robot Manager and click on the LED Color panel to choose the color you like.

CHAPTER 3

Getting Started With Programming Linkbots

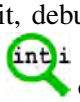
3.1 Get Started with Ch for Computer Programming

In Chapter 2, we learned how to control a Linkbot using the Robot Control Panel. However, in order for a Linkbot to solve complicated problems, we need to write a computer program to control the Linkbot. 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 Linkbot 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 Linkbot.

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 1.4. 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\LearnLinkbot` in Windows, `/opt/C-STEM/LearnLinkbot` in Mac OS X, and `/usr/local/C-STEM/LearnLinkbot` in Linux.

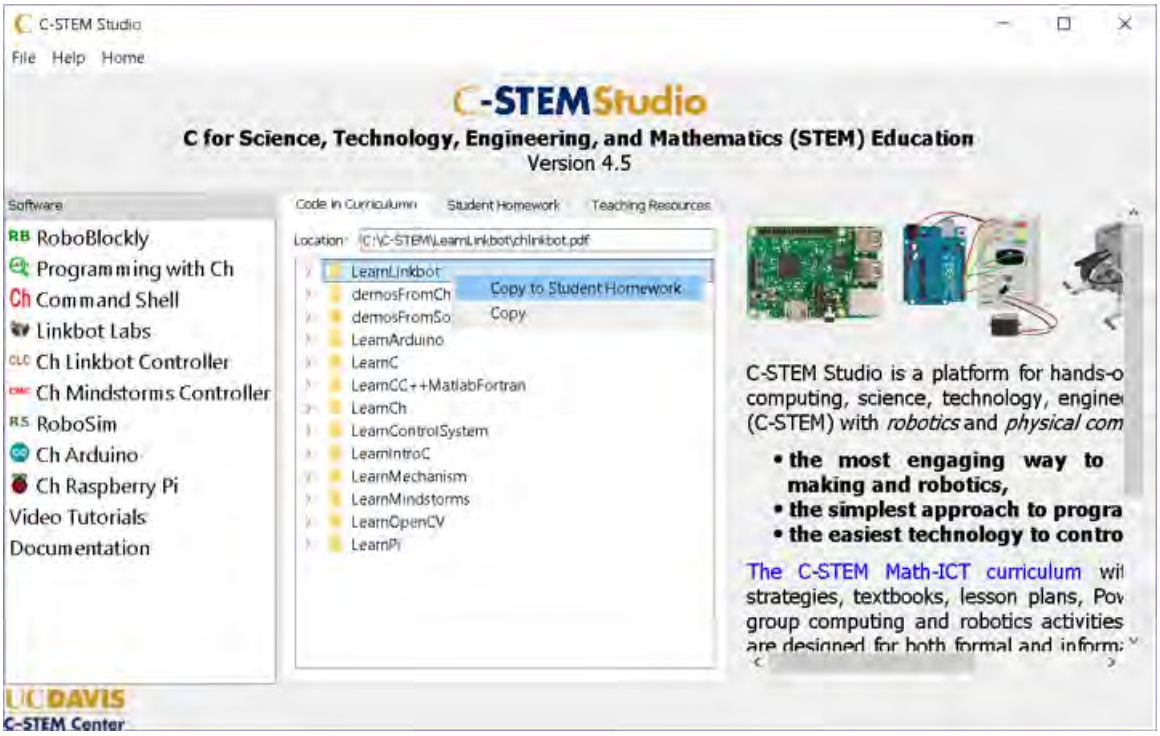


Figure 3.1: Copy the LearnLinkbot folder from "Code in Curriulum" in the C-STEM Studio.

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 `LearnLinkbot` 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 "LearnLinkbot" 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.

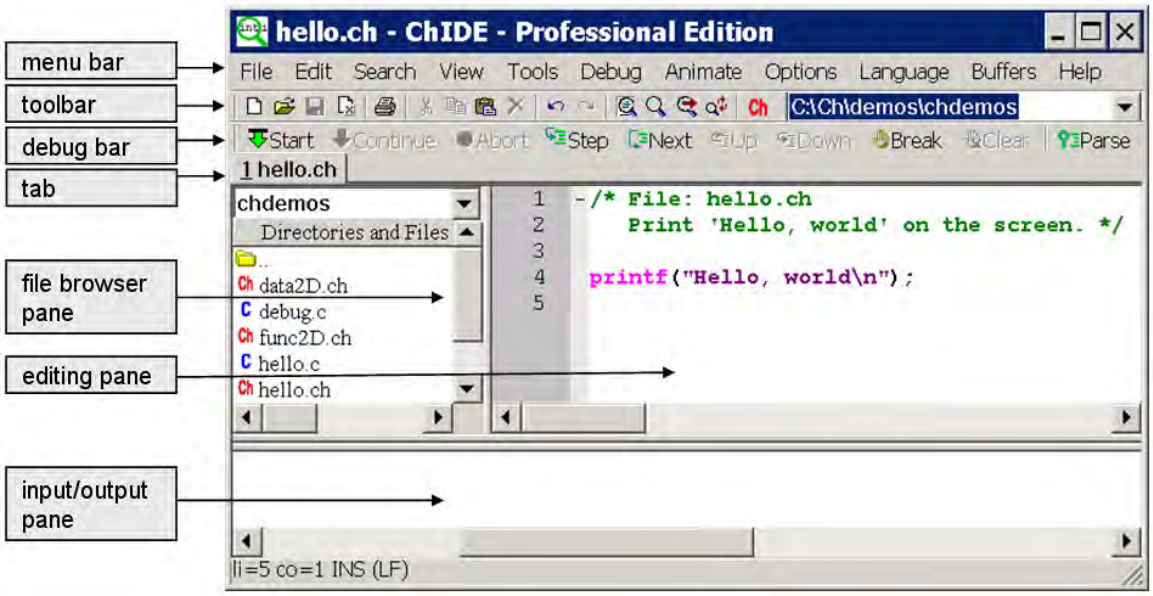


Figure 3.2: A layout and related terminologies in ChIDE.

```
/* 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

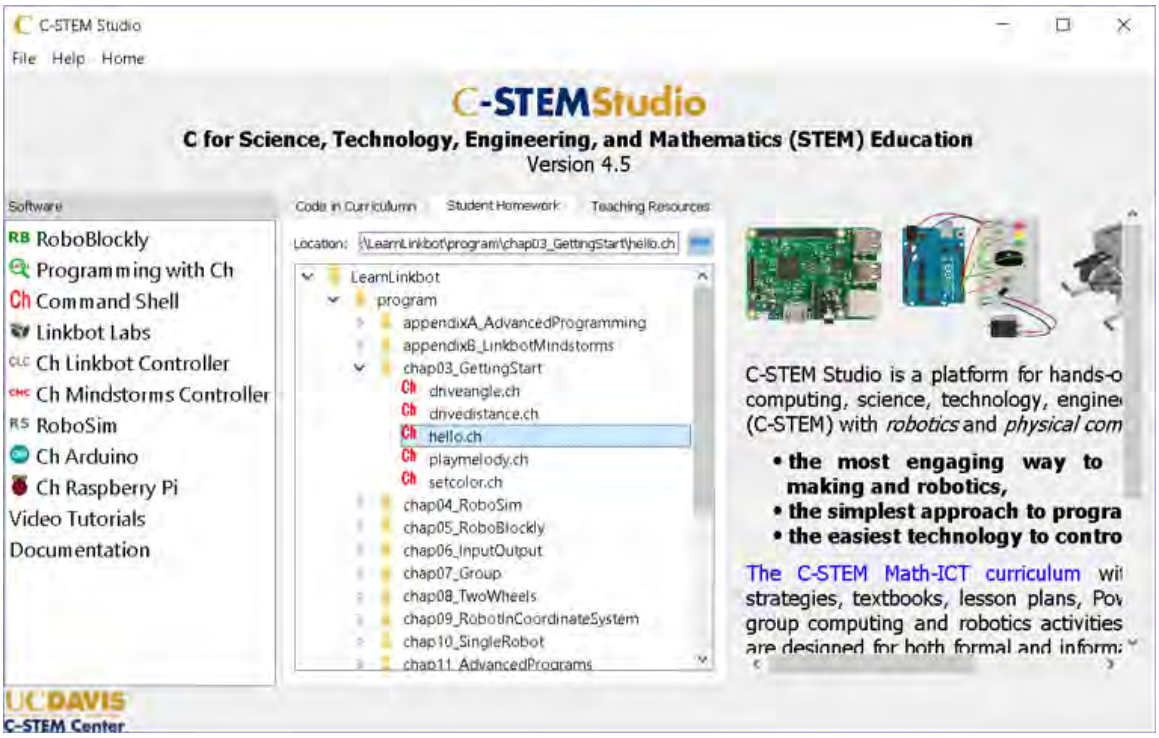


Figure 3.3: Launch the program `hello.ch` in C-STEM Studio.

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.

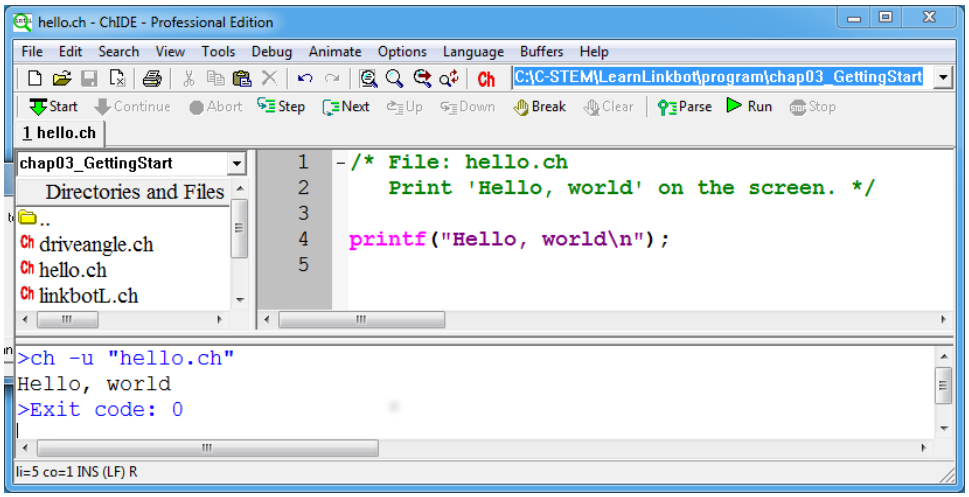


Figure 3.4: Running the program inside the editing pane in ChIDE and its output.

.

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 33.

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.

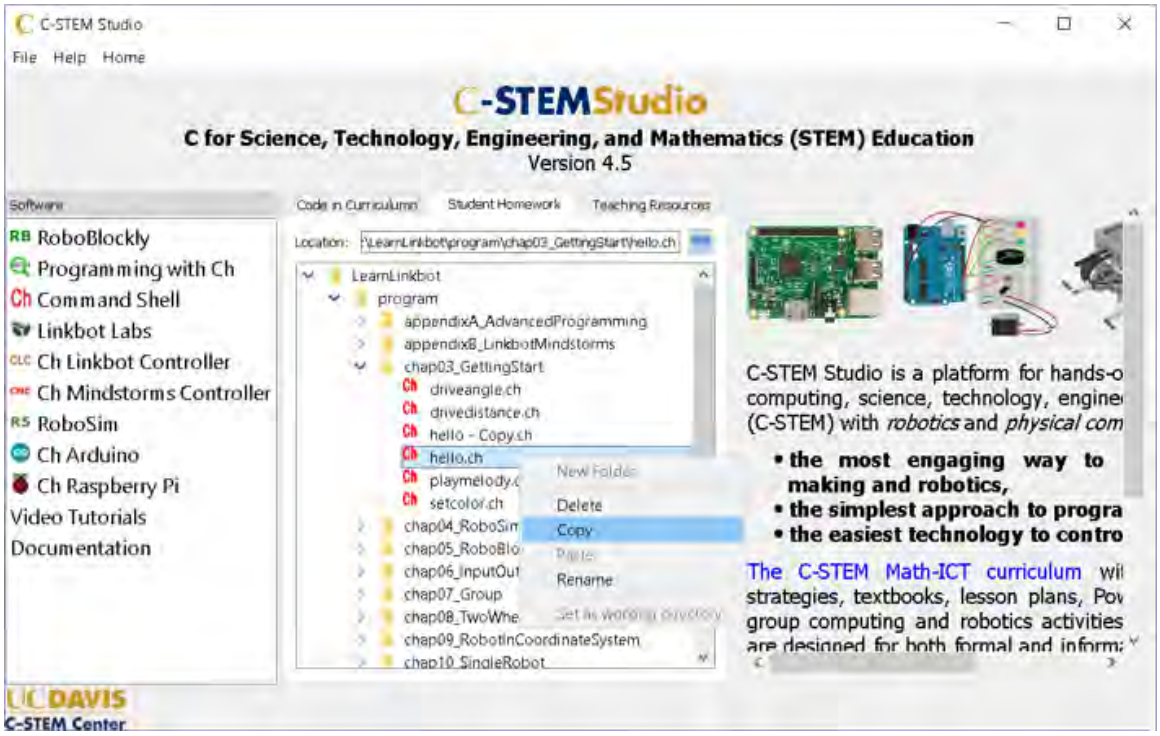


Figure 3.5: Copy the program `hello.ch`, paste, and rename to create a new program `hello2.ch`.

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");
```

Do Exercises 5 and 6 on page 33.

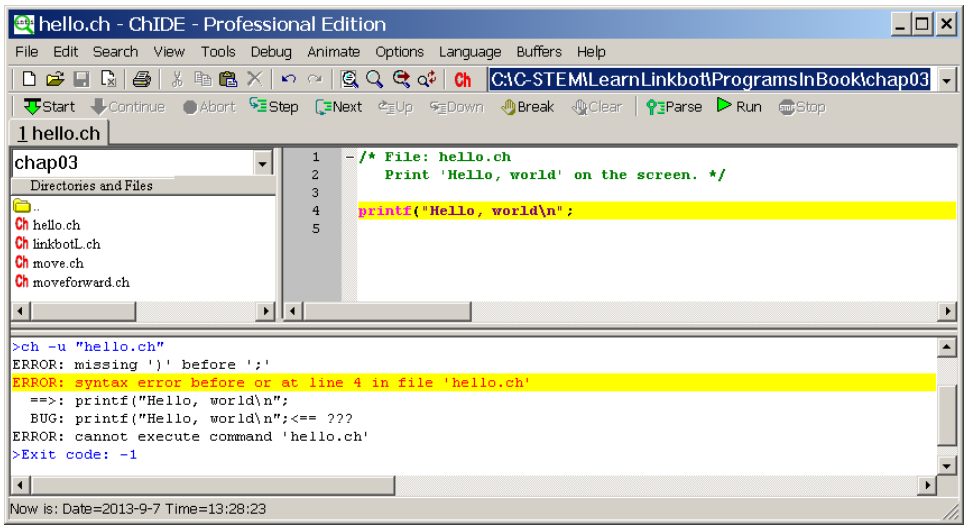


Figure 3.6: The error line in output from executing program hello.c.

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 34.

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**.

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

A Ch program can be developed to control the Linkbot. The working principle of a program is the same as using Linkbot Labs. Program 3.2 contains the code typically used for controlling a Linkbot-I. We will explain the functionality of each statement in this robot program.

```
/* File: driveangle.ch
   Drive forward and backward for Linkbot-I as a two-wheel vehicle */
#include <linkbot.h>
CLinkbotI 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 Linkbot-I by rolling two wheels.

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

```
#include <linkbot.h>
```

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

```
CLinkbotI robot;
```

creates the variable `robot` for controlling a Linkbot-I. The statement also connects the variable `robot` to a Linkbot-I that has been previously configured with the computer as described in Section 2.2 on page 15.

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 **CLinkbotI**. In Program 3.2, this member function of the class **CLinkbotI** is called to move joints of a Linkbot-I.

The line

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

displays the following output in the input/output pane.

```
Here comes a robot!
```

A Linkbot-I can be configured as a two-wheel robot. In this case, both joints 1 and 3 can rotate together to roll forward or backward. The member function **driveAngle**() causes both joints 1 and 3 to drive the Linkbot-I 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 Linkbot-I forward 360 degrees for both joints 1 and 3 by

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

Then, it drives the Linkbot-I backward 360 degrees for both joints 1 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 Linkbot-I configured as a two-wheel robot.

All member functions of the class **CLinkbotI** for motion including **driveAngle()** expect input angles in degrees. In section 10.8, 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 Linkbot 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 Linkbot-I will make a full rotation for both joints 1 and 3. Finally, the following output will be displayed in the input/output pane.

```
Cool!
```


Do Exercise 1 on page 36.

3.2.1 Summary

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

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

   ```
   #include <linkbot.h>
   CLinkbotI robot;
   ```

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

2. Call the **CLinkbotI** member function

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

   to drive a Linkbot-I forward or backward by rolling both joints 1 and 3 with the specified angle, relative to their current positions for both joints.

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

3.2.2 Terminology

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

3.2.3 Exercises

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

3.3 Monitor Joint Angles Using the Robot Control Panel in Debug Mode

When a Linkbot program is executed, you can monitor the motion of its joints in the section of the Joint Position Control on the Robot Control Panel in Linkbot Labs as shown in Figure 2.8 in section 2.3.1 on page 18. As the Linkbot is moving, the vertical sliders for joints and the joint angles displayed above the sliders will be dynamically updated.

You can run the program `driveangle.ch` in Program 3.2 to monitor the joint angles of the Linkbot on the Robot Control Panel in Linkbot Labs. When you run the program, joint 1 will move from 0 to 360 degrees first and then back to 0 degree. Joint 3 will move from 0 to -360 degrees first, then back to 0 degree, as you can monitor the change of joint angles on the Robot Control Panel in Linkbot Labs as shown in Figure 3.7.

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. The joint angles of the Linkbot at this point on the Robot Control Panel on Linkbot Labs is shown in Figure 3.7. Since the low-cost Linkbot is not high precision, the joint angles may not reach and stay in the exact goal positions. When you monitor the joint angles in Linkbot Labs, as shown in Figure 3.7, the joints are close to the goal positions. 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.

Please note that when Linkbot Labs is in Robot Control Panel, it constantly communicates with a connected Linkbot to update the positions on the panel. If a class with a large number of Linkbots connected wirelessly to many computers to run programs at the same time, it may jam the communication channel. It is recommended to disable the Robot Control Panel by switching Linkbot Labs to the opening state when many computers run Ch programs to control Linkbots wirelessly in a classroom.

E Do Exercises 1 and 2 on page 38.

3.3.1 Summary

1. Execute a Linkbot program line-by-line using the command `Next` on the debug bar in ChIDE while monitoring joint angles of a Linkbot on the Robot Control Panel in Linkbot Labs.

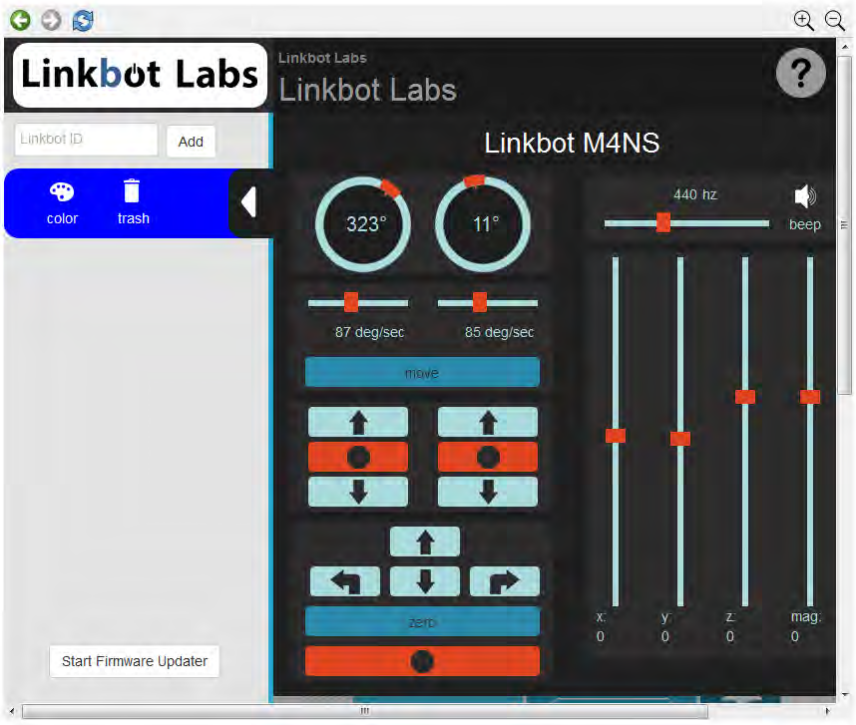


Figure 3.7: Monitoring the movements of joints of a Linkbot on the Robot Control Panel in Linkbot Labs.

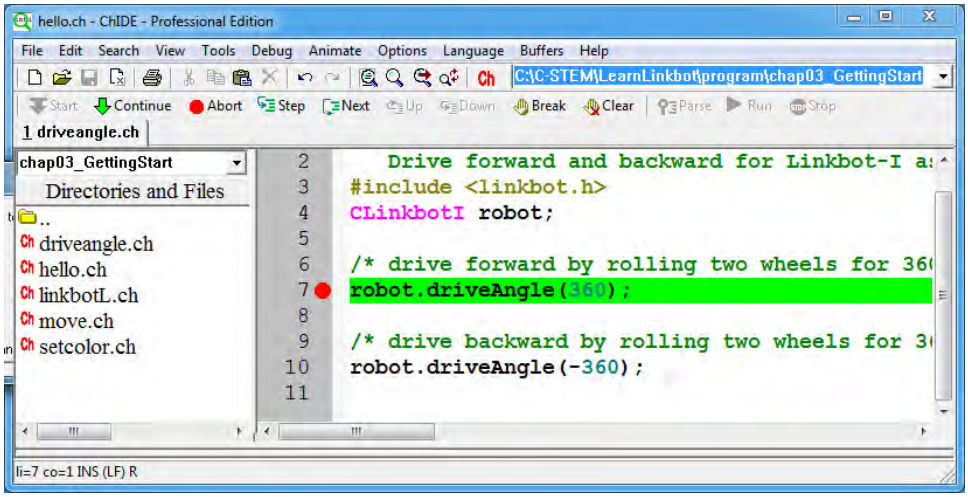


Figure 3.8: Running the program `driveangle.ch` in debug mode.

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 Linkbot-I 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 Linkbot Labs.

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 Linkbot-I 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 faceplates (joints 1 and 3) to roll the Linkbot-I forward. The syntax of the member function **driveDistance**() is as follows.

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

Unlike the member function **driveAngle**(), the distance for the Linkbot-I 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 Linkbot-I, 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 <linkbot.h>
CLinkbotI robot;

/* drive forward for 8 inches with the radius 1.75 inches for two wheels */
robot.driveDistance(8, 1.75);

/* drive backward for 5 inches with the radius 1.75 inches for two wheels */
robot.driveDistance(-5, 1.75);
```

Program 3.3: Moving a Linkbot-I with a specified distance using **driveDistance**().

For example, Program 3.3 drives a Linkbot-I configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. The program drives the Linkbot-I forward for 8 inches by the statement

```
robot.driveDistance(8, 1.75);
```

Then, it drives the Linkbot-I backward for 5 inches

```
robot.driveDistance(-5, 1.75);
```

with a negative value −5 for the distance of the first argument of the member function **driveDistance**().
Do Exercise 1 on page 39.

3.4.1 Summary

1. Call the **CLinkbotI** member function

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

   to drive a Linkbot-I 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 Linkbot-I. 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 1.75 inches.

3.5 Set the LED Color

Section 2.4 demonstrated how to change the LED color of a Linkbot through the Sensors Panel of Linkbot Labs. It is also possible to change the LED color of a Linkbot within a Ch program to add visual effects. Program 3.4 shows an example of how this is done.

```
/* File: setcolor.ch
   Set the LED color before and after the movement */
#include <linkbot.h>
CLinkbotI robot;

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

/* change the LED color to blue */
robot.setLEDColor("blue");
/* drive backward by rolling two wheels for 360 degrees */
robot.driveAngle(-360);

/* change the LED color to red */
robot.setLEDColor("red");
robot.driveAngle(-360);
```

Program 3.4: Setting the LED color for a Linkbot-I.

Program 3.4 builds upon Program 3.2 by changing the LED color of the Linkbot-I after it moves forward, and again after it moves backward. The color of a Linkbot-I's LED can be changed using the **CLinkbotI** member function **setLEDColor()**. The general syntax of this member function is as follows.

```
robot.setLEDColor(color);
```

The argument `color` specifies the desired color. A full list of the 137 possible colors that can be used with the member function **setLEDColor()** can be found in Appendix C. If a color other than those listed in Appendix C is used, the LED color remains unchanged and an error message is printed to the input/output pane in ChIDE.

In Program 3.4, after the Linkbot-I moves forward, the line

```
robot.setLEDColor("blue");
```

changes the LED color of the Linkbot-I to blue. And after the Linkbot-I moves backward, the line

```
robot.setLEDColor("red");
```

changes the LED color of the Linkbot-I to red.

E Do Exercise 1 on page 41.

3.5.1 Summary

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

   ```
   robot.setLEDColor(color);
   ```

   to set the LED color of a Linkbot-I.

3.5.2 Terminology

robot.setLEDColor().

3.5.3 Exercises

1. Write a program `setcolor2.ch`, based on Program 3.4, to change the LED color of a Linkbot-I as follows: First, set the LED color to brown and then drive the Linkbot-I forward by 360 degrees. Then set the LED color to green and drive the Linkbot-I backward by 360 degrees. Finally, set the LED color to pink.

3.6 Play Melody

In addition to LED color, a Linkbot contains a buzzer, which can be used to play melody and music notes. The **CLinkbotI** 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 D.1 in Appendix D lists all available melodies. The user can develop your own melodies as decribed in section 12.7 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 3.5 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, 1.75);
```

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 <linkbot.h>
CLinkbotI robot;

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

/* drive forward for 5 inches with the radius 1.75 inches for two wheels */
robot.driveDistance(5, 1.75);

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

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

Program 3.5: Playing the JingleBells in a Linkbot-I.

In section 12.7, you will learn how to create your own melodies for the member function **playMelody**() to play.

E Do Exercise 1 on page 42.

3.6.1 Summary

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

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

   play a melody.

3.6.2 Terminology

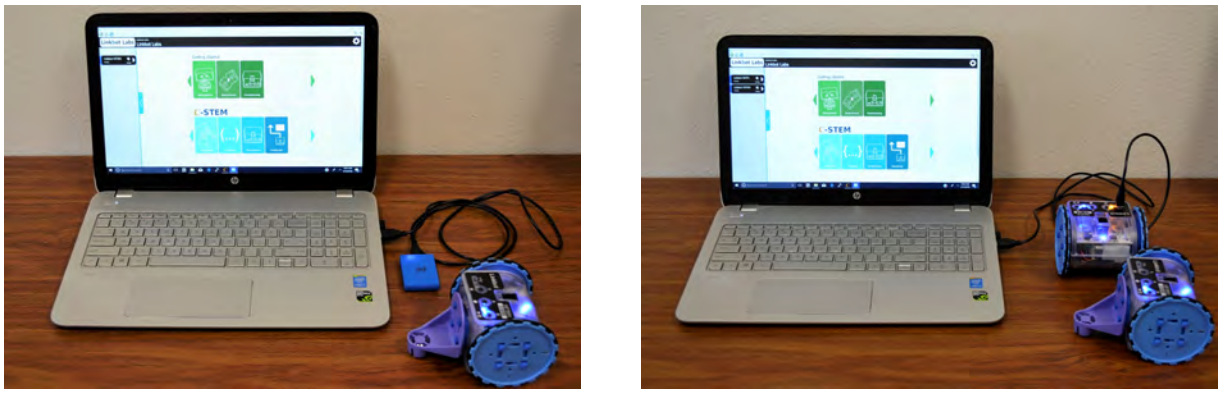
song, melody, **robot.playMelody**().

3.6.3 Exercises

1. Write a program `playmelody2.ch`, based on Program 3.5, 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.

3.7 Control a Linkbot Wirelessly Without Connecting with a USB Cable

Linkbot Labs in a computer can control one or multiple Linkbots. A dongle or a Linkbot needs to be connected to the computer through the USB cable. For controlling multiple Linkbots, a Linkbot connected to the computer through the USB cable can also act as a dongle for Linkbot Labs to connect and control other Linkbots through the ZigBee wireless communication, as shown in Figure 3.10. The other Linkbots in the network can be located as far as 100 meters.



(a) Connecting a Linkbot using a dongle

(b) Using a Linkbot as a dongle

Figure 3.10: The configuration for Linkbot Labs to control a Linkbot wirelessly.

Multiple Linkbots at the top can also be controlled by a program. Also, you do not even need to connect the Linkbot acting as a dongle in Linkbot Labs so long as it is connected physically through the USB cable, as shown in Figure 2.13 on page 22.

E Do Exercise 1 on page 43.

3.7.1 Summary

1. Control a Linkbot wirelessly without connecting to a USB cable.

3.7.2 Terminology

wireless, dongle.

3.7.3 Exercises

1. Use a Linkbot as a dongle connected through a USB cable to a computer. Control another Linkbot wirelessly using a program developed previously.

CHAPTER 4

Robot Simulation with RoboSim

RoboSim is a robot simulation environment for programming Linkbots and Lego Mindstorms NXT/EV3. Almost any program that can control hardware Linkbots, except for programs using a controller 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 Linkbots 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 1.4. 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.

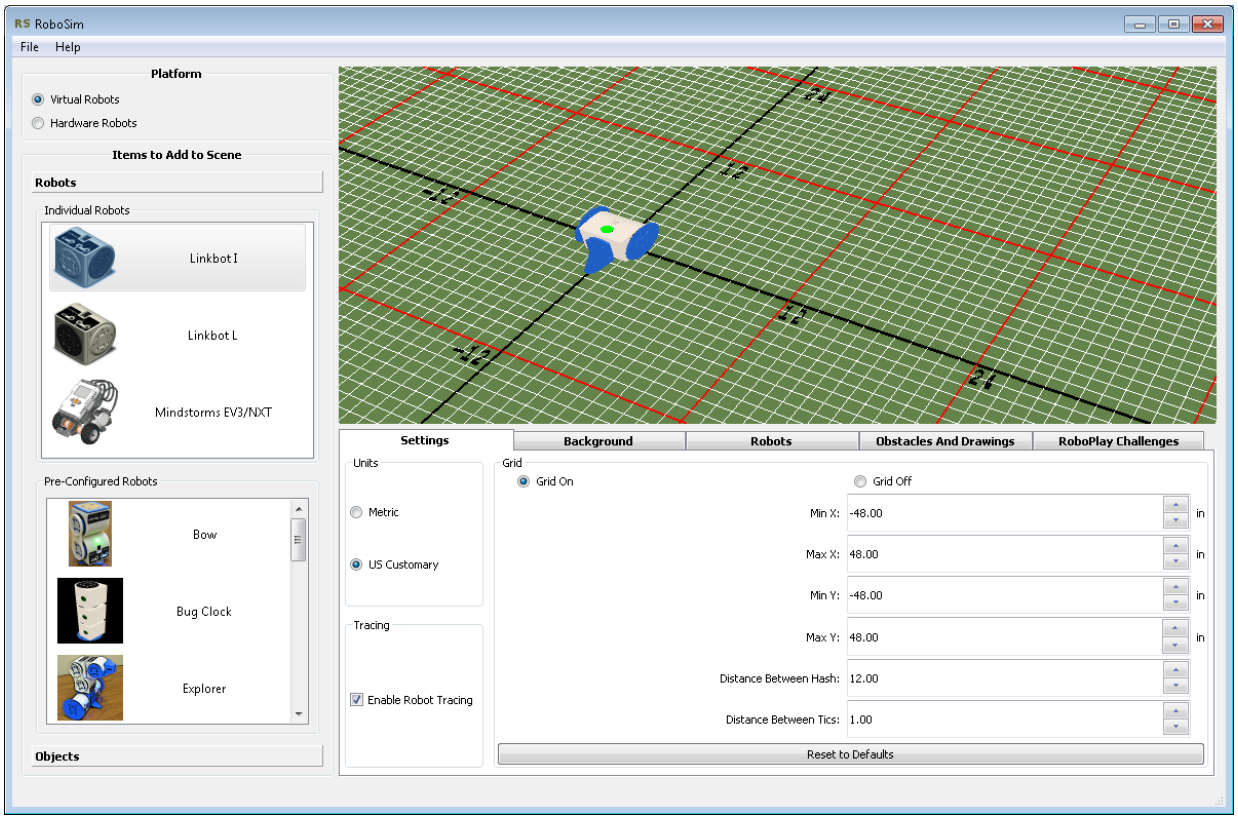


Figure 4.1: The RoboSim GUI.

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.

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.

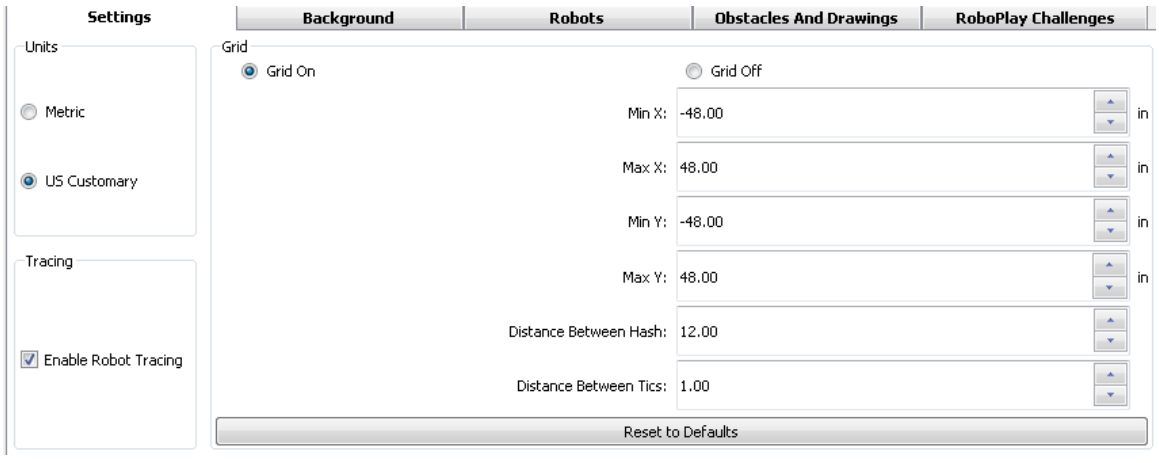


Figure 4.3: The default grid spacing in the US Customary units.

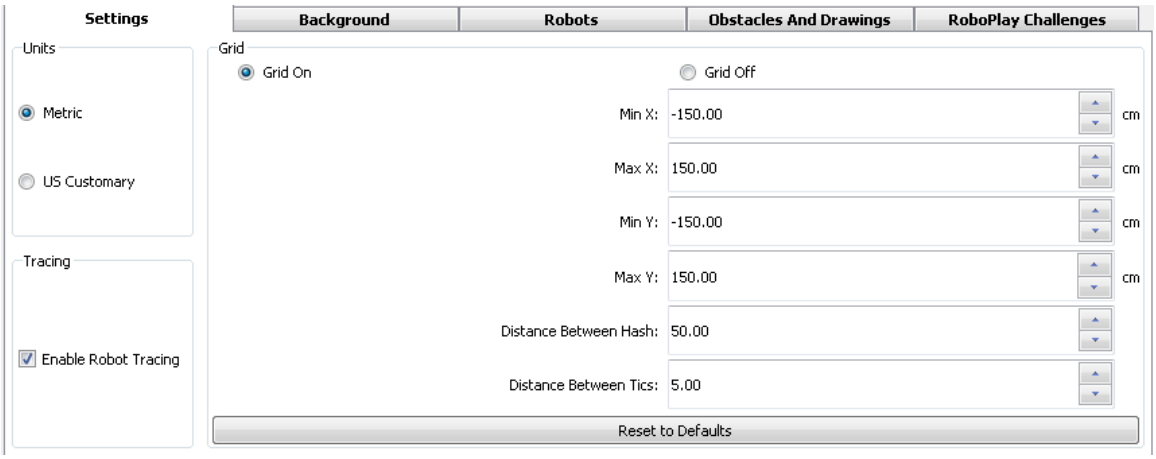


Figure 4.4: The default grid spacing in the Metric units.

4.1.5 Individual Robot Configuration

Initial robot configurations can either be done through the **Individual Robots** or **Pre-Configured Robots** section. 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.

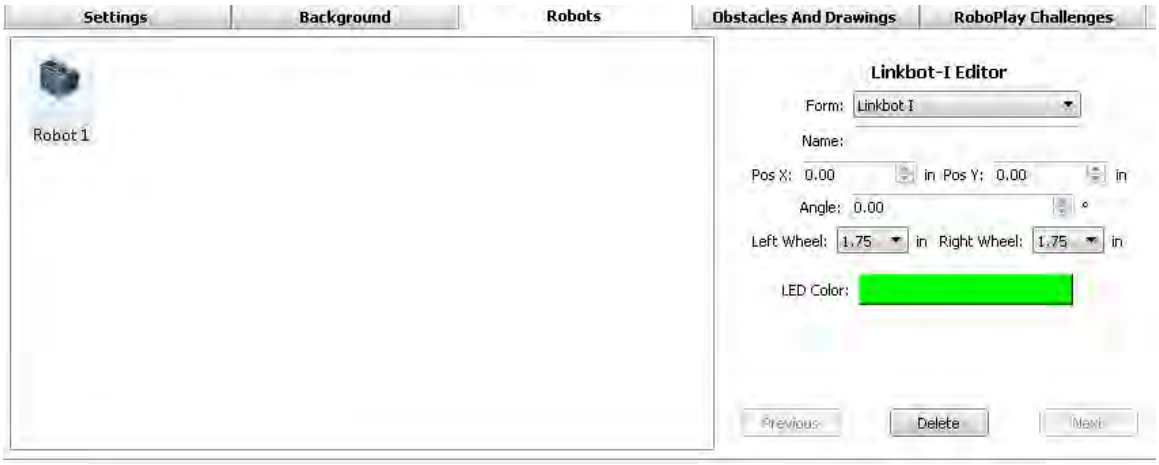


Figure 4.6: Individual robot configuration dialog.

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. Linkbot-I, Linkbot-L, Mindstorms NXT/EV3. The options are presented in a drop down menu as shown in Figure 4.7.

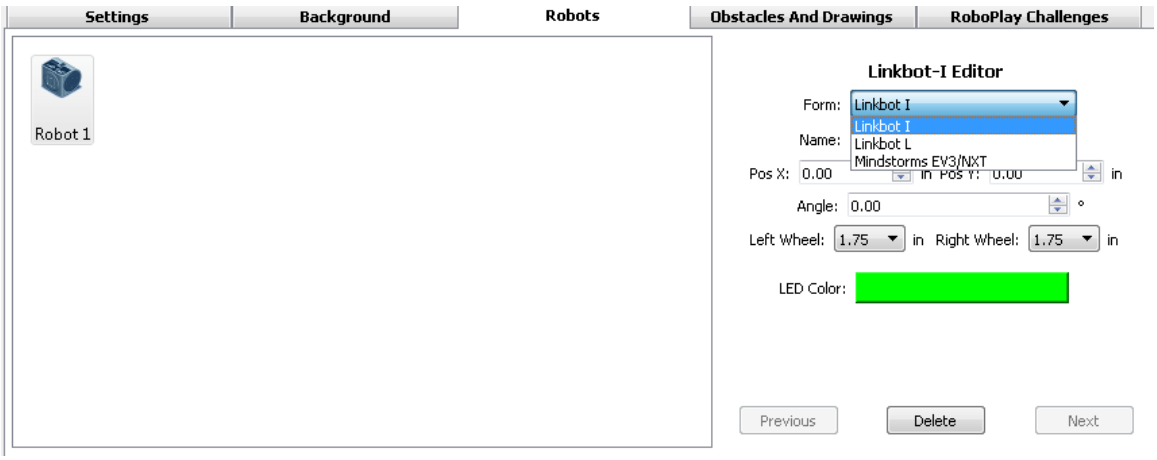


Figure 4.7: Picking a robot type.

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 wheels and a caster connected, a drop down menu is provided to select different wheel sizes. The options listed are the radii of the wheels typically used with Linkbots. Each wheel is drawn with a series of dots along the one radius to easily show the rotation of the wheel. The correlation between wheel radius and number of dots is given in Table 4.1.

Number of Dots	Wheel Radius
2	custom radius
3	1.625 inches / 4.13 centimeters
4	1.75 inches / 4.45 centimeters
5	2.00 inches / 5.08 centimeters

Table 4.1: Wheel sizes and number of dots.

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.

4.1.6 Pre-Configured Robots

In addition to positioning robots independently within the RoboSim, some **Pre-Configured Robots**, as shown in Figure 4.8, which represent commonly used Linkbot configurations are available to the user. Selecting one of these options will display a picture of the configuration built with the hardware Linkbots, corresponding to a Ch robot program presented in Chapter 16. When one of these options is selected, the specific configuration for this setup is passed into Ch.

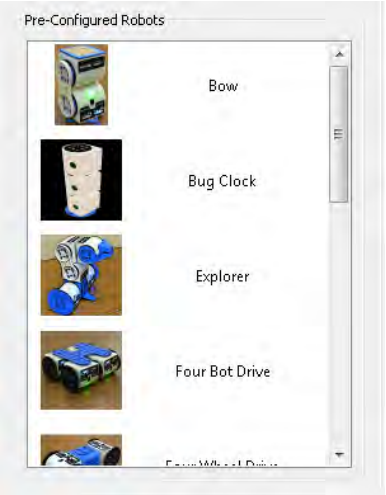


Figure 4.8: Preconfigured robot configurations with Linkbots.

E Do Exercises 1 and 2 on page 50.

4.1.7 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 (Linkbot-I, Linkbot-L, or Mindstorms EV3/NXT), 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. Use Preconfigured Linkbot Configurations to run a robotic system with multiple connected Linkbots.
6. Set grid lines in the RoboSim scene with the US Customary units or Metric units.
7. Enable or disable the tracing of robot trajectories.

4.1.8 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.

4.1.9 Exercises

1. Watch the video tutorial RoboSim in http://c-stem.ucdavis.edu/studio/tutorial/.
2. (a) Launch a RoboSim GUI.
   (b) Add a Linkbot-I 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 1.75 inches to joints 1 and 3.

(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 4.9 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 Linkbot-I as a two-wheel vehicle */
#include <linkbot.h>
CLinkbotI robot;

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

Program 4.1: Moving a Linkbot forward by angle.

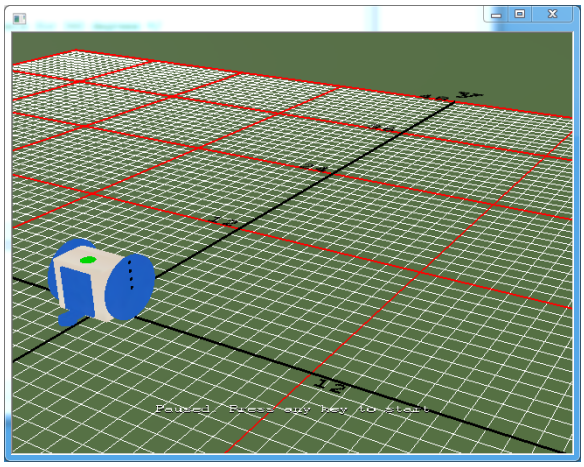


Figure 4.9: A RoboSim scene with a virtual robot at its starting position.

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 4.10.

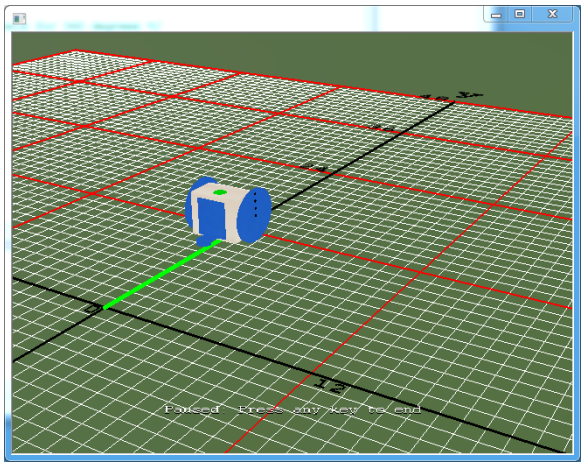


Figure 4.10: A RoboSim scene with a virtual robot and its trajectory traced.

Do Exercise 1 on page 53.

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 <linkbot.h>
CLinkbotI 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.

Do Exercise 2 on page 53.

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 Linkbot-I forward by rotating both joints 1 and 3 for 720 degrees.
   (b) Based on the setup on the RoboSim GUI described in Exercise 2 on page 50, 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 39, 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 4.11 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.

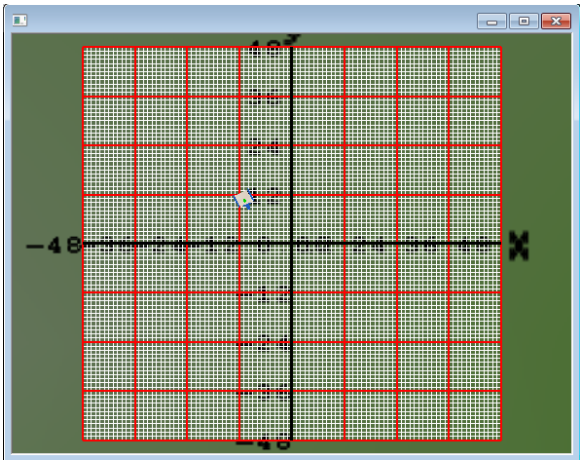


Figure 4.11: A RoboSim scene with the overhead viewing angle.

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 4.12 shows a RoboSim scene with a traced robot trajectory only, without the robot displayed.

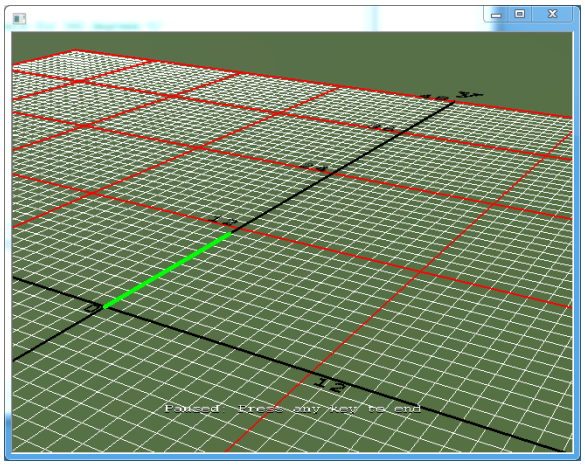


Figure 4.12: A RoboSim scene with a traced robot trajectory only.

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 4.13 with the position (0, 10.9817) inches for the x and y coordinates for robot 1.

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

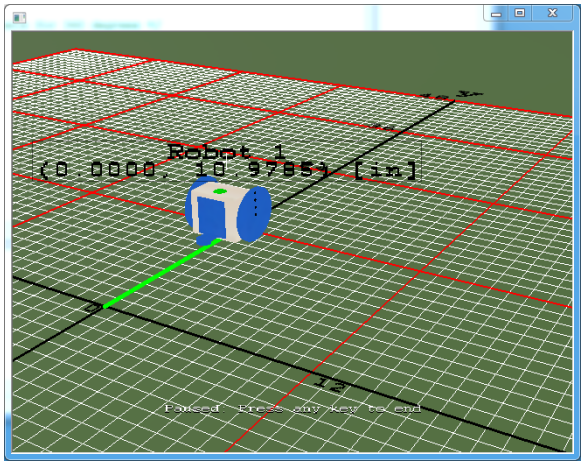


Figure 4.13: A RoboSim scene with a virtual robot and its position displayed.

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 Linkbot-I 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 Linkbot-I as a two-wheel vehicle */
#include <linkbot.h>
CLinkbotI 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 Linkbot 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.

Do Exercise 2 on page 57.

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 57 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 Linkbot-I forward by rotating both joints 1 and 3 for 720 degrees using the member function **driveAngle**(). Then, dirve the robot backward by rotating both joints 1 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 Linkbot-I 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 <linkbot.h>
CLinkbotI robot;
```

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

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 <linkbot.h>
CLinkbotI 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 62.

Application: Using Variables for Distance and Radius

Problem Statement:
Write a program `distance.ch` to drive a Linkbot-I 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=1.75; // the radius of 1.75 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 <linkbot.h>
CLinkbotI robot;
double distance=5;  // the distance of 5 inches to drive forward
double radius=1.75; // the radius of 1.75 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 62.

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 Linkbot. 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 Linkbot-I. 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 **CLinkbotI** 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.

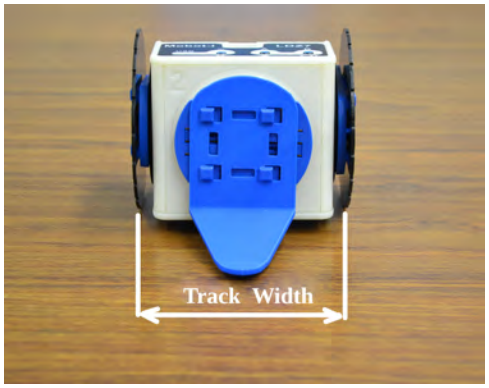


Figure 5.1: The track width for a two-wheel robot.

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 <linkbot.h>
CLinkbotI robot;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;   // 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 Linkbot-I 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.

You can run the program in ChIDE in debug mode with the command `Next` on the debug bar to watch the action of each motion statement, as you monitor the change of joint angles on the Robot Control Panel in Linkbot Labs.

E Do Exercises 1 and 2 on page 64.

5.2.1 Summary

1. Call the **CLinkbotI** member function

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

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

2. Call the **CLinkbotI** member function

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

   to turn a Linkbot-I 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 Linkbot-I 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, as you monitor the change of joint angles on the Robot Control Panel in Linkbot Labs.

5.3 Robot Simulation and Generating Robot Programs with RoboBlockly

Like RoboSim, RoboBlockly can be used to simulate the motion of Linkbots 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.

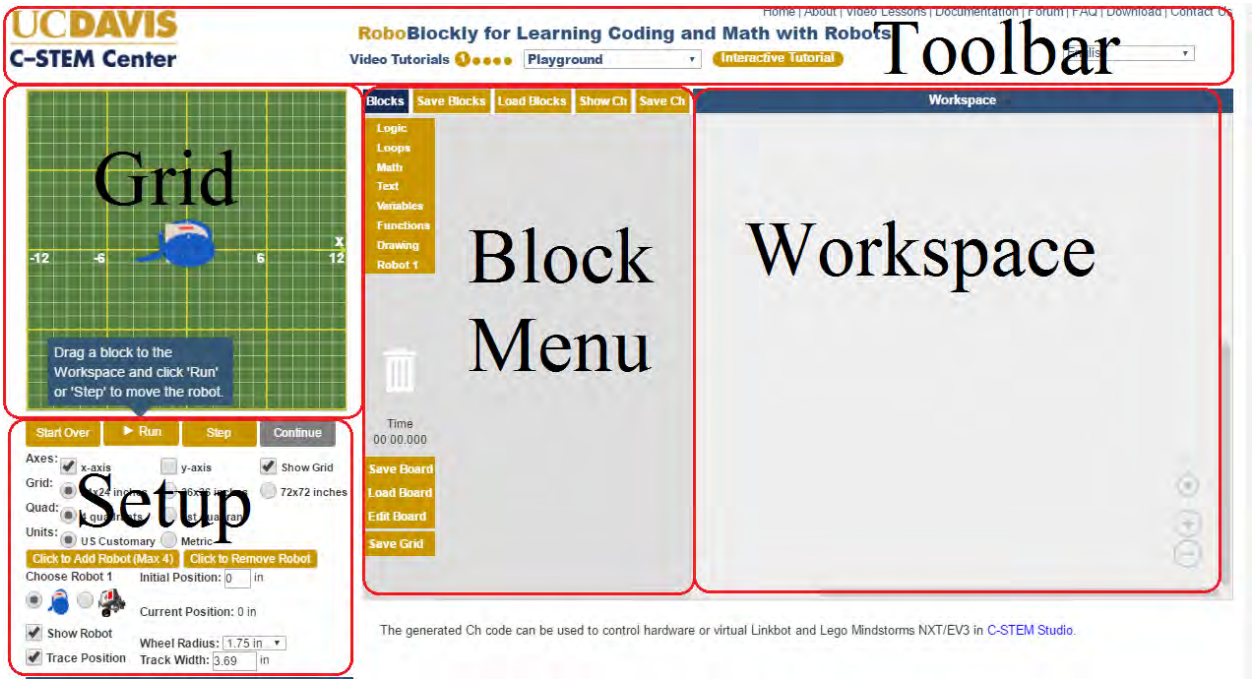


Figure 5.2: The graphical user interface (GUI) in RoboBlockly.

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.

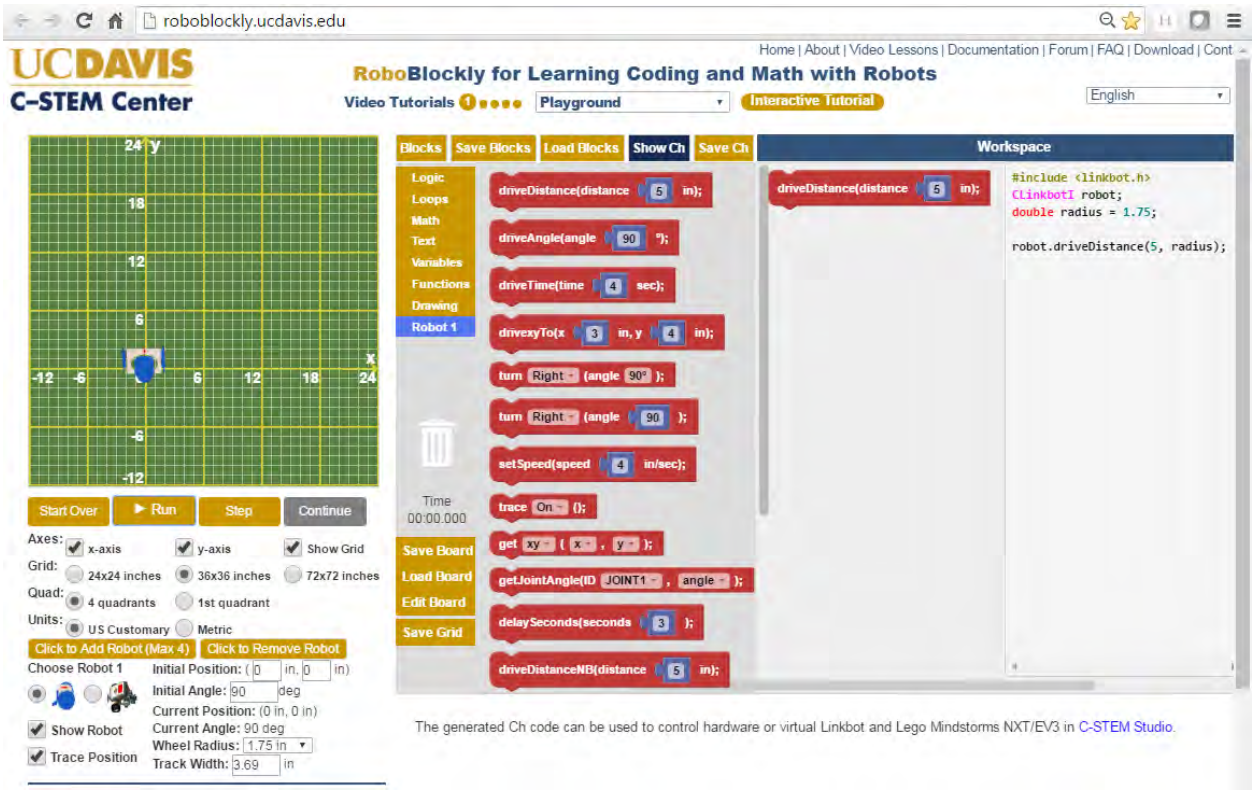


Figure 5.3: Drive a robot using **driveDistance**() in RoboBlockly.

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

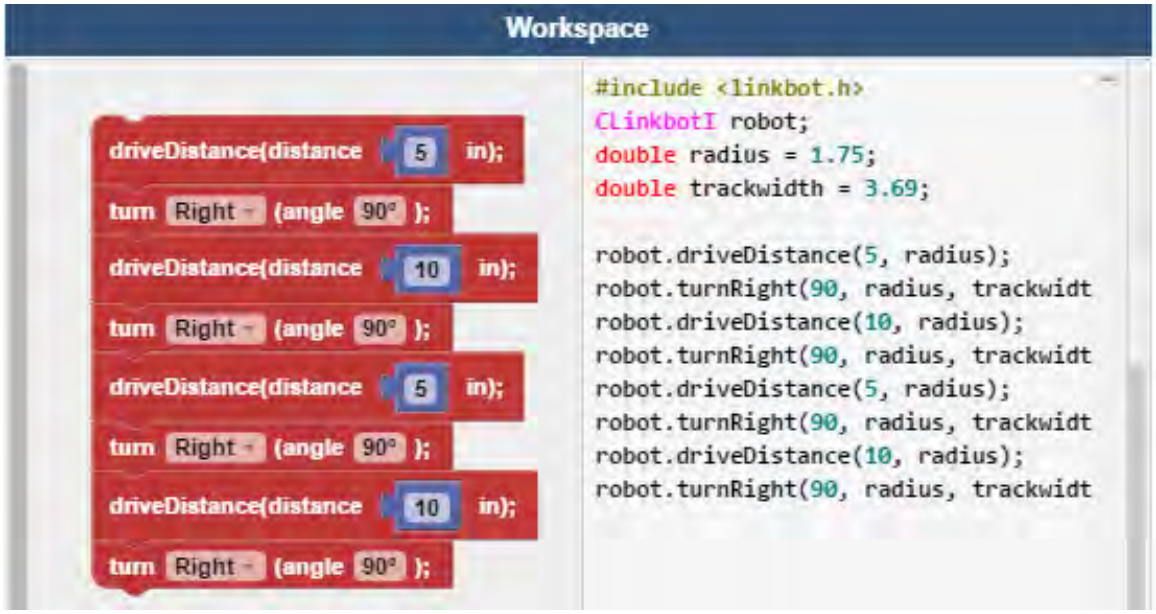


Figure 5.4: The blocks and Ch code for tracing a 5x10 rectangle using a Linkbot in RoboBlockly.

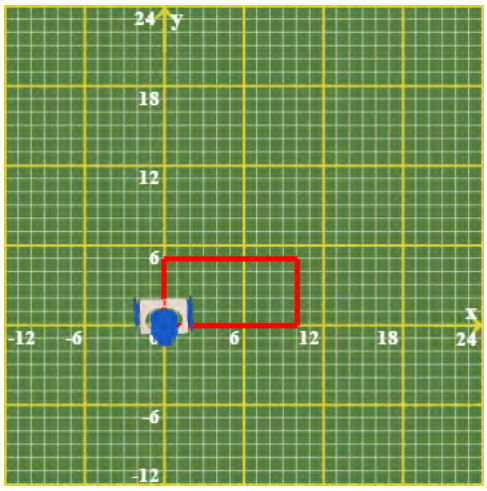


Figure 5.5: The 5x10 rectangle generated by the blocks shown in Figure 5.5.

E Do Exercises 1 and 3 on page 67.

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.

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 Linkbot 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

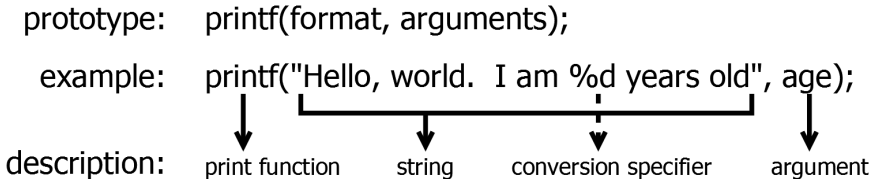


Figure 6.1: A format for using the function **printf**() and an example.

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 125.

```
/* 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 73.

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
```


Do Exercises 1 and 3 on page 73.

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 77.

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

Problem Statement:
Write a program `distanceangle_p.ch` to control a Linkbot-I 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 <linkbot.h>
CLinkbotI robot;
double distance; // declare variable 'distance' to drive
double angle;    // declare variable 'angle' for turnRight
double radius = 1.75;     // radius of the wheels
double trackwidth = 3.69; // track width inches

/* User inputs distance and angle */
printf("Enter the distance to drive\n");
scanf("%lf", &distance);
robot.driveDistance(distance, radius);

/* turn right by angle */
printf("Enter the angle to turn right \n");
scanf("%lf", &angle);
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 77.

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 Linkbot. 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 Linkbot forward and backward. Test the program with the joint angle 360 degrees.

3. Write a program `drivedistance3_p.ch` to control a Linkbot-I. 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 Linkbot forward. Test the program with the distance 6 inches and radius of 1.75 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 <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double radius = 1.75;       // 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.

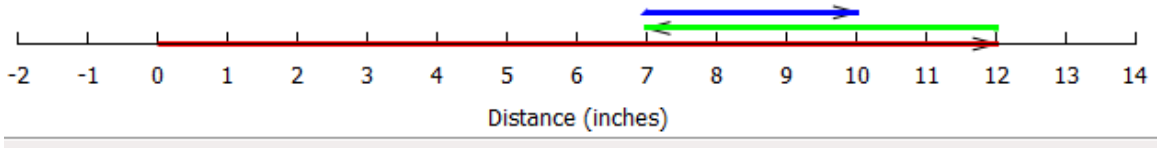


Figure 6.2: A number line for distances of a two-wheel robot, generated by Program 6.5.

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.


Do Exercise 1 on page 81.

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 <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double radius = 1.75;       // 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.

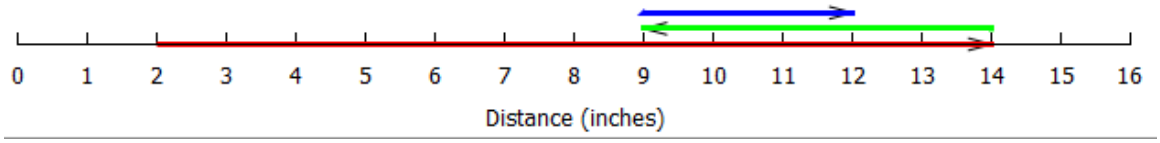


Figure 6.3: A number line for distances of a two-wheel robot with an initial offset from the origin, generated by Program 6.6.

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 81.

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();
   ```

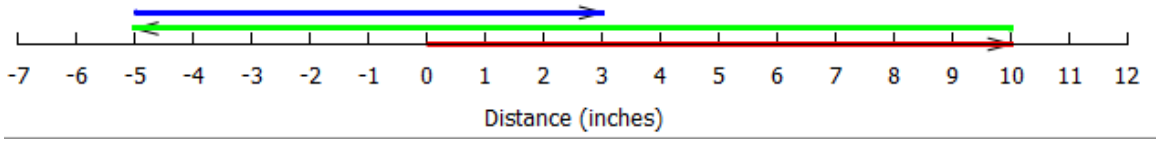
   to generate the final graph.

6.3.2 Terminology

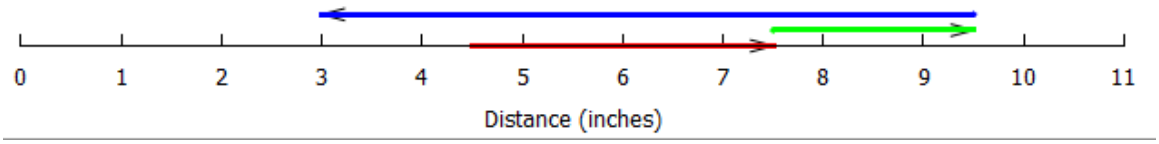
number line, **#include <chplot.h>**, **CPlot**, **plot.label()**, **plot.numberLine()**, **plot.plotting()**.

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 Linkbot-I configured as a two-wheel drive robot and generate a number line for the distance of the robot. Assume the radius of wheels is 1.75 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 Linkbot-I configured as a two-wheel drive robot and generate a number line for the distance of the robot. Assume the radius of wheels is 1.75 inches.

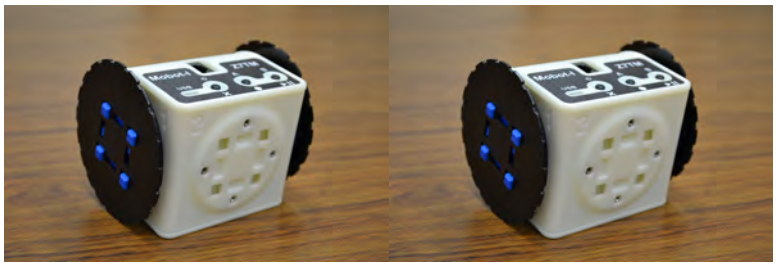


CHAPTER 7

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

7.1 Control a Group of Linkbots with Identical Movements

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



(a) 1st Linkbot (b) 2nd Linkbot

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

```
/* File: group.ch
   Control multiple robot modules simultaneously using the CLinkbotIGroup class */
#include <linkbot.h>
CLinkbotI robot1, robot2;
CLinkbotIGroup 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 Linkbots with identical movements.

Program 3.2 on page 34 controls a single Linkbot-I to roll forward by 360 degrees and then roll backward by 360 degrees. Program 7.1 peforms identical movements for two Linkbot-Is in the same manner as Program 3.2.

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

```
CLinkbotIGroup group;
```

creates a variable `group` of class **CLinkbotIGroup**. The class **CLinkbotIGroup** is defined in the header file **linkbot.h**, just as the classes **CLinkbotI** and **CLinkbotL** are. The general syntax of the **CLinkbotIGroup** member function **addRobot**(), which is used to add a Linkbot-I to a group, is as follows:

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

The argument `name` represents the variable name of the Linkbot-I that you want to add to the group. The next two lines

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

add each Linkbot-I 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 Linkbot-Is to a group.

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

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 Linkbot-I in Program 3.2.

E Do Exercises 1, 2, and 3 on page 84.

7.1.1 Summary

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

```
#include <linkbot.h>
CLinkbotIGroup group;
```

   for controlling a group of Linkbot-Is.

2. Call the **CLinkbotIGroup** member function

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

   to add a single Linkbot-I to a group.

3. Call the **CLinkbotIGroup** member function

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

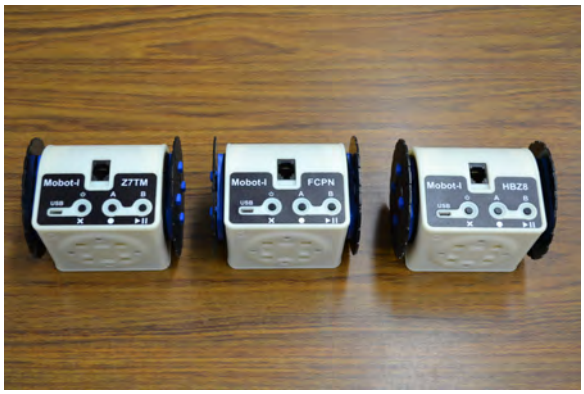
   to drive all Linkbot-Is in a group forward or backward by the same specified angle, relative to their current positions.

7.1.2 Terminology

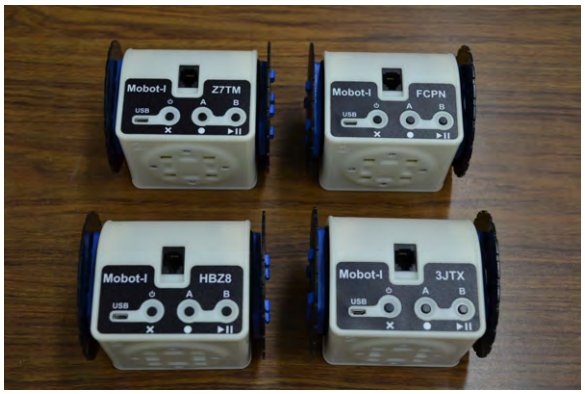
CLinkbotIGroup, **group.addRobot()**, **group.driveAngle()**, groups, identical movements.

7.1.3 Exercises

1. Run the program `group.ch` in Program 7.1 to perform identical movements for two Linkbots.
2. Write a program `group2.ch` to perform identical movements, as presented in Program 7.1, for three Linkbots.



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



7.2 Control an Array of Linkbots 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 Linkbot-I's in synchronized motion using an array of 4 elements.
 *
 *        1       2
 *      |--|    |--|
 *      |__|    |__|
 *
 *       3       4
 *      |--|    |--|
 *      |__|    |__|
 *
 * Make sure that robots are attached with wheels. */

#include <linkbot.h>

CLinkbotI robot[4];        /* declare an array of 4 Linkbot-I's */
CLinkbotIGroup group;      /* declare a Linkbot-I group  */
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69; // the track width, the distance between two wheels

group.addRobots(robot);    /* add 4 Linkbot-I's 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 Linkbot-Is with identical movements using an array of 4 elements.

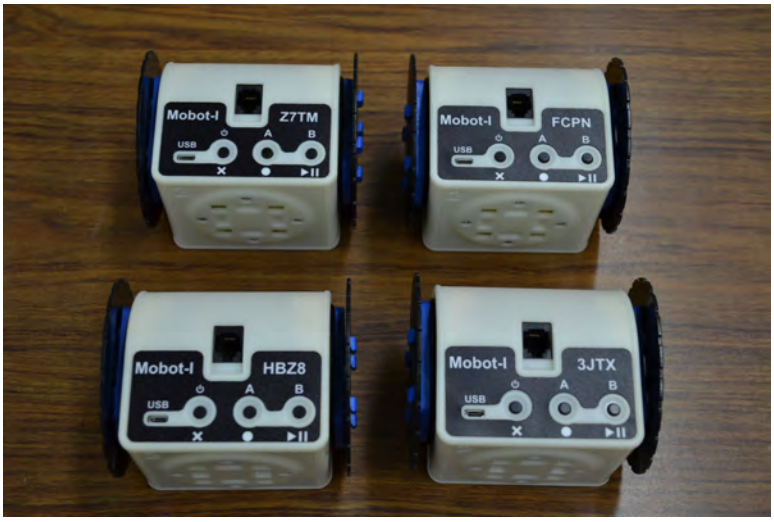


Figure 7.2: A group of four Linkbots with identical movements.

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

```
CLinkbotI robot[4];
```

declares an array of 4 Linkbot-Is. 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 **CLinkbotIGroup** member function **addRobots**(). The syntax for this new function is

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

where name refers to an array of Linkbot-Is, instead of just one Linkbot-I. 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 Linkbot-Is to the group with just one line of code. This is easier than using four lines of code to add four Linkbot-Is to the group.

With Program 7.2 we discovered that the class **CLinkbotIGroup** 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 Linkbot-I but all four Linkbot-Is left by 90 degrees at the same time. Similarly, the line

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

turns all four Linkbot-Is 135 degrees right. The statements

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

drive all Linkbot-Is in the group forward by 5 inches first, then backward by 5 inches.

E Do Exercises 1 and 2 on page 88.

7.2.1 Summary

1. Declare an array variable to store a group of Linkbot-Is.

   ```
   CLinkbotI robot[4];
   ```

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

2. Call the **CLinkbotIGroup** member function

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

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

3. Call the **CLinkbotIGroup** member functions

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

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

4. Call the **CLinkbotIGroup** member function

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

to drive a group of Linkbot-Is 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 Linkbots.
2. Write a program `grouparray2.ch` to control a group of 9 Linkbots with the same motion as presented in the program `grouparray.ch` in Program 7.2.

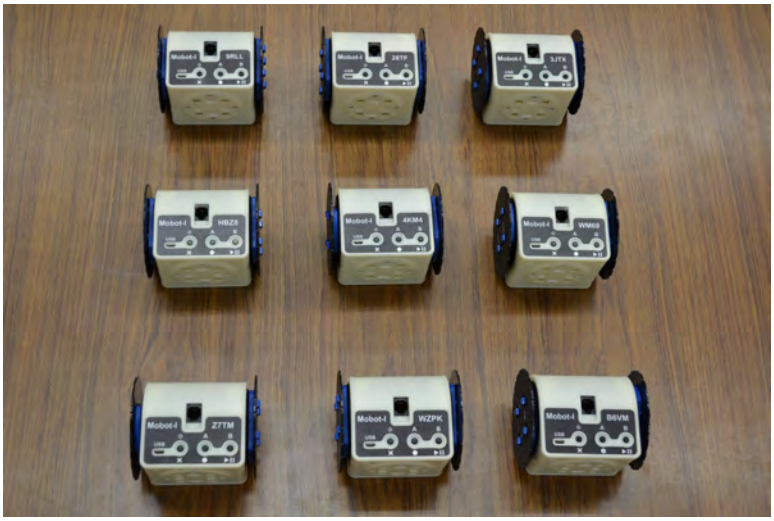


Figure 7.3: A group of 9 Linkbots with identical movements.

CHAPTER 8

Controlling a Linkbot-I as a Two-Wheel Robot

A Linkbot-I can be configured as a two-wheel robot as shown in Figure 8.1. In this configuration, joints 1 and 3 are attached with two wheels. In this Chapter, we will learn more programming features on how to control a Linkbot-I 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 **CLinkbotI** class includes two additional member functions that can be used for the two-wheel Linkbot-I configuration. The first of these two member functions is **setSpeed()**, which can be used to set both joints 1

and 3 of a Linkbot-I 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 Linkbot-I 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 Linkbot-I 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 Linkbot-I with a specified speed and distance, as demonstrated solving the following problem.

> **Problem Statement:**
> A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>
CLinkbotI robot;
double radius=1.75; // 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 1 and 3 */
robot.driveAngle(360);

/* drive the robot 360 degrees backward for joints 1 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 1.75 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 joints 1 and 3 of the Linkbot-I to drive the robot at 2.5 inches per second with a wheel radius of 1.75 inches. Then when the next two lines

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

are executed, the Linkbot-I 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 Linkbot-I forward 5 inches at the same speed of 2.5 inches per second.

E Do Exercises 1 and 2 on page 97.

8.1.2 Control a Linkbot-I 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 Linkbot-I by providing those values in the program specifically. In this section we will learn how to set the speed and distance of a Linkbot-I 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>
CLinkbotI 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
1.75
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 Linkbot-I configured as a two-wheel robot with the radius of 1.75 inches for wheels, the above execution will drive the Linkbot-I forward 5 inches at the speed of 1 inch per second.

E Do Exercise 3 on page 98.

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 Linkbot-I has moved using a specific wheel radius. The general syntax of this function is

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

where `distance` specifies how far the Linkbot-I 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 move all of its joints to the zero position, as shown in Figure 2.3. 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. If the Linkbot is already in the zero position, its joints will not move.

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

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

is used to set all joints to their zero positions.

Problem Statement:
A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 driven distances
   using robot.getDistance() */
#include <linkbot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.75;       // radius of the wheel
double distance = 12;       // distance to drive in inches
double distance2;           // distance driven

/* 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 driven 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 Linkbot. 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 Linkbots.

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.

Do Exercise 4 on page 98.

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 Linkbot-I 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 Linkbot-I has stopped moving using the member function **getJointAngle()**. We can convert this joint angle value into the actual distance that the Linkbot-I 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 Linkbot-I 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 Linkbot-I and `distance` is the distance a Linkbot-I 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.75;       // radius of the wheel
double distance = 12;       // distance to drive in inches
double angle;               // angle corresponding to the driven distance in degrees
double distance2;           // distance driven 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 1 */
robot.getJointAngle(JOINT1, 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 driven 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 Linkbot has stopped moving, then the statement

```
distance2 = angle2distance(radius, angle);
```

gives the actual distance in inches that were traveled by the Linkbot.

When Program 8.4 is executed, the output will be similar to that of Program 8.3.

E Do Exercise 5 on page 98.

8.1.5 Measure the Clock Time Using the Member Function systemTime()

You can time the motion of a Linkbot-I 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>

CLinkbotI robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.75;       // 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 Linkbot-I starts moving in order to record the current system time in seconds. Then the statement

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

drives the Linkbot-I 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 Linkbot-I 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 Linkbot was in motion. This time is then displayed, along with the theoretical time. In the case of Linkbot 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 Linkbot-I took 4.12 seconds.
The motion should take 4.00 seconds in theory.
```

Do Exercise 6 on page 98.

8.1.6 Summary

1. Call the **CLinkbotI** member function

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

   to set both joints 1 and 3 of a Linkbot-I 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Use the program `drivedistance_p.ch` in Program 8.2 to make the Linkbot-I move 12 inches in 5 seconds (Note that the speed is defined as the distance divided by the time).

4. A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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	1.75	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.

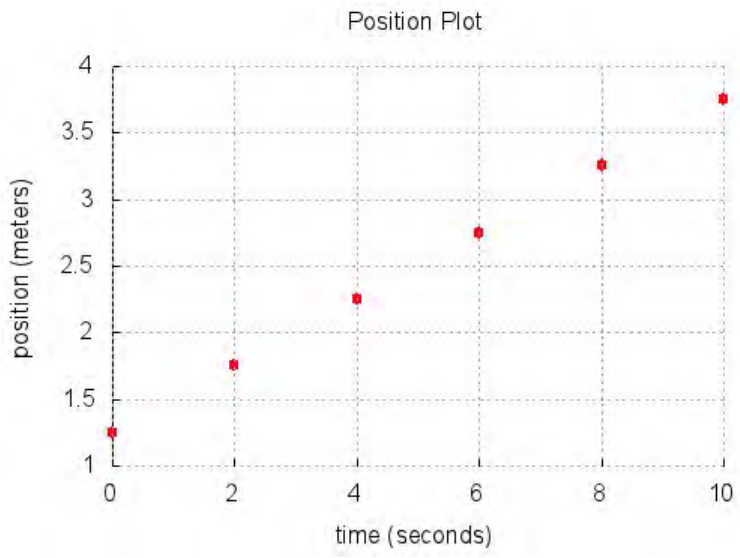


Figure 8.2: The scatter plot for the position versus time from Program 8.6.

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, 1.75, 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 Linkbot 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.

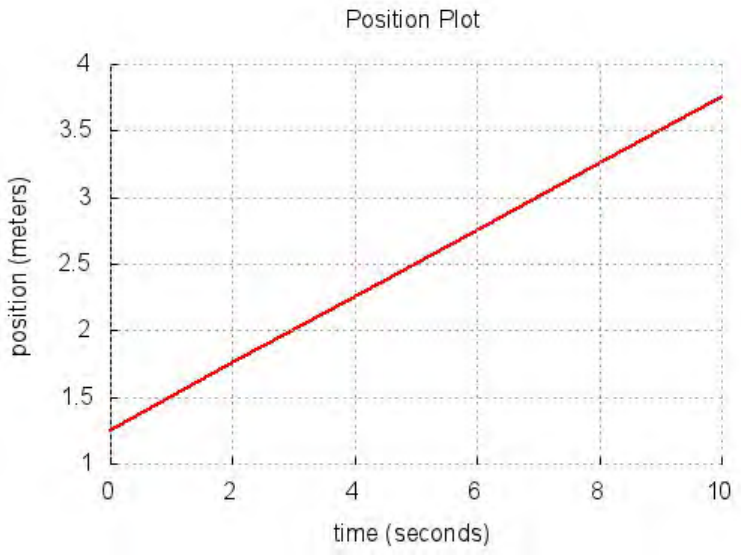


Figure 8.3: The line plot for the position versus time from Program 8.7.

 Do Exercise 1 on page 102.

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 Linkbot-I 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 Linkbot-I with the Specified Speed and Distance

We can record and plot the distance versus time for a Linkbot. To record distance data, we use the two **CLinkbotI** 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 Linkbot. 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 Linkbot-I 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 Linkbot-I.

Problem Statement:
A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Write a program `recorddistancescattern.ch` to

drive the Linkbot for 12 inches at the speed of 2.5 inches per second. Record the distance as the Linkbot 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 Linkbot Controller (CLC) 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 Linkbot Controller" on the C-STEM Studio shown in Figure 1.4 to launch CLC. With the setup for CLC 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 CLC 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.

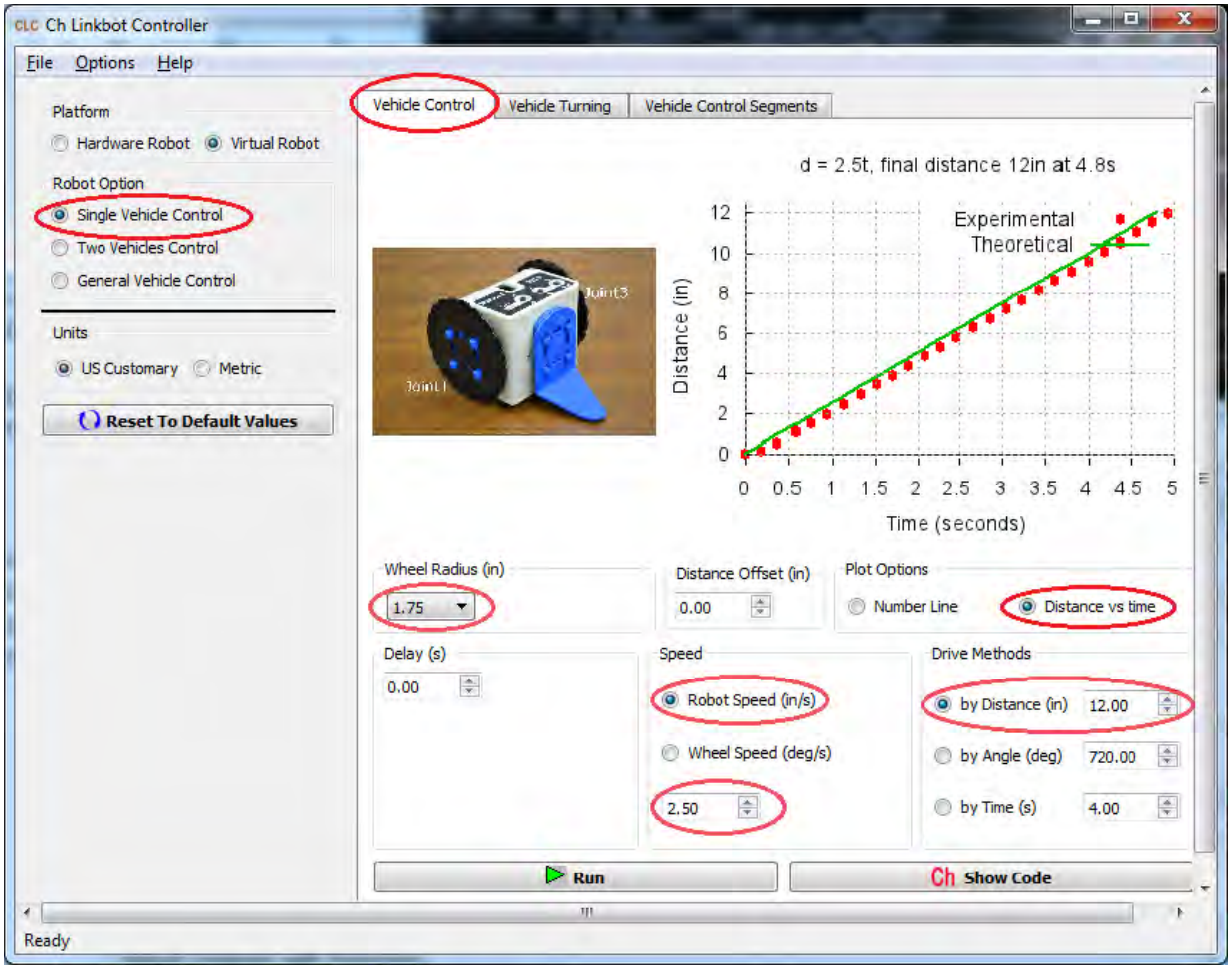


Figure 8.4: The setup of Ch Linkbot Controller for solving the problem.

E Do Exercise 1 on page 119.

```
/* File: recorddistancescattern.ch
   Record time and distances, plot the acquired data */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.75;       // 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 1 */
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 Linkbot-I 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 Linkbot-I starts moving. The statement

```
robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
```

starts recording data of the Linkbot-I every 0.1 second. After the Linkbot stops moving, the statement

```
robot.recordDistanceEnd(numDataPoints);
```

stops recording data from the Linkbot.

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

```
plot.data2DCurve(timedata, distances, numDataPoints);
```

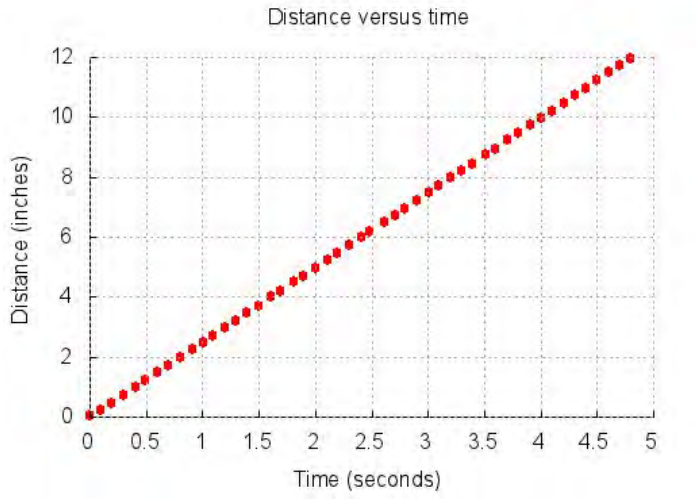


Figure 8.5: The scatter plot for the distance versus time from Program 8.8.

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 <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.75;       // 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 1 */
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 Linkbot-I 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.scattern(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 119.

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`.

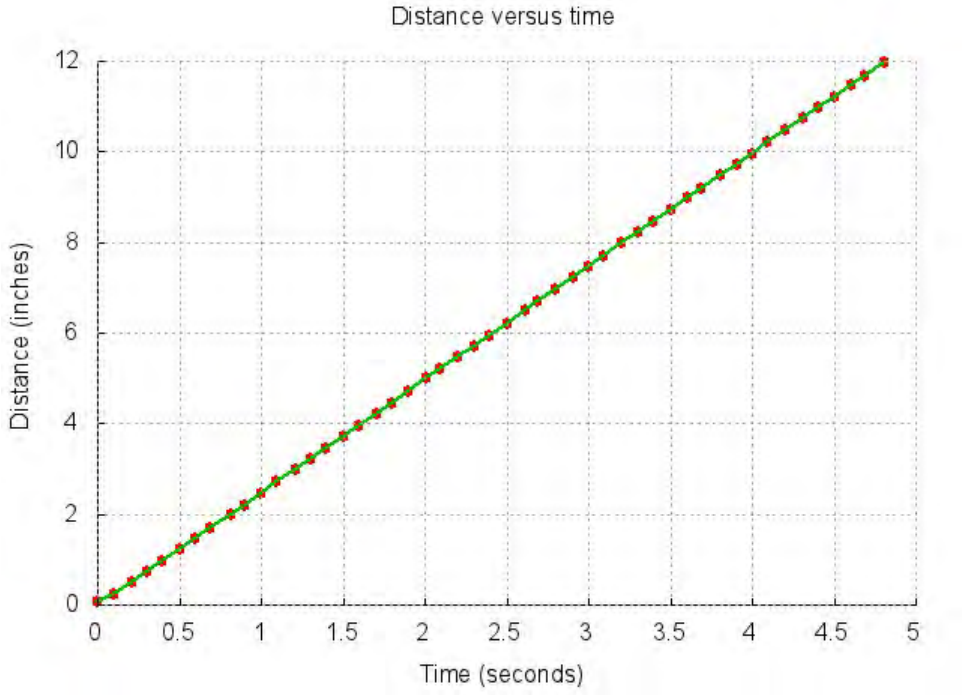


Figure 8.7: The scatter and line plots for the distance versus time from the program `recorddistancescatternline.ch`.

to drive the Linkbot-I 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 Linkbot-I 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.

$$d = -2.5t \tag{8.2}$$

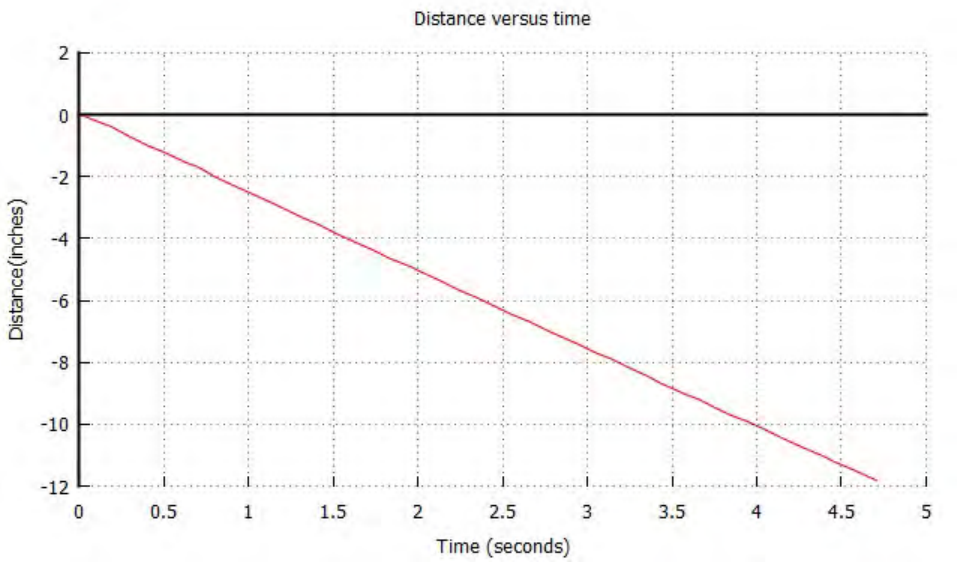


Figure 8.8: The plot for the distance versus time for a Linkbot-I moving in the negative direction from the program `recorddistanceneg.ch` with a specified distance.

E Do Exercise 4 on page 121.

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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Write a program `recorddistancenumline.ch` to drive the Linkbot 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 Linkbot 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 Linkbot Controller (CLC) 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 CLC.

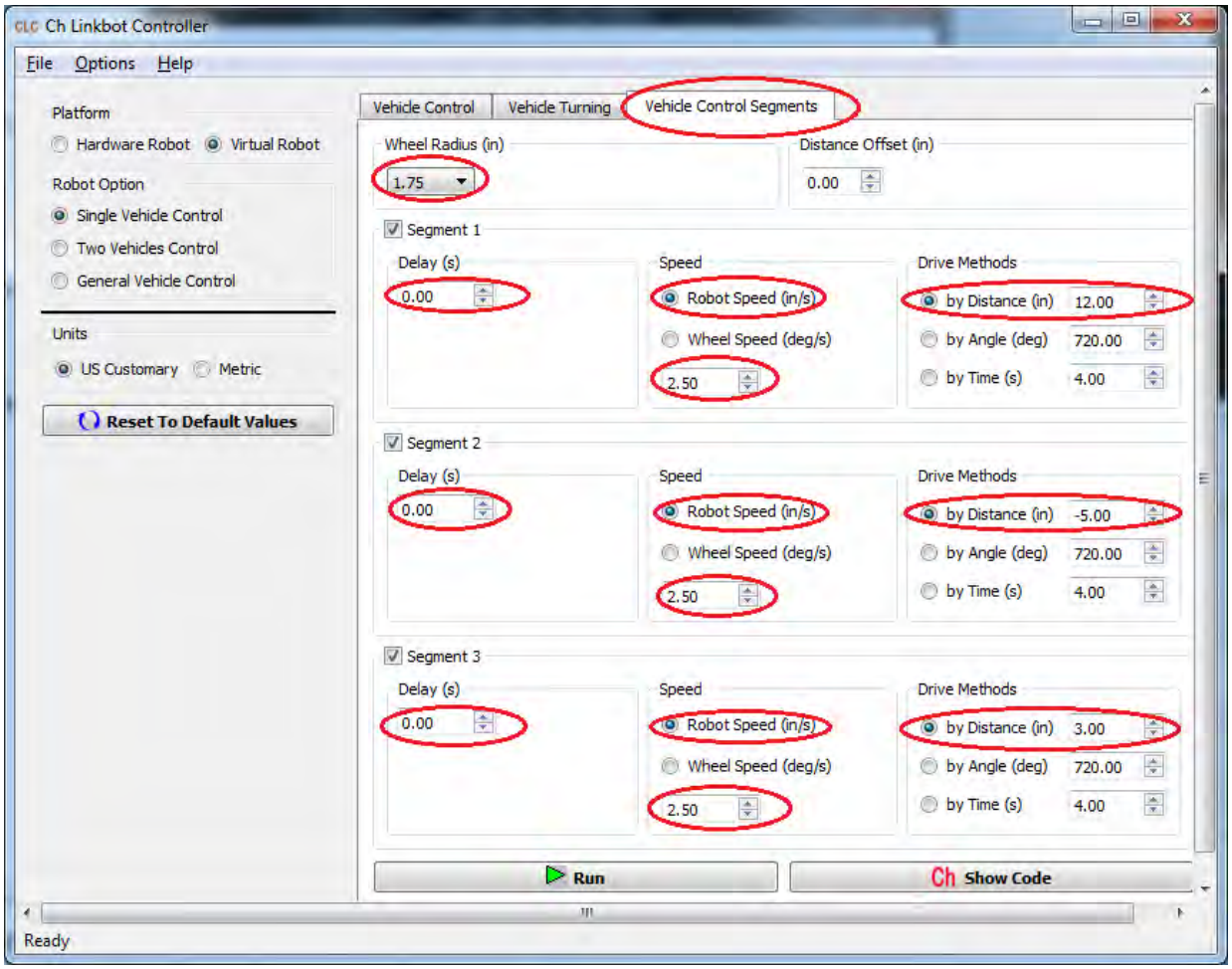


Figure 8.9: The setup of Ch Linkbot Controller for solving the problem.

E Do Exercise 2 on page 119.

```
/* File: recorddistancenumline.ch
   Plot robot distances in number line */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.75;       // 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.

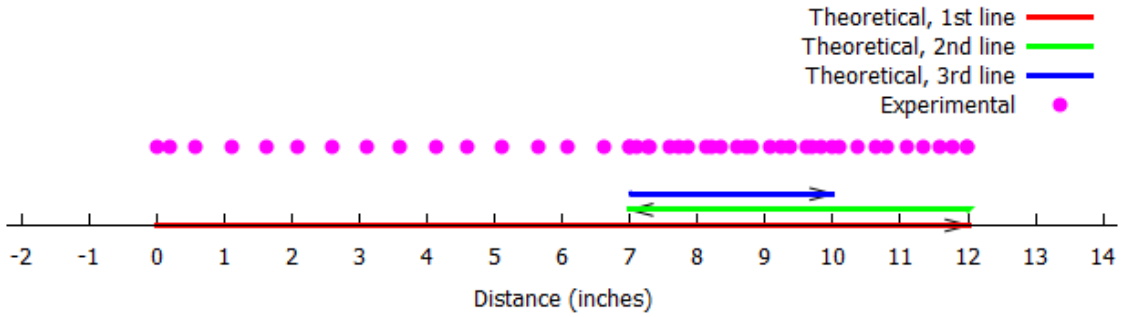


Figure 8.10: The number line for the robot distance, generated by Program 8.10.

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 Linkbot-I every 0.2 second. After the Linkbot stops moving, the statement

```
robot.recordDistanceEnd(numDataPoints);
```

stops recording data from the Linkbot.

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.numberLineScattern(distances, numDataPoints);
plot.legend("Experimental");
```

Do Exercise 5 on page 121.

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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. The robot is placed in an X-Y coordinate system at the coordinate (0, 4). Write a program `recorddistanceoffset.ch` to drive the Linkbot for 8 inches at the speed of 2.5 inches per second. Record the distance as the Linkbot moves with a time interval of 0.1 second. Plot the distance versus time.

We can solve this problem conveniently using Ch Linkbot Controller (CLC) with the setup shown in Figure 8.11.

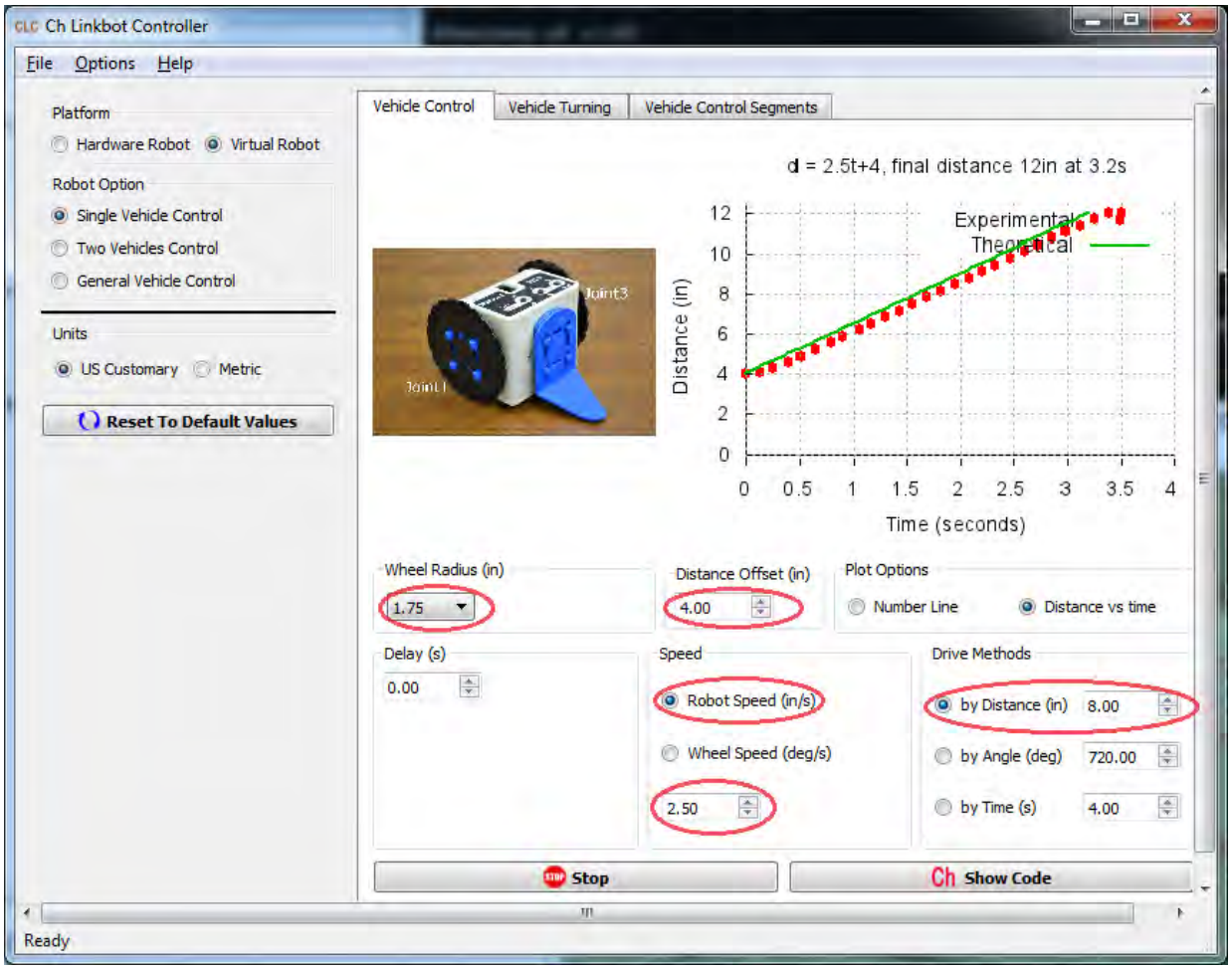


Figure 8.11: The setup of Ch Linkbot Controller for solving the problem.

```
/* File: recorddistanceoffset.ch
   Record time and distances with an initial offset distance by
       robot.recordDistanceOffset(offset);
   plot the acquired data */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.75;       // 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 1 */
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 Linkbot-I 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);
```

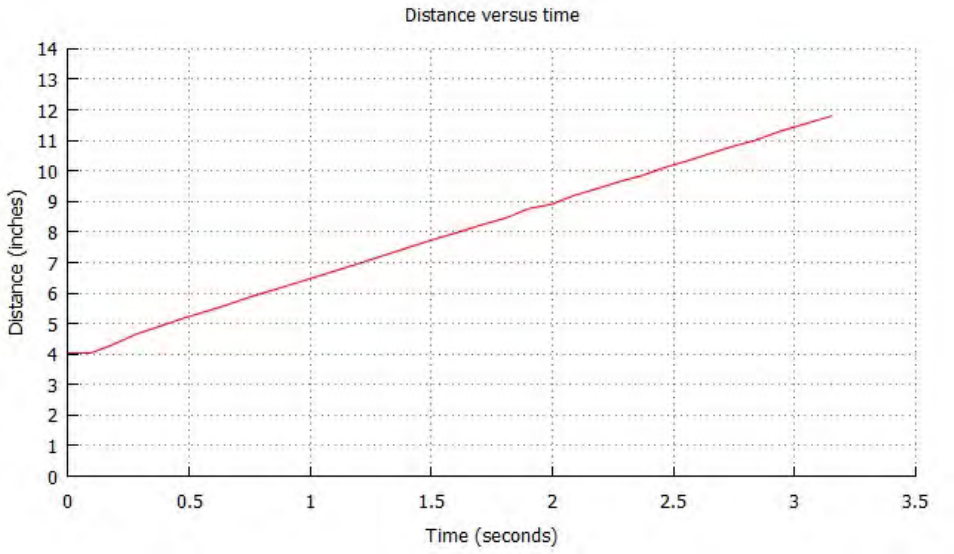


Figure 8.12: The plot for the distance versus time from Program 8.11.

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 121.

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 122.

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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 Linkbot 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 <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 2.5 inches/seconds
double radius = 1.75;       // 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(offset, distance1, distance2, distance3);
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 Linkbot-I every 0.2 second. After the Linkbot stops moving, the statement

```
robot.recordDistanceEnd(numDataPoints);
```

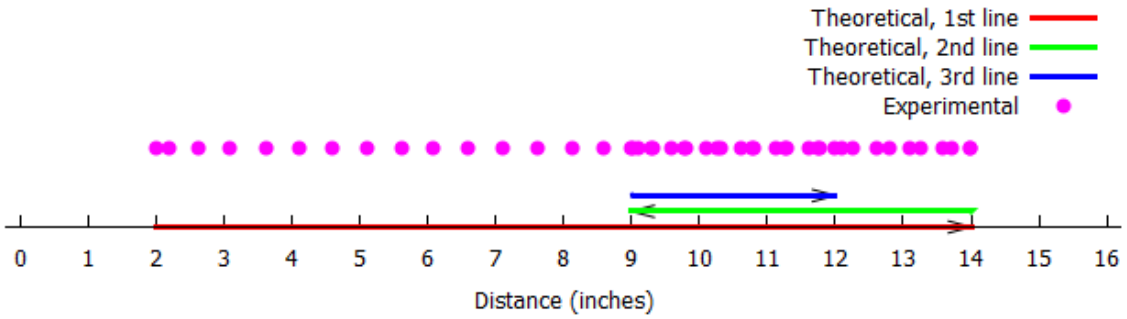


Figure 8.13: The number line for the robot distance with an offset, generated by Program 8.12.

stops recording data from the Linkbot.

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 122.

8.3.5 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.recordDistanceBegin(timedata, distances, radius, timeInterval);
   ```

   to start recording time and distance values of a Linkbot-I, for a specified wheel radius and a specified interval between distance readings.

2. Call the **CLinkbotI** member function

   ```
   robot.recordDistanceEnd(num);
   ```

   to stop recording time and distance values for a Linkbot-I.

3. Call the **CLinkbotI** 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 Linkbot when plotting data for multiple Linkbots.

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

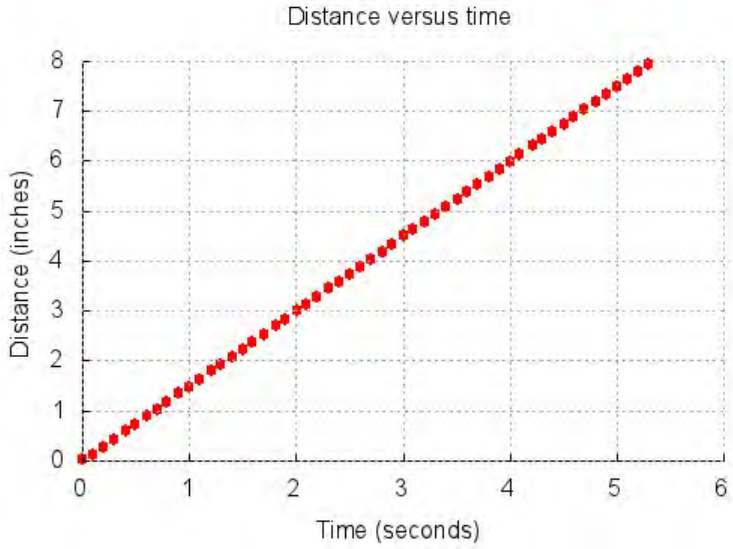
1. Watch the following video tutorials for Ch Linkbot Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Setup for Ch Linkbot Controller
   (b) Control a Robot with One Movement
   (c) Generating Ch Code from Ch Linkbot Controller
   (d) Control a Robot to Turn

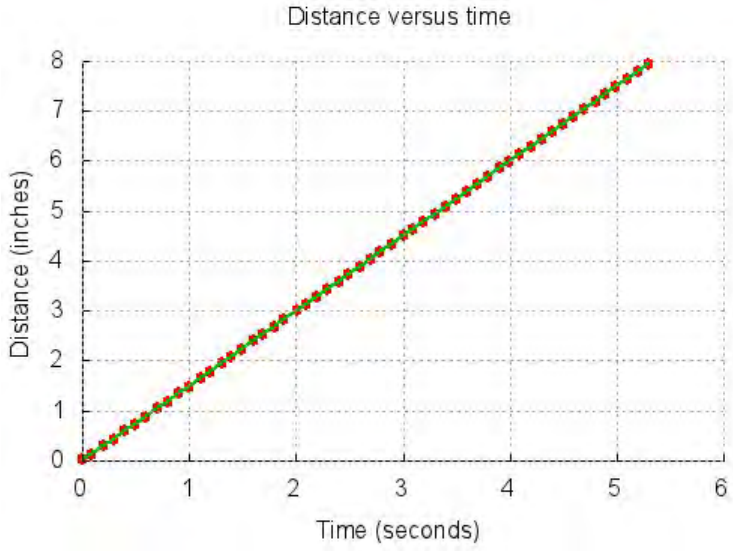
2. Watch the following video tutorial for Ch Linkbot Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Control a Robot with Multiple Movements

3. A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Write a program `recorddistancescattern2.ch` to drive the Linkbot drives 8 inches forward at the speed of 1.5 inches per second. Record the distance as the Linkbot 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?

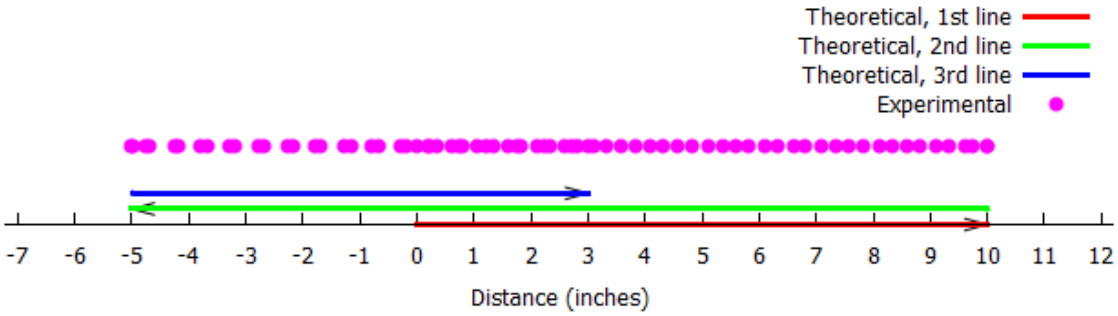

Distance versus time
Distance (inches)
Time (seconds)
0
1
2
3
4
5
6
7
8


Distance versus time
Distance (inches)
Time (seconds)
0
1
2
3
4
5
6
7
8

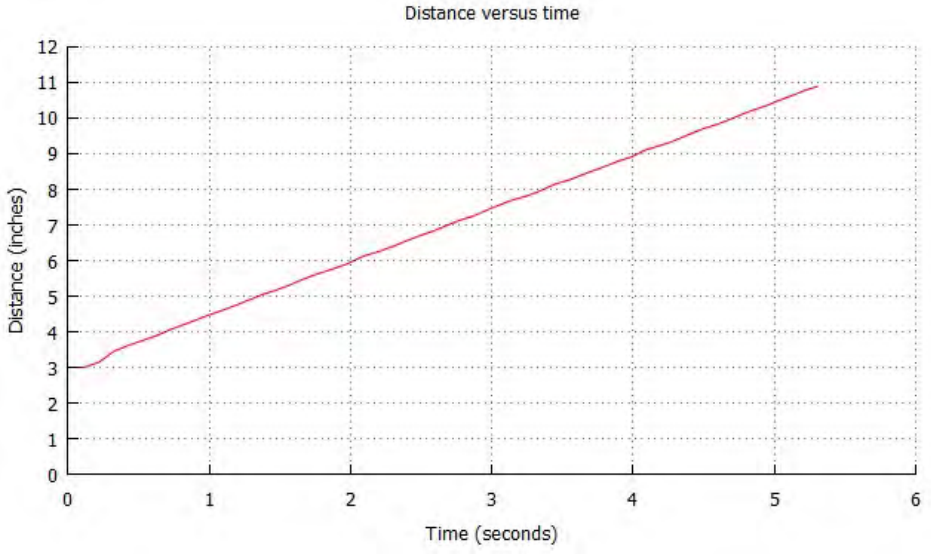


4. Modify the program `recorddistance2.ch` developed in Exercise 3 as the program `recorddistanceneg2.ch` to drive the Linkbot 8 inches *backward*, instead of forward, at the speed of 1.5 inches per second. Plot the distance versus time. What is the equation of motion for the robot?

5. Write a program `recorddistancenumline2.ch` to control a Linkbot-I 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 1.75 inches.



6. A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. The robot is placed in an X-Y coordinate system at the coordinate (0, 3). Write a program `recorddistanceoffset3.ch` to drive the Linkbot 8 inches forward at the speed of 1.5 inches per second. Record the distance as the Linkbot 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?



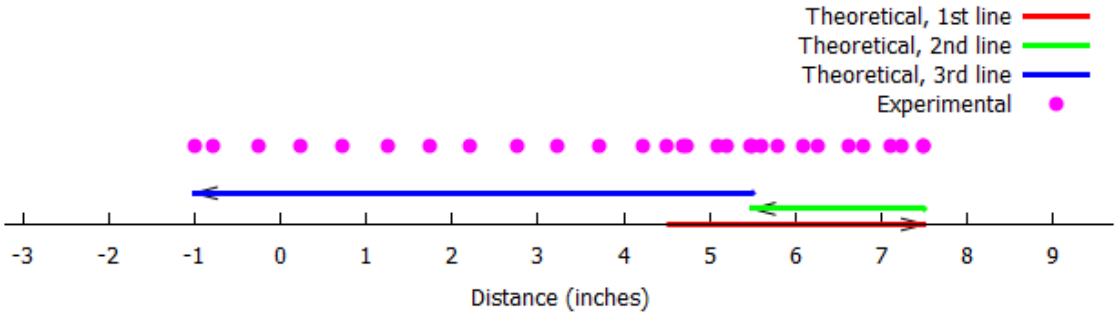
7. Modify the program `recorddistanceoffset3.ch` developed in Exercise 6 as the program `recorddistanceoffset4.ch` by replacing the statement

```
    robot.recordDistanceOffset(offset);
```

with the statement

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

8. Write a program `recorddistancenumlineoffset2.ch` to control a Linkbot-I 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 1.75 inches.



8.4 Move a Two-Wheel Robot with the Specified Time

8.4.1 Control a Linkbot-I 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 Linkbot-I with user input. In this section, we will see that we can also set the speed and time of a Linkbot-I's motion with user input.

Problem Statement:
A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>
CLinkbotI 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 Linkbot-I 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
1.75
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 Linkbot-I configured as a two-wheel robot with the radius of 1.75 inches for wheels, the above execution will move the Linkbot at the speed of 1 inch per second for 10 seconds.


Do Exercises 1 and 2 on page 129.

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 Linkbot 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 Linkbot-I when a distance is not specifically given in the program.

> **Problem Statement:**
> A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 <linkbot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in inches/second
double radius = 1.75;       // 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 Linkbot-I 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 Linkbot-I 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 129.

8.4.3 Plot Distances versus Time for a Two-Wheel Linkbot-I 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 Linkbot performing many different kinds of motions.

Problem Statement:
A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Write a program `recordspecifytime.ch` to drive

the Linkbot-I for 16 seconds at the speed of 2.5 inches per second. Record the distance as the Linkbot-I moves with a time interval of 0.1 second. Plot the distance versus time.

We can solve this problem using Ch Linkbot Controller (CLC) with the setup shown in Figure 8.14 to solve this problem conveniently.

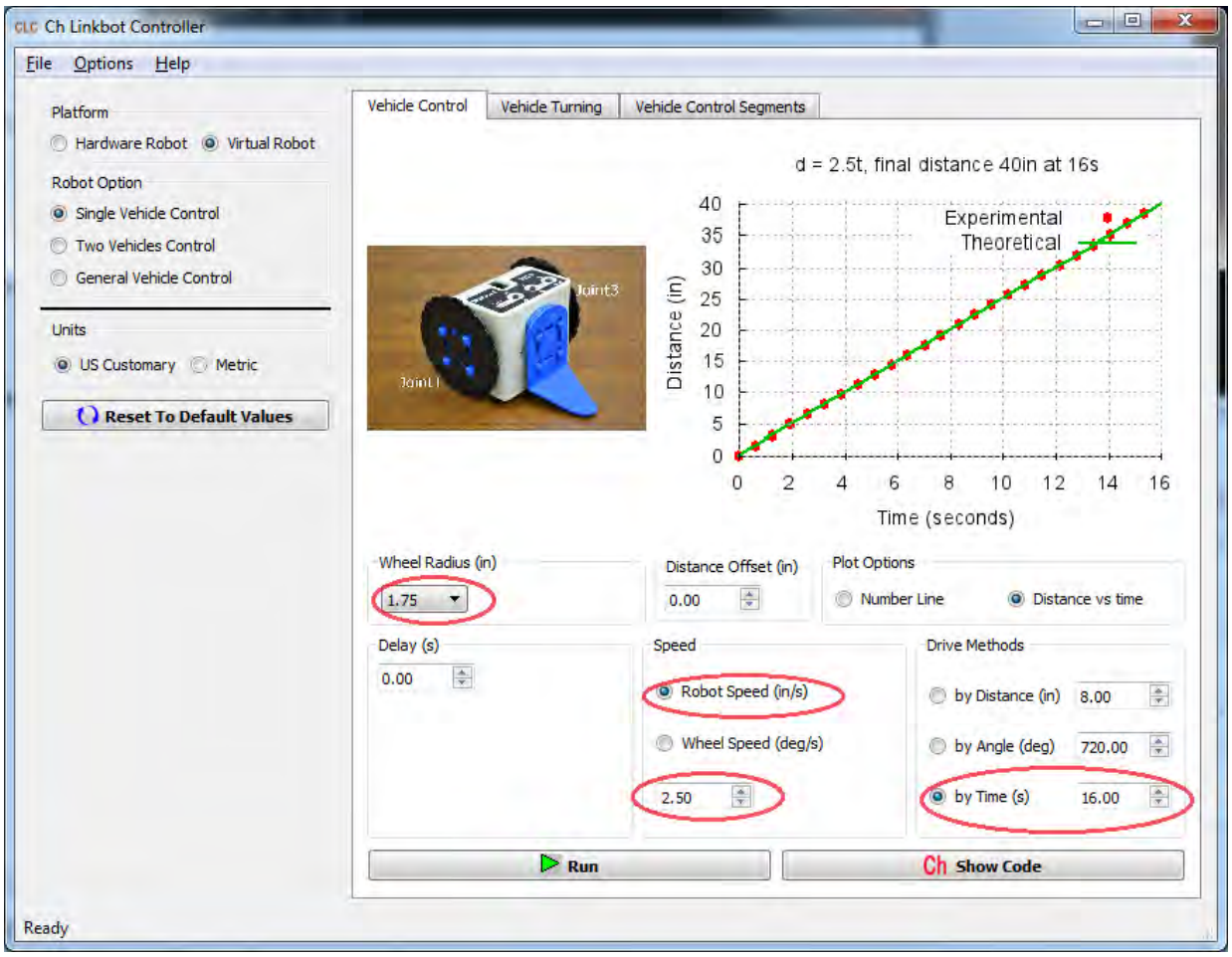


Figure 8.14: The setup of Ch Linkbot Controller for solving the problem.

```
/* File: recordspecifytime.ch
   Record time and distances, plot the acquired data */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 2.5;         // speed in 1.5 inches/seconds
double radius = 1.75;       // 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 Linkbot-I versus time for a two-wheel robot with a specified time.

In Program 8.9 we recorded data for a Linkbot 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 Linkbot-I 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 Linkbot-I 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 (t) in Figure 8.16 can be formulated by the following equation.

$$d = -2.5t \tag{8.5}$$

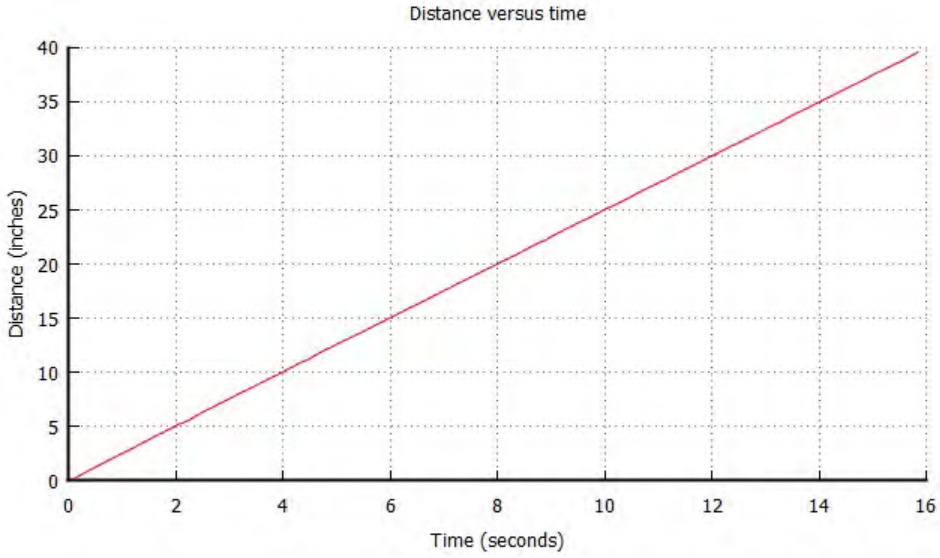


Figure 8.15: The plot for the distance versus time from Program 8.15.

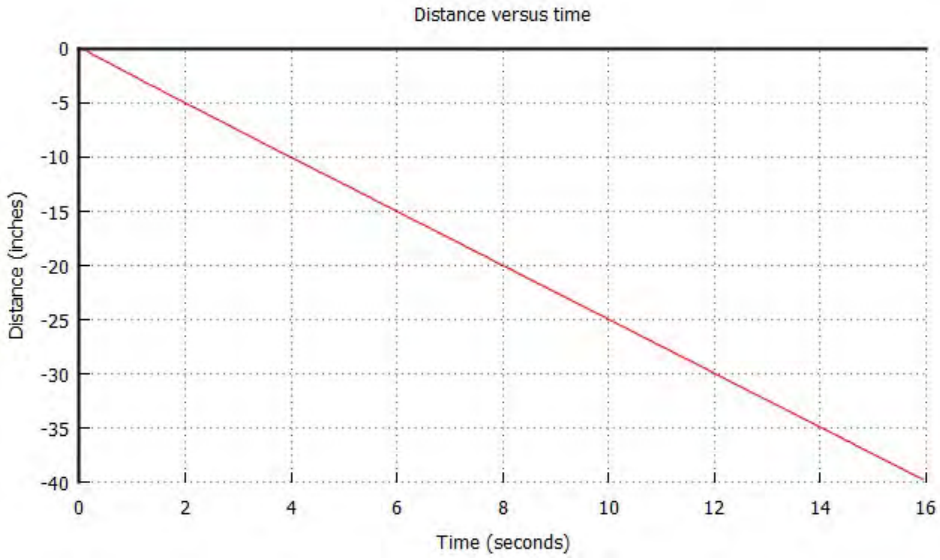


Figure 8.16: The plot for the distance versus time for a Linkbot-I moving in the negative direction from the program `recordspecifytimeneg.ch` with a specified time .

E Do Exercise 4 on page 129.

8.4.4 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.driveTime(time1);
   ```

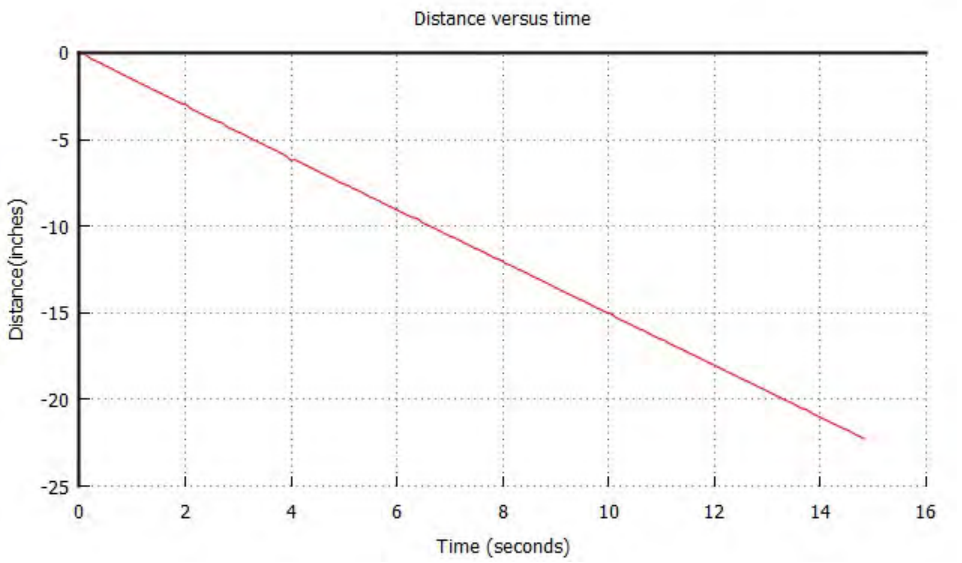
   to drive a Linkbot-I for a specified time ‘time1‘ rather than a specified distance.

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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Use the program `drivetime_p.ch` in Program 8.13 to make the Linkbot-I move 12 inches in 5 seconds (Note that the speed is defined as the distance divided by the time).
3. A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 1.75 inches. Write a program `recordspecifytime2.ch` to drive the Linkbot backward for 15 seconds at the speed of 1.5 inches per second by the member function **driveTime**(). Record the distance as the Linkbot 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 **CLinkbotI** 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 Linkbot-I 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 Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 4.445 centimeters. Write a program `recorddistancecm.ch` to drive the Linkbot for 20 centimeters at the speed of 6.5 centimeters per second. Record the distance as the Linkbot 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 <linkbot.h>
#include <chplot.h>
CLinkbotI 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 Linkbot-I 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.

$$d = 6.5t \tag{8.6}$$

E Do Exercises 1, 2, and 3 on page 132.

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.

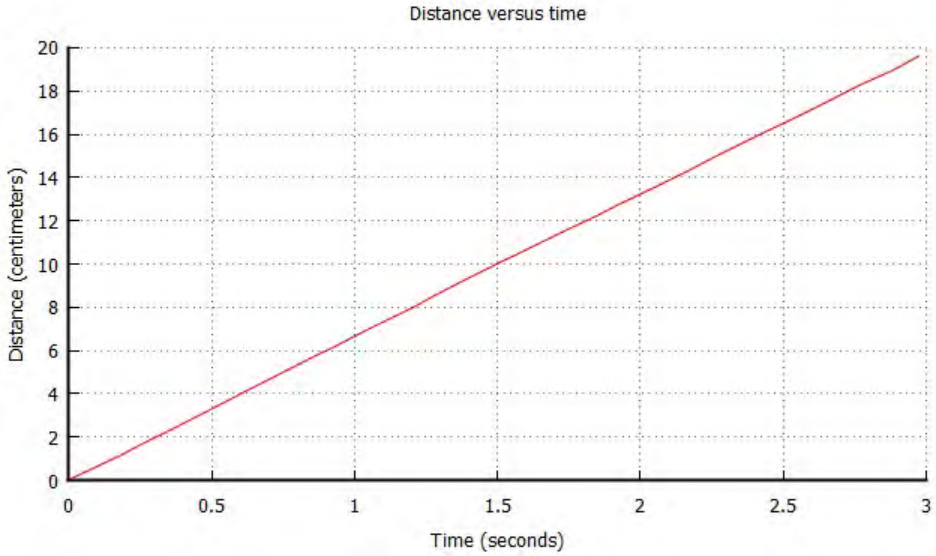


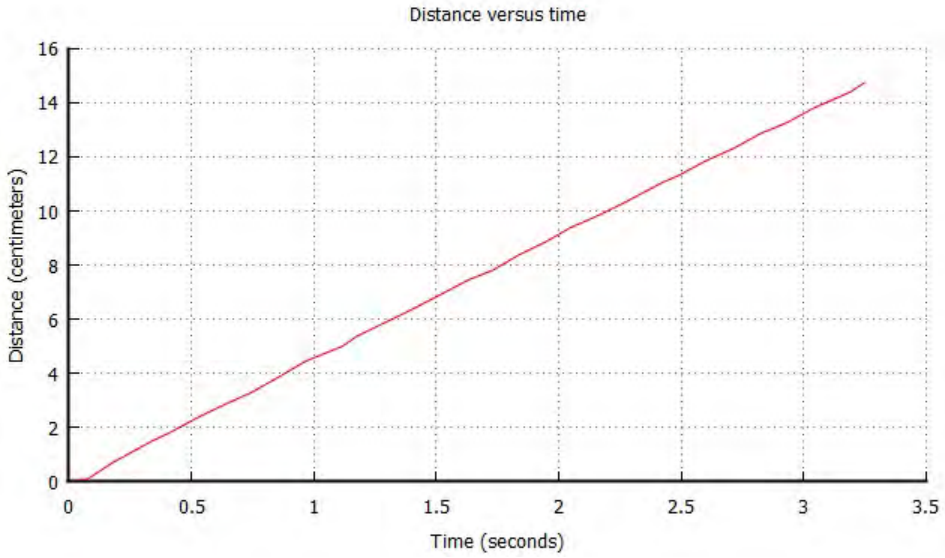
Figure 8.17: The plot for the distance versus time from Program 8.16.

8.5.2 Terminology

different length units, consistent length units.

8.5.3 Exercises

1. A Linkbot-I is configured as a two-wheel robot with wheels attached to joints 1 and 3. The radius of each wheel is 4.445 centimeters. Write a program `recorddistancecm2.ch` to drive the Linkbot 15 centimeters at the speed of 4.5 centimeters per second. Record the distance as the Linkbot-I 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 Linkbot-I configured as a two-wheel robot. Test your program by entering the radius of the wheel of your Linkbot, 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 Linkbot-I configured as a two-wheel robot. Test your program by entering the radius of the wheel of your Linkbot, speed of 5 centimeters per second, and time of 6 seconds.

CHAPTER 9

Moving a Single Robot in a Coordinate System

Ch programs can control robots in RoboSim with an x and y coordinate system as described in Chapter 4. Because of the convenience for defining the motion of robots in a coordinate system, member functions **drivexyTo**(), **getxy**(), **getPosition**(), **drivexyToFunc**(), **drivexyToExpr**(), **drivexy**(), **traceOn**(), **traceOff**(), **recordxyBegin**(), and **recordxyEnd**() are implemented to control a robot in RoboSim. These member functions are especially useful for learning mathematics concepts with RoboSim. They are available previously for controlling Linkbots in RoboSim only. They are now also available for use with the Linkbot Hardware. In this chapter, these member functions and their applications are described.

9.1 Move a Linkbot-I in a Coordinate System

9.1.1 Move a Linkbot-I in a Coordinate System

The member function **drivexyTo**() drives the Linkbot-I to the location (x, y) in the x and y coordinate system. The syntax of the member function **drivexyTo**() is as follows.

```
robot.drivexyTo(x, y, radius, trackwidth);
```

The first two arguments specify the location (x, y) to which the robot will drive. The third argument is the radius of the two wheels. The fourth argument is the track width as shown in Figure 5.1 on page 63. Like the arguments for the member functions **turnLeft**() and **turnRight**(), the units for both radius and track width must be the same. They can be inches, feet, centimeters, meters, etc.

The member function **getxy**() can be used to obtain the position of a robot in the x and y coordinate system. The syntax of the member function **getxy**() is as follows.

```
robot.getxy(x, y);
```

The position (x, y) of the robot specified in the x and y coordinate system is passed through its two arguments x and y. The information obtained by this function is the same as the position information obtained by clicking on a Linkbot in a RoboSim scene. But when **getxy**() is called this information is stored internally, and can be used or analyzed later in the program.

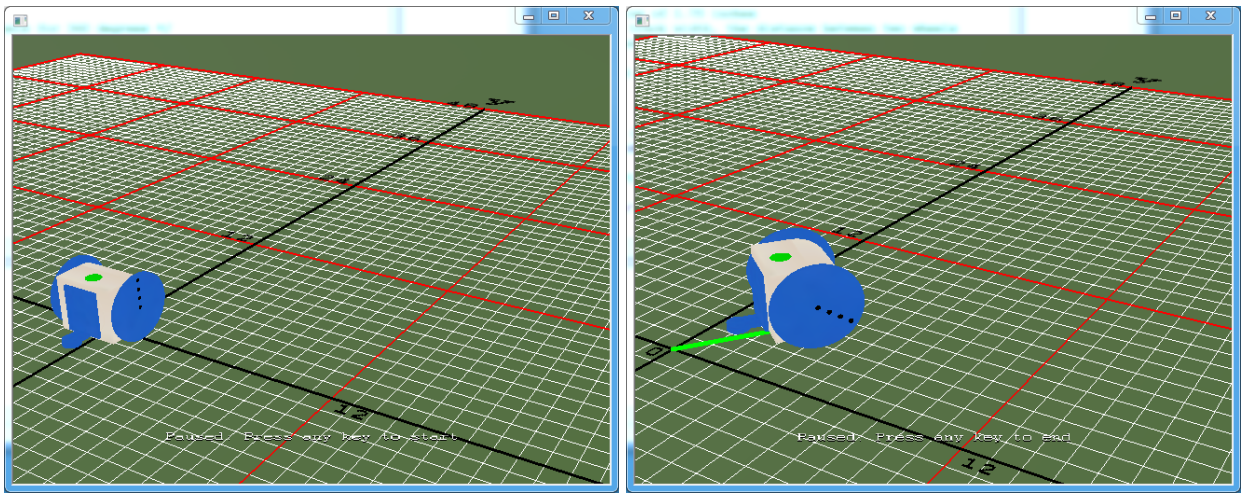
Program 9.1 demonstrates how to use **drivexyTo**() to draw a graph of a straight line in RoboSim as shown in Figure 9.1b, and how to use **getxy**() to get the position of the Linkbot-I after this line is drawn. Before running program 9.1 configure the RoboSim GUI as is shown in Figure 4.1 from Section 4.1, so that the starting position of the Linkbot-I is (0, 0) with the orientiation angle 90 degrees as shown in 9.1a.

```
/* File: drivexyto.ch
   Note: This program uses drivexyTo() available in RoboSim only
         to move a Linkbot-I from (0, 0) to (3, 4).
         Use getxy() to get the x and y coordinates of the robot.
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for robot. */
#include <linkbot.h>   /* for CLinkbotI */
CLinkbotI robot;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;   // the track width, the distance between two wheels
double x, y;                // x and y coordinates

/* move the robot to the position (3, 4) */
robot.drivexyTo(3, 4, radius, trackwidth);

/* get the position of the robot */
robot.getxy(x, y);
printf("getxy(x, y) = (%lf, %lf)\n", x, y);
```

Program 9.1: Moving a Virtual Linkbot-I to a specified location (x, y) using **drivexyTo**().



(a) Initial position (b) Final position

Figure 9.1: The RoboSim scene with the robot trajectory from Program 9.1.

Program 9.1 uses the **CLinkbotI** class member function **drivexyTo**() to drive a Linkbot-I in a straight

line from (0, 0) to (3, 4) in RoboSim. The lines

```
double radius = 1.75; // radius of 1.75 inches
double trackwidth = 3.69; // the track width, the distance between two wheels
double x, y; // x and y coordinates
```

declare and initialize the variables that are needed in order to use **drivexyTo**() and **getxy**(). A variable `radius` of **double** type is declared and assigned a value of 1.75, which is desired wheel radius. Another variable `trackwidth`, also of **double** type, is declared and assigned a value of 3.69, which is the distance between the two wheels of the Linkbot-I. The variables `x` and `y` of **double** type will be used to store the x and y coordinate values that are passed from the arguments of the member function **getxy**(). After the Linkbot-I is connected and set to the zero position, the line

```
robot.drivexyTo(3, 4, radius, trackwidth);
```

uses the **CLinkbotI** member function **drivexyTo**() to drive the virtual Linkbot-I from point (0, 0) to point (3, 4). Pressing the 't' key shows a straight green line between these two points, which looks the same as in Figure 9.1. The final two program statements

```
robot.getxy(x, y);
printf("getxy(x, y) = (%lf, %lf)\n", x, y);
```

uses the **CLinkbotI** member function **getxy**() to retrieve the final x and y coordinates of the Linkbot-I and then prints these coordinates to the ChIDE console using **printf**(). The input/output pane will display the following line

```
getxy(x, y) = (2.993417, 3.997225)
```

Program 9.1 demonstrates the efficiency of the member function **drivexyTo**(). In this case, the function call

```
robot.drivexyTo(x, y, radius, trackwidth);
```

is somehow equivalent to the two statements

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

Using **drivexyTo**() in this case eliminates the need to calculate the angle to turn and the distance to drive. Instead, these are calculated internally by the function **drivexyTo**().

You can also use **getxy**() to obtain the position of a robot after it is driven by **driveAngle**().

E Do Exercises 1 and 2 on 139.

9.1.2 Move a Hardware Linkbot-I Not Starting in its Default Initial Position

The position of a robot is defined by the x and y coordinates for the center of the robot, and its orientation angle. A robot can be added in RoboSim GUI from Individual Robots as shown in Figures 4.1, 4.5, and 4.6. Figure 4.6 shows its x and y coordinates, and orientation angle. The default position for each robot is listed in Table 9.1. The default value for the x coordinate for the first robot is 0. Each subsequent one is 6 inches apart. The default value for the y coordinate for each robot has the same value of 0. The default value for the orientation angle for each robot has the same value of 90 degrees. These initial x and y coordinates, and orientation angle can also be changed before a robot program is executed. When the program is executed, this initial positon information are available internally for tracking the virtual robot.

Table 9.1: The default initial position (x, y, angle) for robots.

Robot #	x (inches)	y (inches)	angle (degrees)
1	0	0	90
2	6	0	90
3	12	0	90
4	18	0	90
5	24	0	90

For controlling a hardware robot, if a robot is not placed in its default position as listed in Table 9.1, the user needs to provide the robot with the initial position by the member function **initPosition**(). The syntax of the member function **initPosition**() is as follows.

```
    robot.initPosition(x, y, angle);
```

The position of the robot specified in the x and y coordinates, and its orientation is passed through its three arguments `x`, `y`, and `angle`.

Program 9.2 can be used to control both hardware and virtual robots. Comparing Program 9.1, the statement

```
    robot.initPosition(0, 0, 90);
```

sets the initial position for x and y coordinates at (0, 0) and orientation angle of 90 degrees as shown in Figure 9.1a.

```
/* File: initposition.ch
   Note: This program uses drivexyTo()
         to move a hardware Linkbot-I from (0, 0) to (3, 4).
         Use getPosition() to get the x and y coordinates of the robot,
         and orientation angle.
   Set the initial position (x, y, angle) to (0, 0, 90) for the robot. */
#include <linkbot.h>   /* for CLinkbotI */
CLinkbotI robot;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y;               // x and y coordinates
double angle;              // orientaion angle

/* set the initial position (x, y, angle) to (0, 0, 90) */
robot.initPosition(0, 0, 90);

/* move the robot to the position (3, 4) */
robot.drivexyTo(3, 4, radius, trackwidth);

/* get the position of the robot */
robot.getPosition(x, y, angle);
printf("getPositin(x, y, angle) = (%lf, %lf, %lf)\n", x, y, angle);
```

Program 9.2: Moving a Hardware Linkbot-I to a specified location (x, y) using **drivexyTo**().

If a robot is not placed in the default position as listed in Table 9.1 for controlling a hardware robot, the member function **initPosition**() must be called if any one of the member functions **drivexyTo**(), **getxy**(), **getPosition**(), **drivexyToFunc**(), **drivexyToExpr**(), **drivexyToNB**(), **drivexyWait**(), **drivexy**(), **drivexyNB**(),

recordxyBegin(), and **recordxyEnd**() is used inside the program. In the remaning of this book, we will add **initPosition**() to the program if a robot is not placed in its default position and one of the above member functions is used. Then, the program can control both virtual robots in RoboSim and hardware robots.

The member function **getPosition**() can be used to obtain the position of a robot in term of its x and y coordinates, and orientation angle. The syntax of the member function **getPosition**() is as follows.

```
robot.getPosition(x, y, angle);
```

The position (x, y, angle) of the robot specified in the x and y coordinate system is passed through its three arguments x, y, and angle. The information obtained by this function is the same as the position information obtained by clicking on a Linkbot in a RoboSim scene. But when **getPosition**() is called this information is stored internally, and can be used or analyzed later in the program.

Program 9.2 demonstrates how to use the function calls

```
/* get the position of the robot */
robot.getPosition(x, y, angle);
printf("getPositin(x, y, angle) = (%lf, %lf, %lf)\n", x, y, angle);
```

to get and display the position of the Linkbot-I after it moves to the position (3, 4).

Do Exercises 3 and 4 on 139.

9.1.3 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.drivexyTo(x, y, radius, trackwidth);
   ```

   to drive a Linkbot-I from one point to another in an x and y coordinate system.
2. Call the **CLinkbotI** member function

   ```
   robot.getxy(x, y);
   ```

   to obtain the position of a Linkbot-I in an x and y coordinate system.
3. Call the **CLinkbotI** member function

   ```
   robot.initPosition(x, y, angle);
   ```

   to set the initial position of a Linkbot-I in an x and y coordinate system, and its orientation angle.
4. Call the **CLinkbotI** member function

   ```
   robot.getPosition(x, y, angle);
   ```

   to obtain the position of a Linkbot-I in an x and y coordinate system, and its orientation angle.

9.1.4 Terminology

robot.drivexyTo(), **robot.getxy**(), **robot.getPosition**(), and **robot.initPosition**().

9.1.5 Exercises

1. Watch the following video tutorials for RoboBlockly in http://roboblockly.ucdavis.edu/videos/.
   (a) R6. Drive to the Location (x, y): drivexyTo()
   (b) R11. Get the Location (x, y) and Print Variable: getxy()

2. Write a program `drivexyto3.ch`, similar to Program 9.1, to drive a Linkbot-I from (0, 0) to (0, 8). Run this program to simulate the motion of the robot in RoboSim.

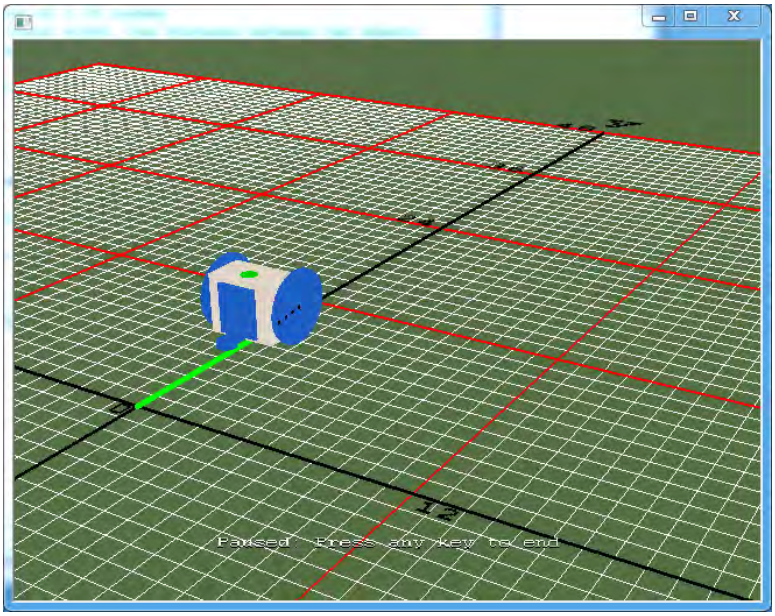


Figure 9.2: Moving a virtual Linkbot-I along the y-axis from (0, 0) to (0, 8).

3. Write a program `drivexyto4.ch` to drive a Linkbot-I from (0, 0) to (6, 0). Run this program to simulate the motion of the robot in RoboSim.

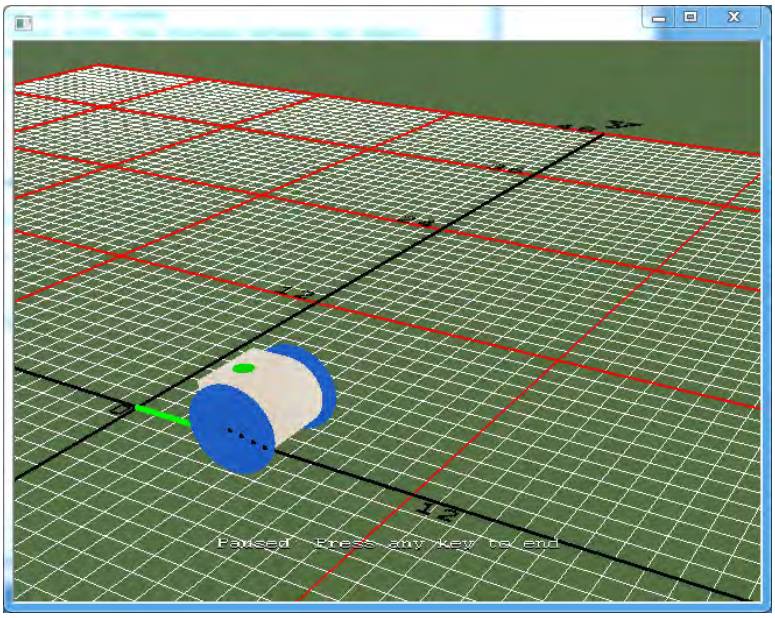


Figure 9.3: Moving a virtual Linkbot-I along the x-axis from (0, 0) to (6, 0).

4. Write a program `drivexyto5.ch` to drive a virtual Linkbot-I in RoboSim from (0, 0) to (6, 8).

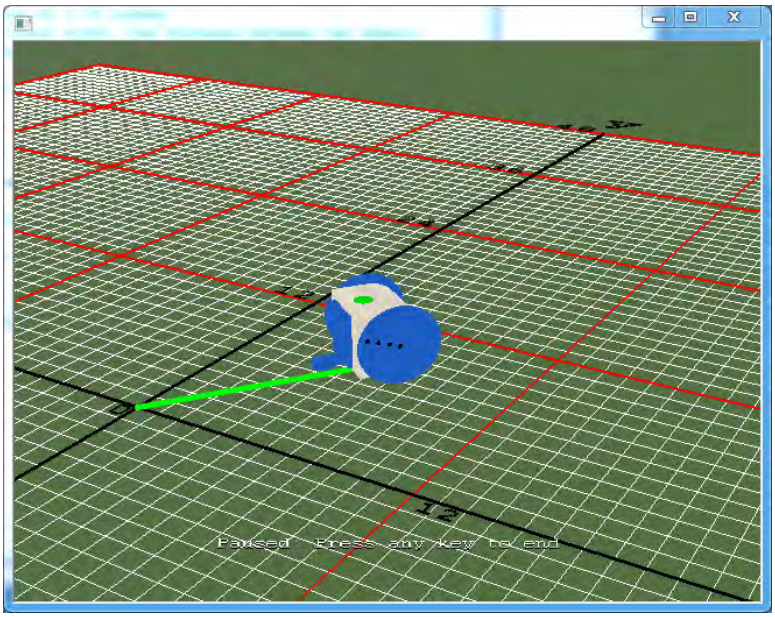


Figure 9.4: Moving a virtual Linkbot-I from (0, 0) to (6, 8).

9.2 Plot Positions of a Robot

The member function **drivexyTo**() can be used multiple times in a program to drive a virtual Linkbot-I to different points in a coordinate system, drawing a graph along the way. Program 9.3 demonstrates this use of **drivexyTo**(), and also uses CPlot member functions to plot the specified points in a separate

window. Member functions **title**(), and **label**() of the **CPlot** class are introduced in section 8.2 in Chapter 11. Comparing a plot, generated using the **CPlot** member functions, to the trajectory of a virtual Linkbot shows the accuracy of the RoboSim GUI's graphing capabilities. Set the initial position of the Linkbot-I in RoboSim GUI to (0, 0) before running Program 9.3. To see the full trajectory of the Linkbot's motion in the RoboSim GUI, hold the right mouse button and drag to zoom out and then hold both mouse buttons and drag to pull the Linkbot-I to the left side of the RoboSim GUI window.

```
/* File: points.ch
   Note: This program uses drivexyTo() available in RoboSim only
         to move one Linkbot-I to multiple points.
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for the robot.
   A robot moves to specified points  */
#include <chplot.h>    /* for CPlot */
#include <linkbot.h>   /* for CLinkbotI */
CPlot plot;
CLinkbotI robot;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y;               // x and y coordinates

/* move the robot to specified points */
robot.drivexyTo(12, 5.5, radius, trackwidth);
robot.drivexyTo(24, 30, radius, trackwidth);
robot.drivexyTo(36, 10, radius, trackwidth);
robot.drivexyTo(48, 5, radius, trackwidth);

/* plot the points */
plot.title("");
plot.label(PLOT_AXIS_X, "x (in)");
plot.label(PLOT_AXIS_Y, "y (in)");
plot.point(0, 0);
plot.point(12, 5.5);
plot.point(24, 30);
plot.point(36, 10);
plot.point(48, 5);
plot.plotting();
```

Program 9.3: Moving a Linkbot-I to multiple specified locations using **drivexyTo**().

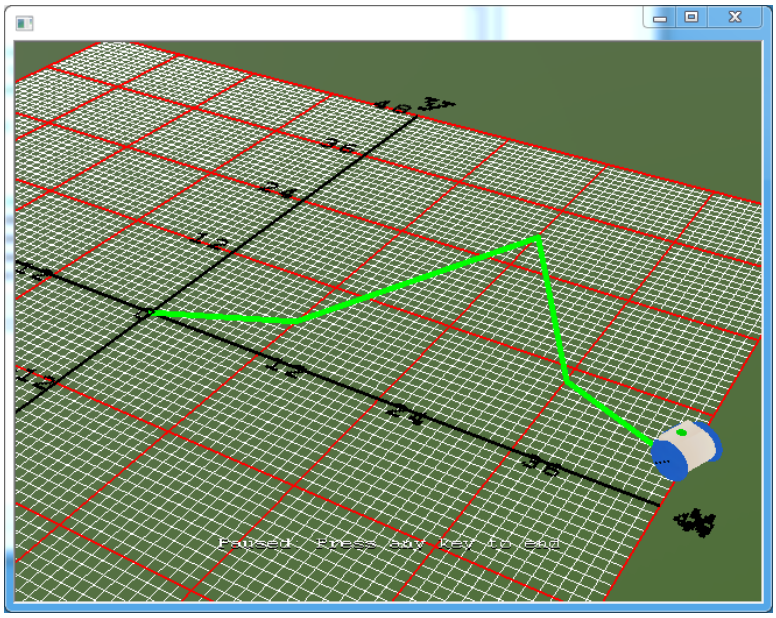


Figure 9.5: The RoboSim scene with the robot trajectory from Program 9.3.

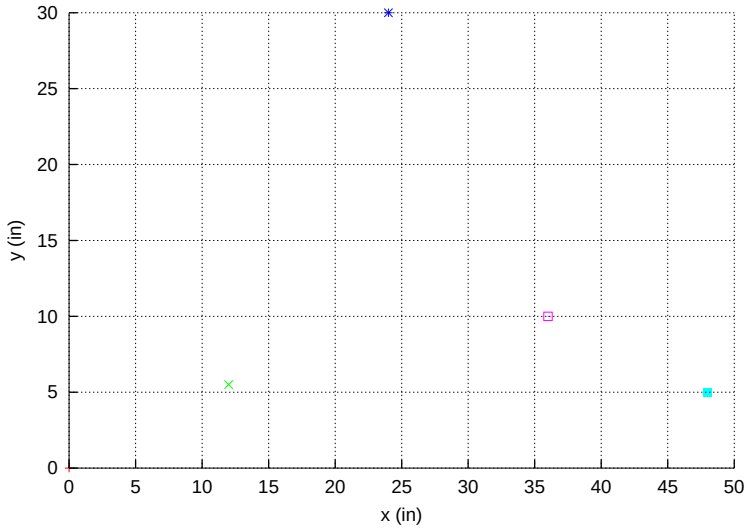


Figure 9.6: The plot for the locations traveled by the robot from Program 9.3.

Program 9.3 drives a virtual Linkbot-I to four different points in a coordinate system. The line

```
robot.drivexyTo(12, 5.5, radius, trackwidth);
```

drives the virtual Linkbot-I from (0, 0) to (12, 5.5) in the coordinate system. The next line

```
robot.drivexyTo(24, 30, radius, trackwidth);
```

drives the virtual Linkbot-I from (12, 5.5) to (24, 30). Notice that the member function **drivexyTo**() drives a virtual Linkbot to an absolute position in an x and y coordinate system. That is, the Linkbot does not need to start at (0, 0) in order to get to the point (24, 30) specified in the second **drivexyTo**() function call. Instead, the program internally calculates how to drive in a straight line from the virtual Linkbot's current location to the absolute location specified by the arguments `x` and `y` passed to **drivexyTo**(). Similarly, the lines

```
robot.drivexyTo(36, 10, radius, trackwidth);
robot.drivexyTo(48, 5, radius, trackwidth);
```

drive the virtual Linkbot-I from (24, 30) to (36, 10) and then from (36, 10) to (48, 5) in the coordinate system. The final trajectory of the Linkbot-I is shown in Figure 9.5. To plot the individual start and end points along the Linkbot-I's trajectory, the **CPlot** member function **point()** is used. The general syntax of this function is

```
plot.point(x, y);
```

where `x` and `y` are the coordinates of the point to be plotted. The lines

```
plot.point(0, 0);
plot.point(12, 5.5);
plot.point(24, 30);
plot.point(36, 10);
plot.point(48, 5);
```

plot each of the individual points along the Linkbot-I's trajectory. The resulting graph is shown in Figure 9.6.

E Do Exercises 1 and 3(a) on page 147.

A virtual Linkbot-I can also be used to draw geometric shapes. Program 9.4 drives a Linkbot-I along a trajectory that forms a right triangle using the member function **drivexyTo**(). A plot of the Linkbot's triangular trajectory is also generated using **CPlot** member functions. Before running Program 9.4, set the initial position of the Linkbot-I in the RoboSim GUI to (0, 0).

```
/* File: righttriangle.ch
   Note: This program uses drivexyTo() available in RoboSim only
         to move one Linkbot-I from (0, 0) to (3, 0), (3, 4), then to (0, 0).
   Set the initial position (x, y) in RoboSim to (0, 0) for the robot. */
#include <chplot.h>    /* for CPlot */
#include <linkbot.h>   /* for CLinkbotI */
CPlot plot;
CLinkbotI robot;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y;               // x and y coordinates

/* move the robot to specified points */
robot.drivexyTo(3, 0, radius, trackwidth);
robot.getxy(x, y);
printf("getxy(x, y) = (%lf, %lf)\n", x, y);

robot.drivexyTo(3, 4, radius, trackwidth);
robot.getxy(x, y);
printf("getxy(x, y) = (%lf, %lf)\n", x, y);

robot.drivexyTo(0, 0, radius, trackwidth);
robot.getxy(x, y);
printf("getxy(x, y) = (%lf, %lf)\n", x, y);

/* the distance between points (0, 0) and (3, 4) */
printf("distance = %lf\n", sqrt(3*3 + 4*4));


/* plot the points */
plot.title("");
plot.label(PLOT_AXIS_X, "x (in)");
plot.label(PLOT_AXIS_Y, "y (in)");
plot.line(0, 0, 3, 0);
plot.line(3, 0, 3, 4);
plot.line(3, 4, 0, 0);
plot.axisRange(PLOT_AXIS_X, 0, 5); // set the x-axis range
plot.axisRange(PLOT_AXIS_Y, 0, 5); // set the y-axis range
plot.sizeRatio(1);
plot.plotting();
```

Program 9.4: Moving a Linkbot-I to follow a right triangle using **drivexyTo**().

Program 9.4 drives a virtual Linkbot-I along the trajectory of a right triangle, prints each of the triangle's three points, calculates the length of the hypotenuse, and then plots the Linkbot's triangular trajectory. The lines

```
    robot.drivexyTo(3, 0, radius, trackwidth);
    robot.getxy(x, y);
    printf("getxy(x, y) = (%lf, %lf)\n", x, y);
```

drive the virtual Linkbot-I from (0, 0) to (3, 0), obtain the x and y coordinates of the virtual Linkbot after the movement, and print these coordinates to the input/output pane. This printed point represents the first of the right triangle's three points. These actions are repeated by the program to drive the virtual Linkbot-I from (3, 0) to (3, 4), and then from (3, 4) to (0, 0), printing the coordinates of the right triangle's second and third points along the way.

In Program 9.4, the Pythagorean Theorem is used to calculate the hypotenuse of the right triangle. For

the right triangle shown in Figure 9.7, c is the length of the hypotenuse. a and b are the lengths of the two legs. Based on the Pythagorean theorem, they are related by the formula

$$c^2 = a^2 + b^2$$

Given the lengths of the two legs, the length of the hypotenuse for a right triangle can be calculated by the formula

$$c = \sqrt{a^2 + b^2}$$

In Ch, the function **sqrt**() is used to calculate square roots. The general syntax of the **sqrt()** function is

```
sqrt(x);
```

where `x` is a number or expression that evaluates as a **double** data type. In Program 9.4, the length of the hypotenuse is represented by the distance between points (0, 0) and (3, 4). Given that the length of leg a is represented by the distance between (0, 0) and (3, 0), which is 3, and the length of leg b is represented by the distance between (0, 0) and (0, 4), which is 4, the line

```
printf("distance = %lf\n", sqrt(3*3 + 4*4));
```

calculates the hypotenuse of the triangle drawn by the virtual Linkbot-I and then prints it to the input/output pane. After this statement, the input/output pane will display the following lines

```
getxy(x, y) = (3.000506, -0.004528)
getxy(x, y) = (3.004118, 3.994140)
getxy(x, y) = (-0.001259, 0.004295)
distance = 5.000000
```

The **CPlot** member functions **line()** and **sizeRatio()** are used to generate the plot of the virtual Linkbot-I's trajectory. The syntax of the member function **line()** is

```
plot.line(x1, y1, x2, y2);
```

This function draws a straight line between the two points (x_1, y_1) and (x_2, y_2). The lines

```
plot.line(0, 0, 3, 0);
plot.line(3, 0, 3, 4);
plot.line(3, 4, 0, 0);
```

plot the three sides of the right triangle, from point (0, 0), to point (3, 0), back to point (3, 4).

A plot is typically displayed in a rectangular area with the proper scale. The ***aspect ratio*** of a plot is the ratio of the length of the y-axis to the length of the x-axis. It can be set by the **CPlot** member function **sizeRatio()**, which has the following general syntax

```
plot.sizeRatio(ratio);
```

where the argument `ratio` specifies the aspect ratio. The line

```
plot.sizeRatio(1);
```

displays a plot in a square box where the ranges for x and y are the same. The plot generated by Program 9.4 is shown in Figure 9.9. Toggle the 'r' key in the RoboSim GUI to get a clear view of the right triangle traced by the movement of the virtual Linkbot-I, as Figure 9.8 shows.

E Do Exercise 2 on page 148.

Unlike the member function **drivexyTo**() moving to the absolute location (x, y) in the x and y coordinate system, the member function **drivexy**() drive the Linkbot-I by x and y relative to its current position in the x and y coordinate system. The syntax of the member function **drivexy**() is as follows.

```
robot.drivexy(x, y, radius, trackwidth);
```

Programs `drivexyto2.ch` and `drivexy2.ch`, distributed along with other sample programs, demonstrate the differences between the member functions **drivexyTo**() and **drivexy**().

E Do Exercise 3(b) on page 148.

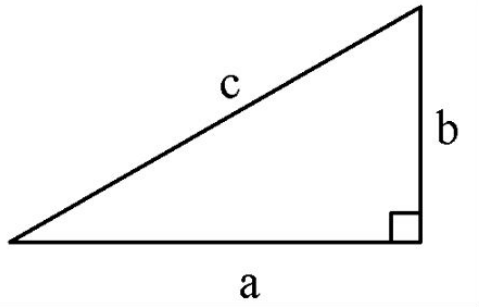


Figure 9.7: A right triangle with length *c* for the hypotenuse, and lengths *a* and *b* for the other two legs.

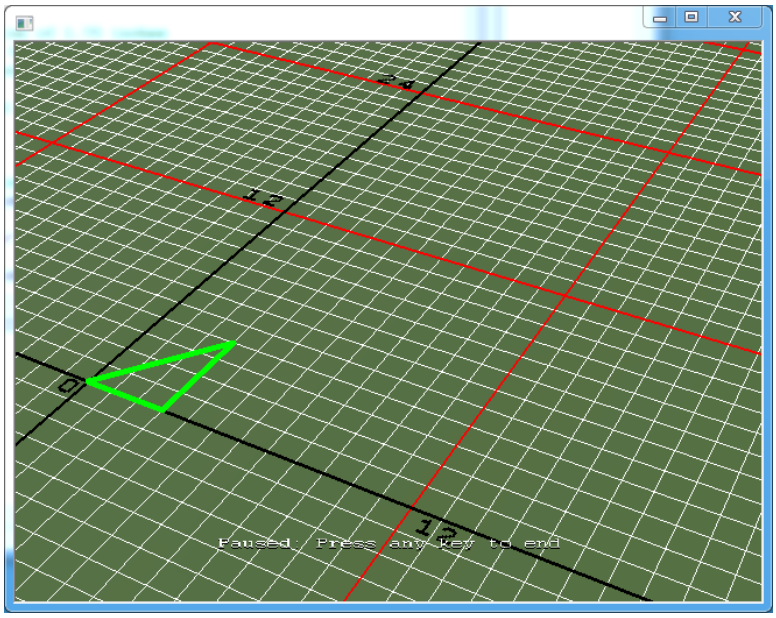


Figure 9.8: The RoboSim scene with the robot trajectory of a right triangle from Program 9.4.

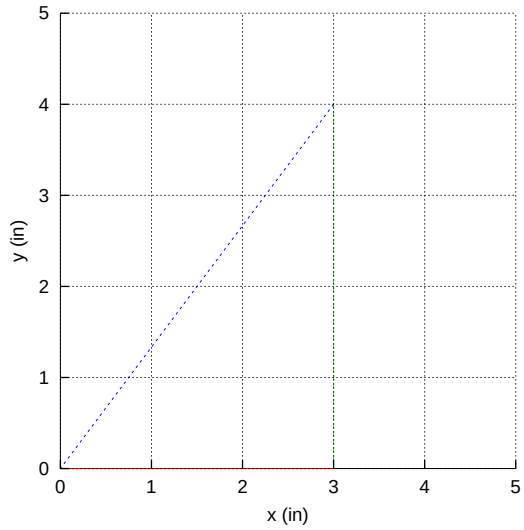


Figure 9.9: The plot for the right triangle traveled by the robot from Program 9.4.

9.2.1 Summary

1. Call the **CLinkbotI** member function

```
robot.drivexy(x, y, radius, trackwidth);
```

to drive a Linkbot-I by x and y relative to its current position in the x and y coordinate system.

2. Call the **CPlot** member function

```
plot.point(x, y);
```

to plot a point.

3. Call the **CPlot** member function

```
plot.line(x1, y1, x2, y2);
```

to plot a straight line between (x_1, y_1) and (x_2, y_2).

4. Call the **CPlot** member function

```
plot.sizeRatio(ratio);
```

to set the aspect ratio of a plot.

5. Call the function

```
sqrt(x);
```

to find the square root of a number.

9.2.2 Terminology

robot.drivexy(), **plot.point()**, **plot.line()**, **plot.sizeRatio()**, aspect ratio, **sqrt**().

9.2.3 Exercises

1. Write a program `points2.ch`, based on Program 9.3, to drive a virtual Linkbot-I in RoboSim from (0, 0) to the following multiple points (-12, -5.5), (-24, -30), (-36, -10), and (-48, -5). Use **plot.point()** to plot the individual points. How does this graph relate to the graph generated by Program 9.3?

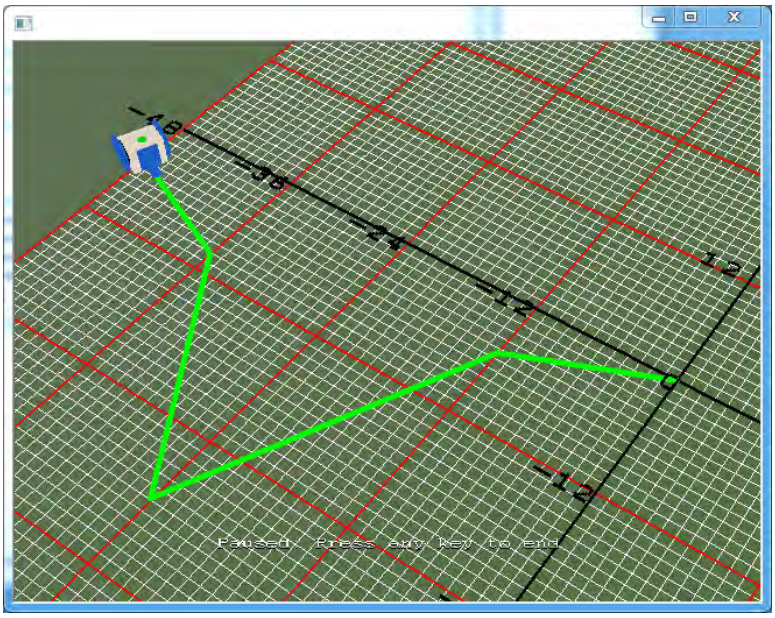


Figure 9.10: The RoboSim scene with the robot trajectory for program `points2.ch`

2. Write a program `righttriangle2.ch`, based on Program 9.4, to drive a virtual Linkbot-I in RoboSim from (0, 0), to (-6, 0), then to (-6, 8), and back to (0, 0). Calculate the length of the hypotenuse for this triangle, and use **plot.line()** and **plot.sizeRatio()** to plot the right triangle. How does this graph relate to the graph generated by Program 9.4?

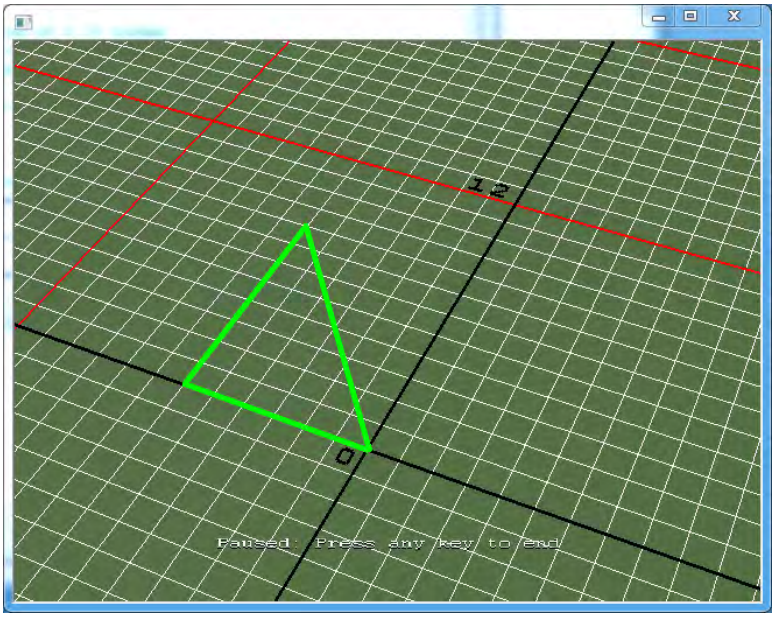


Figure 9.11: The RoboSim scene with the robot trajectory for program `righttriangle2.ch`

3. Move a Linkbot-I from I from (0, 0) to (12, 10), then to (7, 24). Use getxy() to get the x and y coordinates of the robot when the robot is driven from one position to the other.

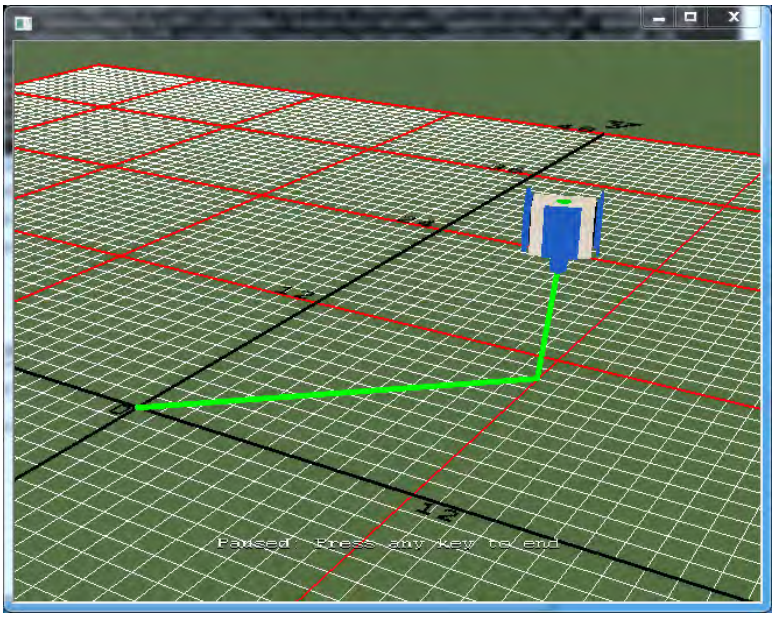


Figure 9.12: The RoboSim scene with the robot trajectories for programs `drivexyto6.ch` and `drivexy6ch`

(a) Write a program `drivexyto6.ch` using the member function **drivexyTo**().

(b) Write a program `drivexy6.ch` using the member function **drivexy**().

9.3 Move a Linkbot-I Along a Trajectory

A Linkbot-I can also be programmed to drive along the trajectory based on an expression or function, such as a polynomial curve. In order to achieve this effect, the Linkbot must travel to many points, with shorter distances between these points. In this section, we will learn how to move a robot along a trajectory.

9.3.1 Move a Linkbot-I Along a Trajectory Using an Expression

Moving a Linkbot along a trajectory specified by an expression can be accomplished by the member function **drivexyToExpr**(). The syntax of the member function **drivexyToExpr**() is as follows.

```
robot.drivexyToExpr(x0, xf, num, expr, radius, trackwidth);
```

This member function drives a Linkbot-I following the trajectory specified by the expression `expr` in the range `[x0, xf]` using `num` points. The expression `expr` should be a valid Ch expression in terms of the variable `x`. The last two arguments specify the radius of the two wheels and the track width of the robot.

Program 9.5 drives a Linkbot-I along the trajectory of a parabola in a coordinate system. Before running Program 9.6, set the initial position of the Linkbot-I in the RoboSim GUI to $(-6, 5.5)$ or place the hardware robot in the position $(-6, 5.5)$.

```
/* File: drivexytoexpr.ch
   Note: This program uses drivexyToExpr() available in RoboSim only.
   Set the initial position (x, y, angle) in RoboSim GUI to (-6, 5.5, 90) for the robot.

   A robot moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6
     with 30 points.
   Plot the polynomial y for x from -6 to 6 with 500 points.
   The range of x-axis is from -12 to 12.
   The range of y-axis is from -15 to 12.
   The tics range for x and y axes is 1. */
#include <chplot.h>    /* for CPlot */
#include <linkbot.h>   /* for CLinkbotI */
CPlot plot;
CLinkbotI robot;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y;               // x and y coordinates

// uncomment line below for controlling a hardware robot
//robot.initPosition(-6, 5.5, 90);

/* move the robot along the polynomial curve */
robot.drivexyToExpr(-6, 6, 30, "0.5*(x+5)*(x-5)", radius, trackwidth);

/* plot the polynomial curve */
plot.title("y = 0.5(x+5)(x-5)");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 12);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.expr(-6, 6, 500, "0.5*(x+5)*(x-5)");
plot.plotting();
```

Program 9.5: Moving a Linkbot-I to follow a polynomial curve using **drivexyToExpr**().

Recall that a parabola is a graph of a quadratic equation. A quadratic equation is a second-degree polynomial equation of the form

$$y = ax^2 + bx + c$$

where $a \neq 0$. Graphs of parabolas are smooth, cup-shaped symmetric curves with a top or bottom point where the curve changes direction, called a vertex. If $a > 0$, the curve opens up and the y-coordinate of the vertex is the minimum value of y. If $a < 0$, it opens down and the y-coordinate is the minimum value of y. The quadratic equation in Program 9.6 is

$$y = 0.5(x + 5)(x - 5) = 0.5x^2 - 1.25$$

Since $a > 0$ in this case, the curve opens up.

The initial position for the robot is $(-6, 5.5, 90)$ is different from the default initial position (0, 0, 90). As described in section 9.1.2, in order to move a hardware robot not starting in its default position, Program 9.5 will need to contain the following statement

```
    robot.initPosition(-6, 5.5, 90);
```

In Program 9.5, this line is commented out as it is not needed to control a robot in RoboSim.

The statement

```
robot.drivexyToExpr(-6, 6, 30, expr, radius, trackwidth);
```

drives the robot for x from -6 to 6 along the parabola with 30 points. When x is -6 or 6, y is 5.5. Therefore, the robot drives from the coordinate $(-6, 5.5)$ to (6, 5.5). Figure 9.13 shows the resulting smooth parabolic trajectory.

To plot the parabolic trajectory, the CPlot member functions **axisRange()**, **ticsRange()**, and **expr()** are used. An ***axis range*** describes the beginning and ending numbers shown on each axis. The axis range can be set by the member function **axisRange()**, which has the general syntax

```
plot.axisRange(axis, minimum, maximum);
```

The argument `axis` specifies either the x or y axis of the plot. The arguments `minimum` and `maximum` specify the start and end points of the range.

A ***tick mark*** is the interval between units on each axis. This interval can be set by the member function **ticsRange()**, which has the general syntax

```
plot.ticsRange(axis, incr);
```

Like the member function **axisRange()**, the first argument `axis` specifies either the x or y axis. The second argument `incr` gives the increment between tick marks.

The relation represented in an expression can be plotted by using the CPlot member function **expr()**. The syntax of this function is

```
plot.expr(x0, xf, num, expre);
```

This member function will plot a function defined as a Ch expression `expre` in the range `[x0, xf]` using `num` points.

In Program 9.6, the following lines

```
plot.axisRange(PLOT_AXIS_X, -12, 12);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.expr(-6, 6, 500, "0.5*(x+5)*(x-5)");
```

plot the trajectory of the Linkbot-I in the ranges `[-12, 12]` and `[-15, 12]` on the x and y axes, with an increment of 1 between tick marks. 500 points are used to plot the curve, since plotting with CPlot member functions requires more accuracy. Figure 9.14 shows the generated graph.

E Do Exercise 1 on page 155.

9.3.2 Move a Linkbot-I Along a Trajectory Using a Function

Moving a Linkbot along a trajectory specified by a function can also be accomplished by the member function **drivexyToFunc**(). The syntax of the member function **drivexyToFunc**() is as follows.

```
robot.drivexyToFunc(x0, xf, num, func, radius, trackwidth);
```

This member function drives a Linkbot-I following the trajectory specified by the function defined as `func()` in the range `[x0, xf]` using `num` points. The last two arguments specify the radius of the two wheels and the track width of the robot.

Program 9.6 is the same as Program 9.5, except that it uses the statement

```
robot.drivexyToFunc(-6, 6, 30, func, radius, trackwidth);
```

to drive the robot for x from -6 to 6 along the parabola with 30 points using a function `func()`, instead of using the member function **drivexyToExpr**().

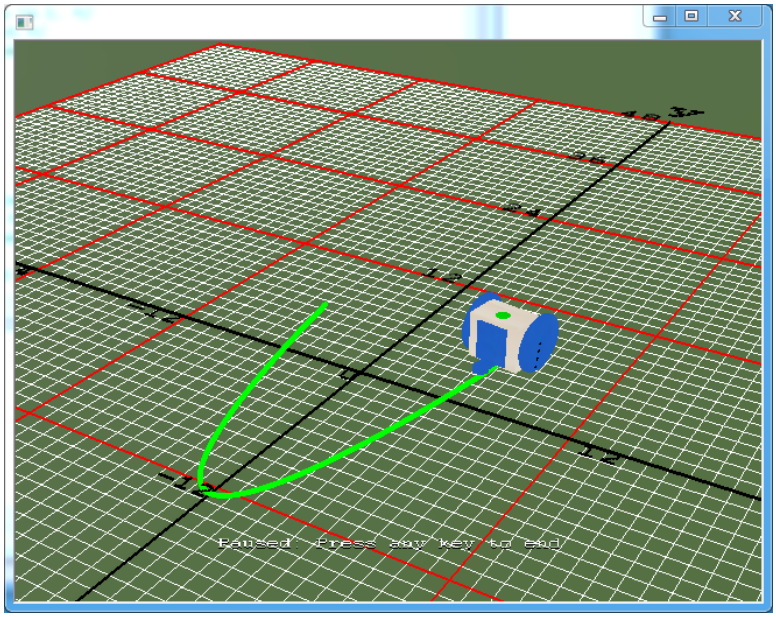


Figure 9.13: The RoboSim scene with the robot trajectory of a polynomial from Programs 9.5, 9.6, and 14.6.

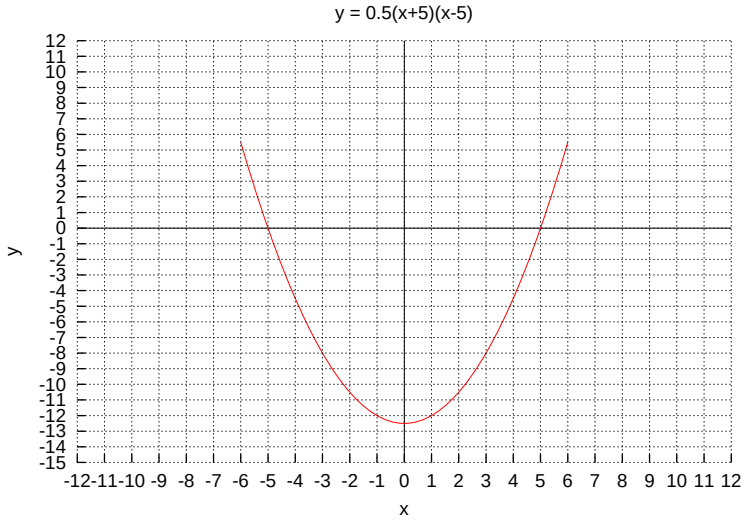


Figure 9.14: The plot for the polynomial traveled by the robot from Programs 9.5, 9.6, and 14.6.

In Program 9.6 a function `func()` is used to define this quadratic equation, which calculates the y-coordinate for each point the virtual Linkbot-I travels to on the parabola. The function `func()` is defined as

```
/* define the function func() */
double func(double x) {
    return 0.5*(x+5)*(x-5);
}
```

This function takes in an argument of type **double**, which is the x-coordinate of a point. The function returns a value of type **double**, which is the corresponding y-coordinate. Since `func()` is defined before it is used in Program 9.6, a function prototype is not needed.

The relation represented in a function in Program 9.6 is plotted by using the **CPlot** member function **func2D**(). The syntax of this function is

```
plot.func2D(x0, xf, num, func);
```

This member function will plot a function defined as `func()` in the range `[x0, xf]` using `num` points.

The output from Program 9.6 is the same as that from Program 9.5, as shown in Figures 9.13 and 9.14.

```
/* File: drivexytoFunc.ch
   Note: This program uses drivexyToFunc() available in RoboSim only.
   Set the initial position (x, y, angle) in RoboSim GUI to (-6, 5.5, 90) for the robot.

   A robot moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6.
   Plot the polynomial y for x from -6 to 6 with 500 points.
   The range of x-axis is from -12 to 12.
   The range of y-axis is from -15 to 12.
   The tics range for x and y axes is 1. */
#include <chplot.h>    /* for CPlot */
#include <linkbot.h>   /* for CLinkbotI */
CPlot plot;
CLinkbotI robot;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y;               // x and y coordinates

double func(double x) {
    return 0.5*(x+5)*(x-5);
}

// uncomment line below for controlling a hardware robot
//robot.initPosition(-6, 5.5, 90);

/* move the robot along the polynomial curve */
robot.drivexyToFunc(-6, 6, 30, func, radius, trackwidth);

/* plot the polynomial curve */
plot.title("y = 0.5(x+5)(x-5)");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 12);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.func2D(-6, 6, 500, func);
plot.plotting();
```

Program 9.6: Moving a Linkbot-I to follow a polynomial curve using **drivexyToFunc**().

Do Exercise 2 on page 155.

9.3.3 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.drivexyToFunc(x0, xf, num, func, radius, trackwidth);
   ```

   drives a Linkbot-I based on a function `func()`, for x from `x0` to `xf` with `num` number of points.

2. Call the **CLinkbotI** member function

   ```
   robot.drivexyToExpr(x0, xf, num, expr, radius, trackwidth);
   ```

   drives a Linkbot-I based on an expression `expr` in terms of the variable x, for x from `x0` to `xf` with `num` number of points.

3. Call the CPlot member function

```
plot.expr(x0, xf, num, expre);
```

to plot the math expression **expre** in terms of variable x in the range `[x0, xf]` using `num` points.

4. Call the CPlot member function

```
plot.func2D(x0, xf, num, func);
```

to plot the function **func()** for x in the range `[x0, xf]` using `num` points.

9.3.4 Terminology

robot.drivexyToFunc(), **robot.drivexyToExpr()**, **plot.axisRange()**, **plot.ticsRange()**, **plot.func2D()**, axis range, tick mark.

9.3.5 Exercises

1. Write a program `drivexytoexpr2.ch`, using the member function **drivexyToExpr()**, based on Program 9.5, that drives a Linkbot-I along the trajectory of a parabola from (-6, -5.5) to (6, -5.5), as shown in Figure 9.15. The parabola is defined by the formula $y_1 = -0.5(x + 5)(x - 5)$. This parabola will open downward, with vertex at (0, 12.5). Use the CPlot member functions **axisRange()**, **ticsRange()**, and **expr()** to generate a plot of the Linkbot-I's trajectory.

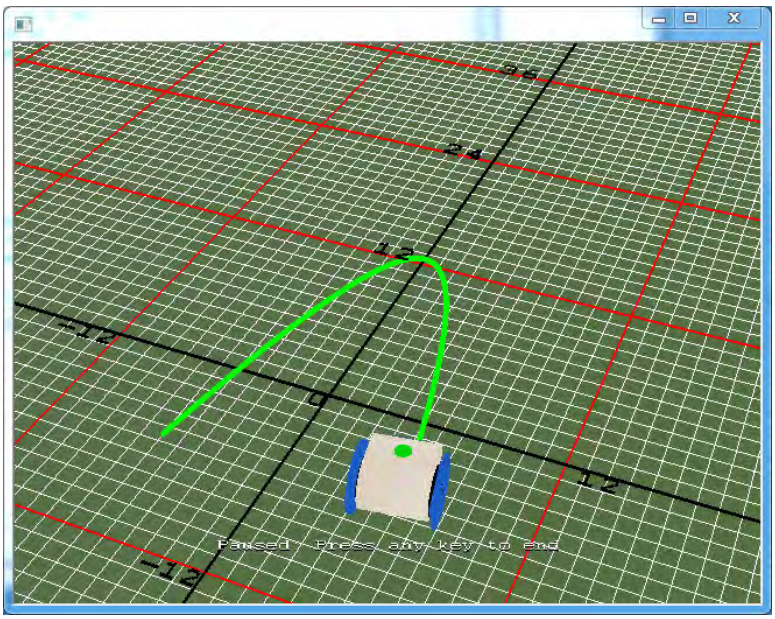


Figure 9.15: The RoboSim scene with the robot trajectory for program `drivexytofunc2.ch`.

2. Write a program `drivexytofunc2.ch` using the member function **drivexyToFunc()**, based on Program 9.6, that drives a Linkbot-I along the trajectory of a parabola from (-6, -5.5) to (6, -5.5), as shown in Figure 9.15. Use the CPlot member function **func2D()** to generate a plot of the Linkbot-I's trajectory.

9.4 Trace the Positions of a Linkbot-I in RoboSim

When the check box "Enable Robot Position Tracing", as shown in Figure 4.1 on page 45, is selected on the RoboSim GUI, the trajectory for each Linkbot will be traced and displayed on the RoboSim scene. In some applications, it will be desirable to only trace certain parts of a trajectory. This can be accomplished by the member functions **traceOn**() and **traceOff**(). The member function **traceOn**() turns on tracing for a robot whereas **traceOff**() turns off tracing. The syntaxes of the member functions **traceOn**() and **traceOff**() are as follows.

```
    robot.traceOn();
    robot.traceOff();
```

These two member functions inside a program will overwrite the setting for the tracing selected on the RoboSim GUI.

Program 9.7 demonstrates how to draw two parallel lines using a Linkbot-I as shown in Figure 9.16. The motion and tracing of lines are handled by the following code segment in Program 9.7.

```
    robot.traceOn();
    robot.drivexyTo(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)

    robot.traceOff();
    robot.drivexyTo(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)

    robot.traceOn();
    robot.drivexyTo(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)
```

Before the robot is driven from (0, 0) to (6, 5), the tracing is turned on to draw the first line. For the movement from (6, 5) to (0, 10), the tracing is turned off. Before the robot is driven from (0, 10) to (6, 15), the tracing is turned on again to draw the second line, which is parallel to the first line.

```
/* File: traceon.ch
   Turn trace on and off with two Linkbot-Is
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for robot. */
#include <linkbot.h>
CLinkbotI robot;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels

robot.traceOn();
robot.drivexyTo(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)

robot.traceOff();
robot.drivexyTo(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)

robot.traceOn();
robot.drivexyTo(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)
```

Program 9.7: Drawing two parallel lines from (0, 0) to (6, 5) and from (0, 10) to (6, 15).

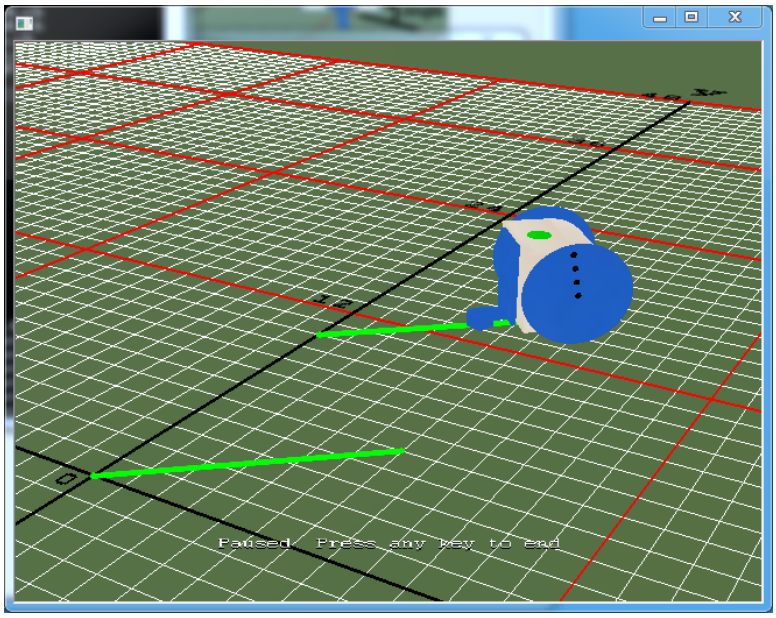


Figure 9.16: The RoboSim scene with the robot trajectory from Programs 9.7 and 9.8.

E Do Exercises 1 and 2 on 157.

9.4.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.traceOn();
   ```

   to turn on tracing of positions for a Linkbot-I.

2. Call the **CLinkbotI** member function

   ```
   robot.traceOff();
   ```

   to turn off tracing of positions for a Linkbot-I.

9.4.2 Terminology

robot.traceOn(), **robot.traceOff()**, and trace the trajectory of a robot.

9.4.3 Exercises

1. Watch the video tutorial "R7. Turn Trace On or Off: traceOn() and traveOff()" in http://roboblockly.ucdavis.edu/videos.

2. Write a program `traceon3.ch` to trace the letter 'A' in RoboSim as shown in Figure 9.17. The horizontal line for the letter 'A' can be drawn from the point $(-2.5, 5)$ to the point (2.5 5).

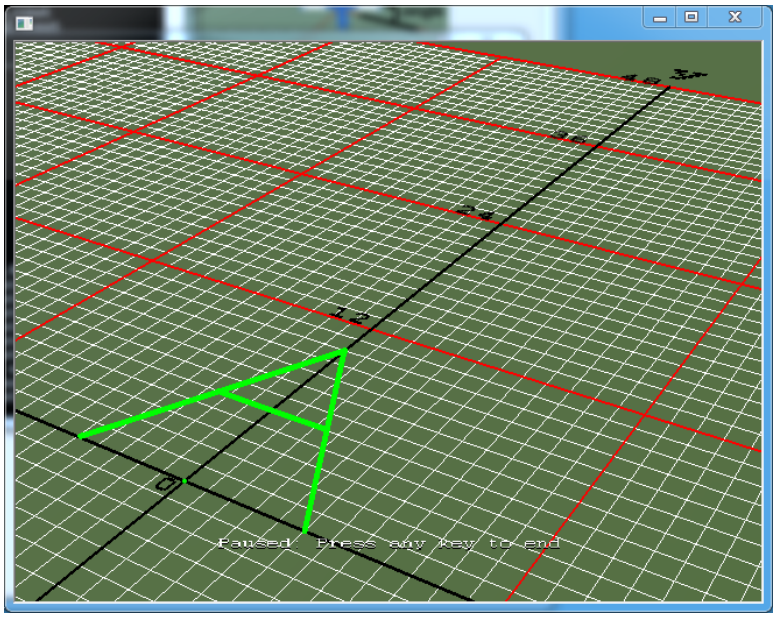


Figure 9.17: Moving a virtual Linkbot-I to trace a letter 'A'.

9.5 Record the Positions of a Linkbot-I

We learned in Section 8.3.1 on how to record and plot the distances versus time for a Linkbot. It is also possible to record and plot the positions (x, y) in a coordinate system for a Linkbot-I in RoboSim. To record the position data in RoboSim, we use the two **CLinkbotI** member functions **recordxyBegin()** and **recordxyEnd()**.

The general syntax for the function **recordxyBegin()** is

```
robot.recordxyBegin(xdata, ydata, timeInterval);
```

The first and second arguments `xdata` and `ydata` contain the data points for positions (x, y) of the Linkbot-I. They are variables of type **robotRecordData_t**. The reason for this is the same as why the `timedata` and `distances` arguments are of type **robotRecordData_t** in Program 9.8. The third argument `timeInterval` is the time interval between position readings. The minimum possible value for `timeInterval` is 0.05 seconds.

The general syntax for the function **recordxyEnd()** is

```
robot.recordxyEnd(numDataPoints);
```

The argument `numDataPoints` is the total number of data points that were recorded while the Linkbot-I was moving.

Program 9.7 draws two parallel lines as shown in Figure 9.16. Based on Program 9.7, Program 9.8 records the positions (x, y) for the Linkbot-I and plots the acquired data using the Ch plotting member functions with the output shown in Figure 9.18.

```
/* File: recordxy.ch
   Trace and record the x and y positions, plot the acquired data
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for robot. */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;   // the track width, the distance between two wheels
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t xdata, ydata; // recorded x and y positions
CPlot plot;                 // plotting class

/* begin recording x and y positions */
robot.recordxyBegin(xdata, ydata, timeInterval);

robot.traceOn();
robot.drivexyTo(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)

robot.traceOff();
robot.drivexyTo(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)

robot.traceOn();
robot.drivexyTo(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)

/* end recording x and y positions */
robot.recordxyEnd(numDataPoints);

/* plot the data */
plot.title("Position");
plot.label(PLOT_AXIS_X, "X (inches)");
plot.label(PLOT_AXIS_Y, "Y (inches)");
plot.axisRange(PLOT_AXIS_X, -5, 15);
plot.axisRange(PLOT_AXIS_Y, 0, 20);
plot.scattern(xdata, ydata, numDataPoints, "green");
plot.sizeRatio(1);
plot.plotting();
```

Program 9.8: Plotting the positions of a Linkbot-I.

Similar to Program 8.9, we begin the recording of positions when the tracing of the trajectory is turned on. The statement

```
    robot.recordxyBegin(xdata, ydata, timeInterval);
```

starts recording the positions of the Linkbot-I every 0.1 second. Only the positions displayed on the traced trajectory will be recorded. After the Linkbot stops moving, the statement

```
    robot.recordxyEnd(JOINT1, numDataPoints);
```

stops recording the positions for the traced trajectory.

By default, adjacent data points in a plot are connected to each other to form a line. To plot the positions of the traced trajectory, we need to plot the position data using a scatter plot. This is accomplished by the member function **scattern**().

```
    plot.scattern(xdata, ydata, numDataPoints, "green");
```

E Do Exercise 1 on page 160.

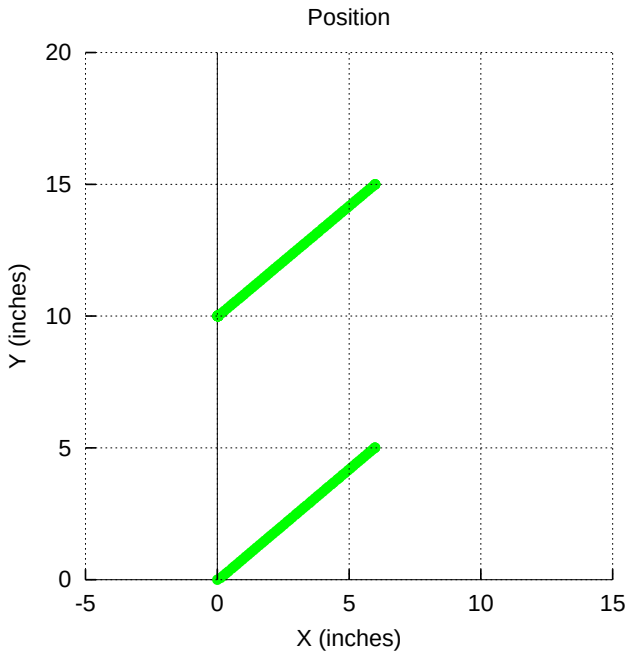


Figure 9.18: The plot for the positions Program 9.8.

9.5.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.recordxyBegin(xdata, ydata, timeInterval);
   ```

   to start recording positions (x, y) of the traced trajectory for a Linkbot-I in a specified interval between position readings.

2. Call the **CLinkbotI** member function

   ```
   robot.recordxyEnd(id, num);
   ```

   to stop recording positions of the traced trajectory for a Linkbot-I.

3. Call the **CPlot** member function

   ```
   plot.scattern(x, y, n, "green");
   ```

   to plot n data points stored in arrays x and y in a scatter plot with a specified color.

9.5.2 Terminology

robot.recordxyBegin(), **robot.recordxyEnd**(), **plot.plotType**(), **plot.pointType**(), and record positions.

9.5.3 Exercises

1. Based on the program `traceon3.ch` developed in Exercise 1 on on page 160, write a program `recordxy3.ch` to record and plot the traced trajectory with the letter 'A' as shown in Figure 9.19.

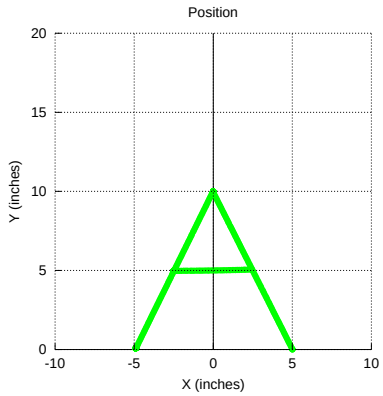


Figure 9.19: A plot generated by the program `recordxy3.ch`.

9.6 Create an Obstacle Course with Points, Lines, and Text on a RoboSim Scene

Points, lines, and text shown in Figure 9.20 can be drawn on the RoboSim scene easily through the graphical user interface (GUI) in RoboSim. Figure 9.21 shows the GUI for drawing this scence. The user can also easily reproduce this scene by loading the RoboSim configuration file `pointlinetext.xml`, available along other Ch programs for this book, through `File->Load Configuration` in the RoboSim. This XML file draws five points at $(-9, 9)$, $(-3, 9)$, $(3, 9)$, $(9, 9)$, and $(15, 9)$ with the point size of 1, 2, 3, 4, 5 in the red, green, blue, purple, and aqua colors, respectively, as shown in Figure 9.20.

We can also place objects such as boxes, cylinders, and spheres with masses on the RoboSim for simulation.

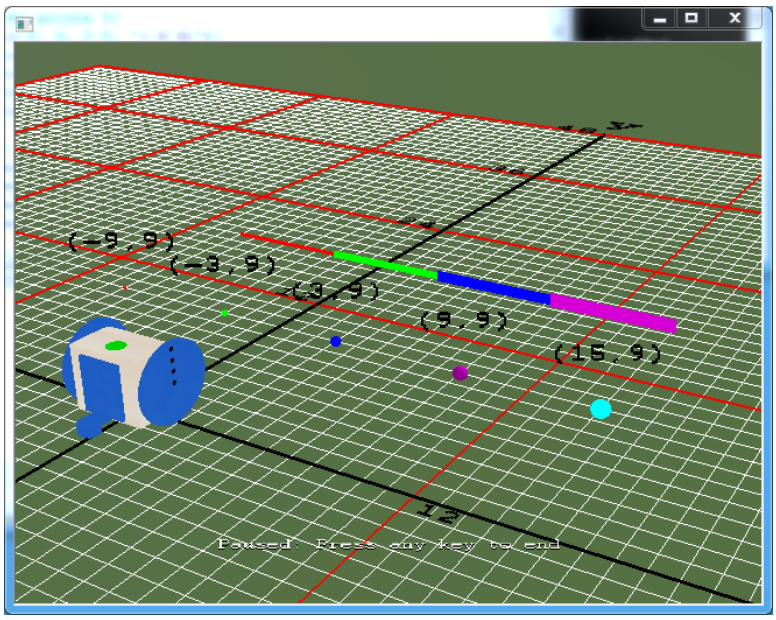


Figure 9.20: The RoboSim scene with points, lines, and text, created by the RoboSim configuration file pointlinetext.xml.

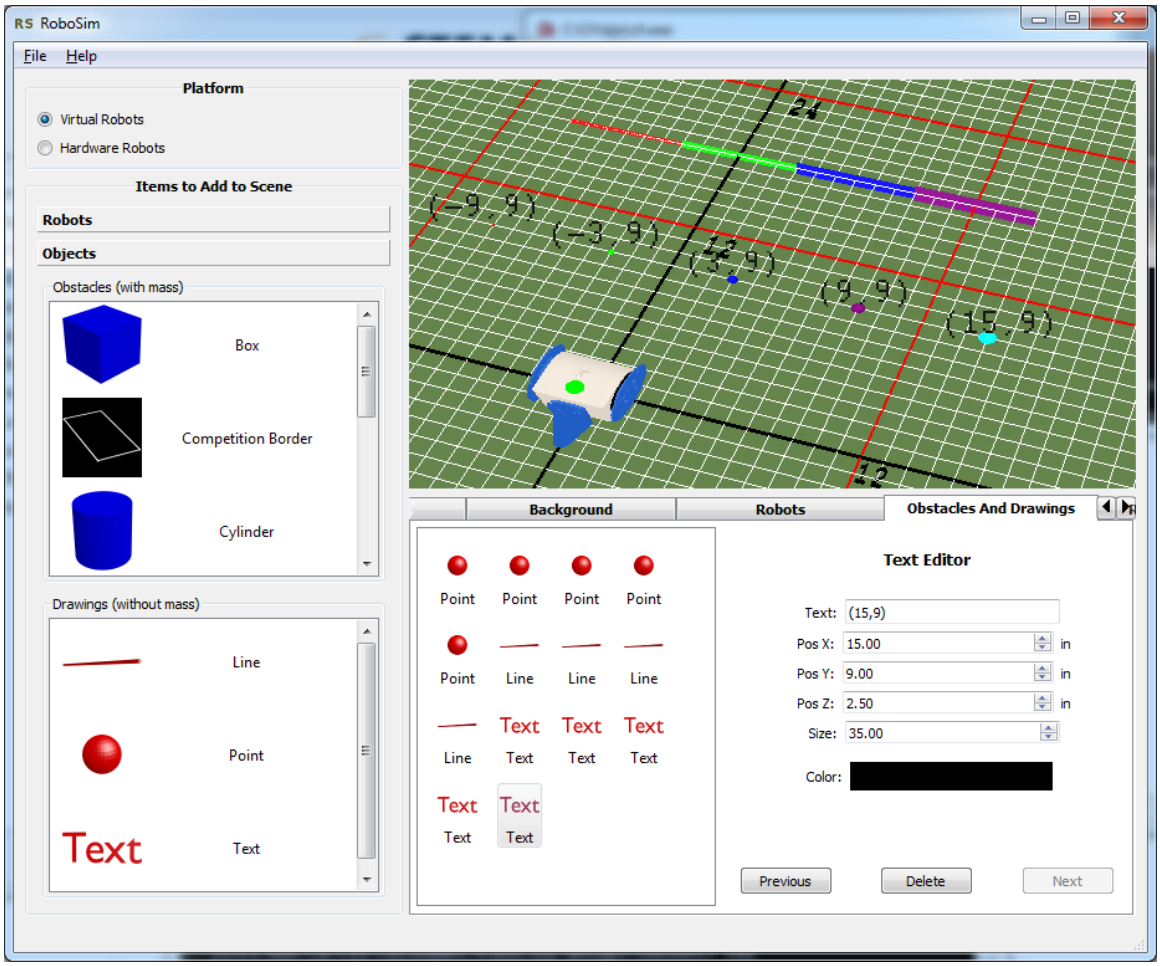


Figure 9.21: The RoboSim GUI to draw the sence shown in Figure 9.20.

E Do Exercises 2 and 3 on 165.

The RoboSim configuration file obstaclecourse.xml draws an obstacle course shown in Figure 9.22.

Program 9.9 drives a robot through the specified points along the obstacle course first. Then, it displays the obstacle course in a separate plot. The robot placed at the origin (0, 0) will travel through these points (12, 5.5), (24, 30), (36, 10), and (48, 5) in different colors. The program statements

```
    /* drive the robot to specified points */
    robot.drivexyTo(12, 5.5, radius, trackwidth);
    robot.drivexyTo(24, 30, radius, trackwidth);
    robot.drivexyTo(36, 10, radius, trackwidth);
    robot.drivexyTo(48, 5, radius, trackwidth);
```

drive the robot through the specified points. The same points and lines are drawn in a Ch plot.

```
/* File: obstaclecourse.ch
   Note: This program uses drivexyTo() to move a Linkbot-I to
   multiple points in an obstacle course marked by the configuration file
   obstaclecourse.xml
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for the robot. */
#include <chplot.h>    /* for CPlot */
#include <linkbot.h>   /* for CLinkbotI */
CPlot plot;
CLinkbotI robot;
```

```
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;   // the track width, the distance between two wheels
double speed = 7;           // the speed of the robot in inches per second

/* set the speed for a two-wheel robot */
robot.setSpeed(speed, radius);

/* move the robot to specified points */
robot.drivexyTo(12, 5.5, radius, trackwidth);
robot.drivexyTo(24, 30, radius, trackwidth);
robot.drivexyTo(36, 10, radius, trackwidth);
robot.drivexyTo(48, 5, radius, trackwidth);

/* plot the points and obstacle course */
plot.title("");
plot.label(PLOT_AXIS_X, "x (in)");
plot.label(PLOT_AXIS_Y, "y (in)");
/* mark points */
plot.point(0, 0);
plot.point(12, 5.5);
plot.point(24, 30);
plot.point(36, 10);
plot.point(48, 5);

/* make obstacle course */
plot.line(-7.2, 0, 9.77, 7.78);
plot.line(9.77, 7.78, 23.73, 36.28);
plot.line(23.73, 36.28, 36+1.75, 12.9);
plot.line(37.75, 12.9, 48, 8.63);
plot.line(7.2, 0, 14.22, 3.22);
plot.line(14.22, 3.22, 24.27, 23.73);
plot.line(24.27, 23.73, 36-1.75, 7.1);
plot.line(36-1.75, 7.1, 48, 1.37);
plot.axisRange(PLOT_AXIS_X, -10, 50); // set the x-axis range
plot.axisRange(PLOT_AXIS_Y, 0, 60);   // set the y-axis range
plot.sizeRatio(1);
plot.plotting();
```

Program 9.9: Moving a Linkbot-I through an obstacle course using **drivexyTo**().

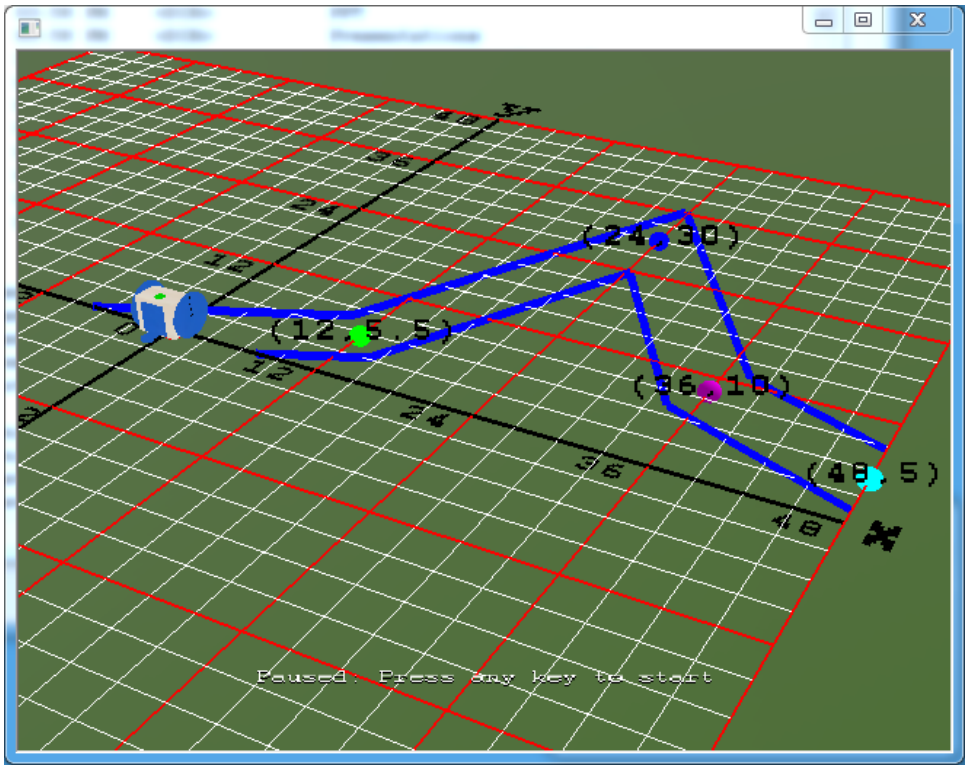


Figure 9.22: The RoboSim scene with the obstacle course created by the XML configuration file obstacle-course.xml.

E Do Exercises 4 and 5 on 166.

You can select different background by clicking the tab “Background” as shown in Figure 9.21. You may also use the pre-built background for RoboPlay Challenges. Figure 9.23 shows how to select a board for the task “Supply Recovery” for for Division 1 of the 2016 RoboPlay Challenge Competition

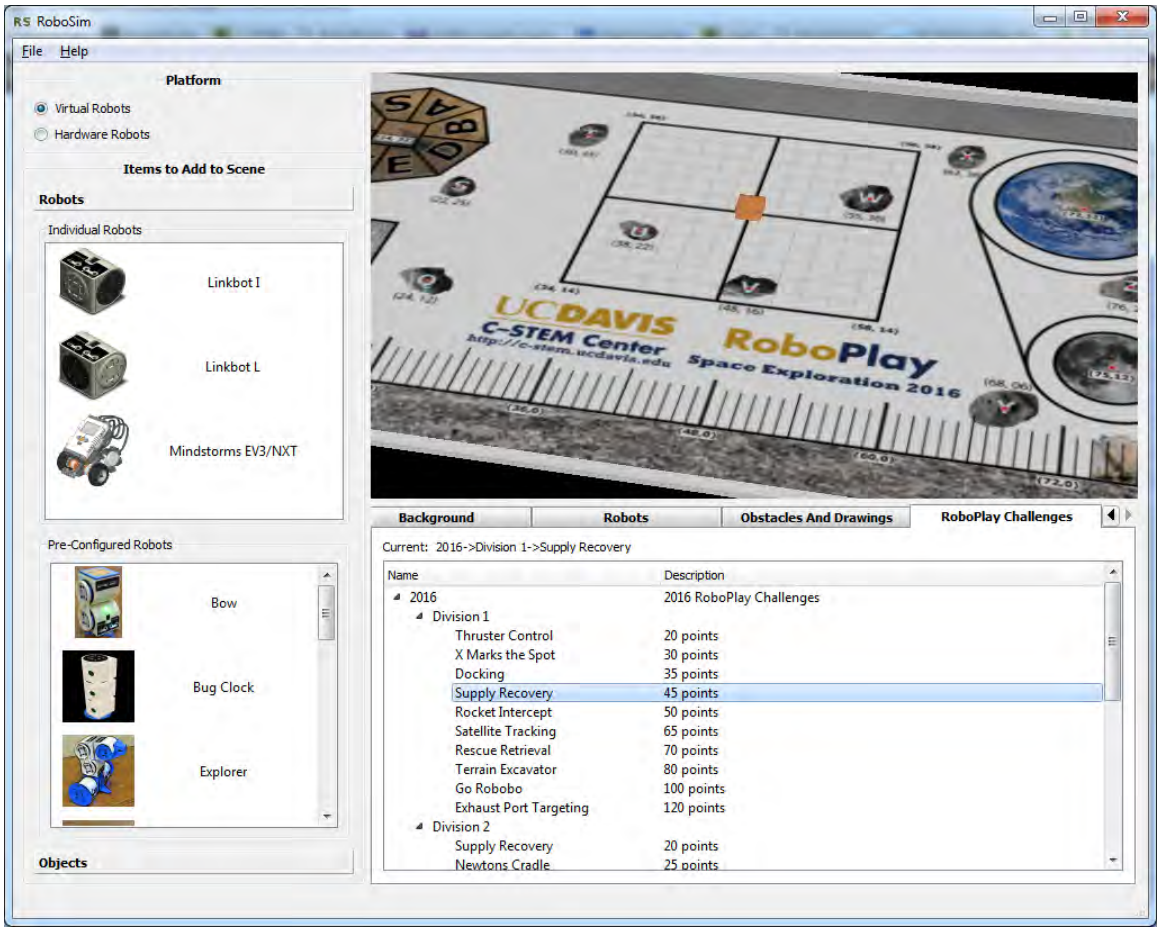


Figure 9.23: Selecting the board for the task "Supply Recovery" for Division 1 of the 2016 RoboPlay Challenge Competition.

E Do Exercise 6 on 167.

9.6.1 Summary

1. Create drawings on the RoboSim.
2. Create objects on the RoboSim.

9.6.2 Terminology

drawings, objects.

9.6.3 Exercises

1. Watch the video tutorial RoboSim in http://c-stem.ucdavis.edu/studio/tutorial/.
2. Create an XML RoboSim configuration file `pointlinetext2.xml` to draw an obstacle course as shown in Figure 9.24. The size for points in the obstacle course is 2. The line widths for the red, green, blue, and purple color are 1, 2, 3, and 4, respectively.

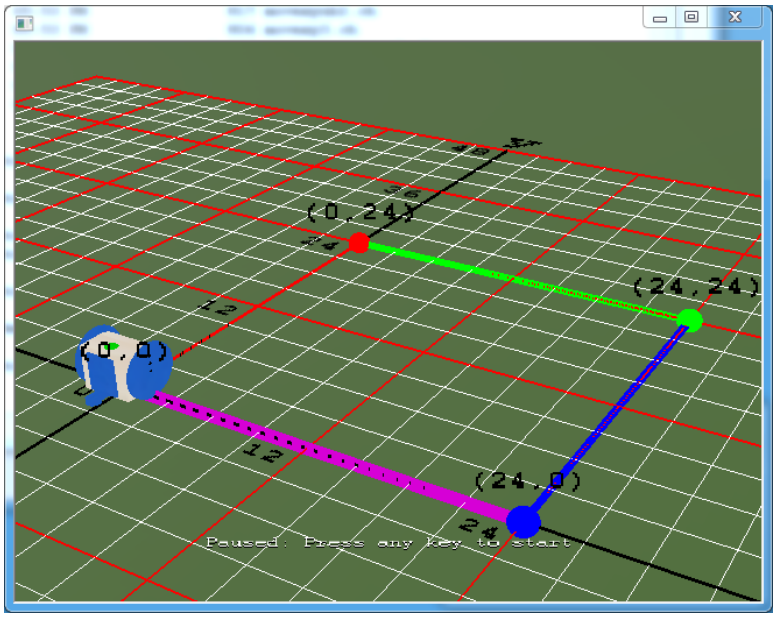


Figure 9.24: The RoboSim scene with an obstacle course with an XML configuration file `pointlinetext2.xml`

3. Create an XML RoboSim configuration file `pointlinetext3.xml` to draw an obstacle course as shown in Figure 9.25. The size for points in the obstacle course is 2. The line width for all lines is 2.

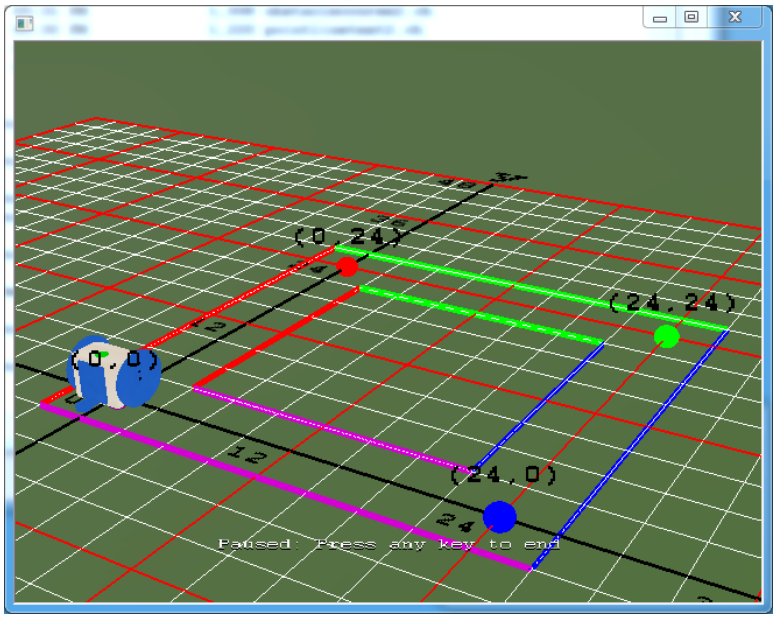


Figure 9.25: The RoboSim scene with an obstacle course by the program `pointlinetext3.ch`

4. Write a program `obstaclecourse2.ch` to drive a robot following the obstacle course as shown in Figure 9.24.

5. Write a program `obstaclecourse3.ch` to drive a robot following the obstacle course as shown in Figure 9.25.

6. Write a program to solve a task for a RoboPlay Challenge Competition. The tasks for RoboPlay Challenge Competition in past years are available in C-STEM Studio −> Documentation.

CHAPTER 10

Writing Programs to Control a Single Linkbot with Different Motion Characteristics

In the previous chapters, we learned the member functions **robot.driveAngle**(), **robot.driveDistance**() **robot.turnLeft**(), **robot.turnRight**(), and of the class **CLinkbotI** for controlling the Linkbot-I. In this chapter, we will learn more features of the Linkbot-I and additional member functions to control the Linkbot-I for various applications. We will also learn the member functions of the **CLinkbotI** class for controlling two joints of a Linkbot-I with different characteristics. In addition, we will learn how to control a Linkbot-L.

10.1 Move Joints Relative to their Current Positions

The member functions of the class **CLinkbotI**, such as **driveDistance**(), treat a Linkbot-I as a two-wheel vehicle. Both joints 1 and 2 are controlled together internally. For some applications, we may need to

control each joint of a Linkbot individually. For example, when multiple Linkbots are connected together to form a different robotic system, we may need to control each joint of connected Linkbots separately. The member function **move**() can be used for such a contol purpose as shown in Program 10.1.

```
/* File: move.ch
   Move joints relative to the current positions. */
#include <linkbot.h>
CLinkbotI robot;

/* move joint 1 by 360 degrees and joint 3 by -360 degrees */
robot.move(360, NaN, -360);
```

Program 10.1: The first program to control a Linkbot-I.

The general syntax of the **CLinkbotI** member function **move**() to move two joints of a Linkbot relative to their current positions is as follows.

```
    robot.move(angle1, NaN, angle3);
```

It has three arguments for three joints, each representing a joint angle relative to its current positions. Since joint 2 cannot be moved, the argument **NaN** which stands for Not-a-Number is used. In Program 10.1, the statement

```
    robot.move(360, NaN, -360);
```

moves joint 1 by 360 degrees and joint 3 by -360 degrees relative to their current position at the same time. In this way, the robot will move forward. The above function call is equivalent to

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

If the value for a joint angle for the member function **move()** is negative, the joint moves in the oppositive direction. For example, the statement

```
    robot.move(-360, NaN, 360);
```

would move joint 1 by -360 degrees and joint 3 by 360 degrees relative to their current positions. The robot will move backward for 360 degrees for both joints 1 and 3, which is equivalent to

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

E Do Exercises 1, 2, 3 on page 170.

10.1.1 Summary

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

1. Call the **CLinkbotI** member function

   ```
       robot.move(angle1, NaN, angle3);
   ```

   to move joints 1 and 3 of a Linkbot-I relative to their current positions specified in its first and third arguments.

10.1.2 Terminology

#include <linkbot.h>, **NaN**, Not-a-Number, **robot.move()**.

```
/* File: multiplemove.ch
   Move all joints relative to the current positions . */
#include <linkbot.h>
CLinkbotI robot;

/* move the joint 1 by 180 degrees and joint 3 by -180 degrees */
robot.move(180, NaN, -180);

/* move the joint 1 by 180 degrees and joint 3 by -180 degrees */
robot.move(180, NaN, -180);

/* move the joint 1 by 360 degrees and joint 3 by -360 degrees */
robot.move(360, NaN, -360);
```

Program 10.2: Moving joints relative to the current position using **move()**.

10.1.3 Exercises

1. Write a program `move2.ch` to move the two rotating joints of a Linkbot-I two full rotations (hint: one full rotation is 360 degrees) to the position of 720 degrees.
2. Write a program `move3.ch` to make a Linkbot-I turn left by rotating only joint 3 by -360 degrees.
3. Write a program `move4.ch` to make a Linkbot-I turn left fast by rotating joint 1 by -360 degrees and joint 3 by -360 degrees.

10.2 Multiple Motion Statements in a Program

A Linkbot program can contain multiple motion statements. For example, Program 10.2 has three motion statements. The program calls the member function **move**() three times. The first function call

```
robot.move(180, NaN, -180);
```

moves joint 1 by 180 degrees and joint 3 by -180 degrees relative to the zero position. The second time

```
robot.move(180, NaN, -180);
```

is called, joint 1 moves by 180 degrees and joint 3 by -180 degrees relative to their current position. After this function call, the absolute positions of joints 1 and 3 are 360 degrees and -360 degrees, respectively. The function call

```
robot.move(360, NaN, -360);
```

moves joint 1 by 360 degrees and joint 3 by -360 degrees relative to the current position. At the end of the program, the absolute positions of joints 1 and 3 are 720 degrees and -720 degrees, respectively.

E Do Exercise 1 on page 171.

10.2.1 Summary

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

1. Call the **CLinkbotI** member function

   ```
   robot.move(angle1, NaN, angle3);
   ```

   multiple times to move a Linkbot-I.

10.2.2 Terminology

multiple movements.

10.2.3 Exercises

1. Write a program `multiplemove2.ch` to move a Linkbot-I forward by 360 degrees and then backward by 360 degrees using the function **move**() only.

10.3 Move Joints to their Absolute Positions and Reset to the Zero Positions

The member function **move**(), first presented in section 10.1, can be used to move joints of a Linkbot relative to their current positions. Similar to **move**(), the member function **moveTo**() can move joints of a Linkbot to its specified absolute positions. For the most part, the relative position is the most frequently used, in comparison with the absolute position.

The syntax of the member function **moveTo**() is as follows.

```
robot.moveTo(angle1, NaN, angle3);
```

The function uses the first and third arguments for joints 1 and 3, respectively, each representing an absolute position of a joint.

```
/* File: moveto.ch
   Move all joints to the absolute positions. */
#include <linkbot.h>
CLinkbotI robot;

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

/* move joints 1 to 180 degrees and joint 3 to -180 degrees */
robot.moveTo(180, NaN, -180);

/* move joints 1 to 360 degrees and joint 3 to -360 degrees */
robot.moveTo(360, NaN, -360);

/* move joints 1 to 720 degrees and joint 3 to -720 degrees */
robot.moveTo(720, NaN, -720);
```

Program 10.3: Calling **moveTo()** multiple times.

As mentioned in section 8.1.3, before moving to an absolute joint position using the member function **moveTo**(), the function call

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

is used to set all joints to their zero positions.

The program has three motion statements using the member function **moveTo**(). As indicated in the comments, the following statements

```
/* move joint 1 to 180 degrees and joint 3 to -180 degrees */
robot.moveTo(180, NaN, -180);

/* move joint 1 to 360 degrees and joint 3 to -360 degrees */
robot.moveTo(360, NaN, -360);

/* move joint 1 to 720 degrees and joint 3 to -720 degrees */
robot.moveTo(720, NaN, -720);
```

first move joint 1 to 180 degrees and joint 3 to -180 degrees. As soon as the joints reach these positions, joint 1 is commanded to move to the position of 360 degrees and joint 3 to the position of -360 degrees. Subsequently, joint 1 is moved to the position of 720 degrees and joint 3 to the position of -720 degrees. When you run the program, joints 1 and 3 appear continuously moving from 0 to 720 degrees.

The motion of Program 10.3 using the member function **moveTo**() is the same as that of Program 10.2 using the member function **move**(). When you run these programs, joint 1 appears continuously moving from 0 to 720 degrees whereas joint 3 moving from 0 to -720 degrees. You can monitor the change of joint angles on the Robot Control Panel in Linkbot Labs as described in section 3.3. You can execute the program in debug mode by the command `Next` in ChIDE to watch the action of each motion statement.


Do Exercises 1, 2, 3, and 4 on page 172.

10.3.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.moveTo(angle1, NaN, angle3);
   ```

   to move joints 1 and 3 of a Linkbot-I to their absolute positions.

2. Call the **CLinkbotI** member function

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

   to move all two joints of a Linkbot to their zero positions.

10.3.2 Terminology

absolute position, **robot.moveTo(), robot.resetToZero()**.

10.3.3 Exercises

1. What is the difference between the member functions **moveTo**() and **move**().

2. Write a program `moveto2.ch` to move joints 1 and 3 forward by 360 degrees and then backward by 360 degrees using the function **moveTo**() only.

3. Write a program `moveto3.ch` to first move joints 1 and 3 to the absolute positions of 90 degrees. Then, move joint 1 by 45 degrees and joint 3 by 90 degrees relative to its current position. Finally, move joints 1 and 3 to the absolute positions of 360 degrees.

4. Run the program `moveto3.ch` developed in Exercise 3 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 Linkbot Labs.

10.4 Control the Linkbot-L

Figure 10.1: A Linkbot-L.

In the previous chapter, we learned how to control a Linkbot-I using Linkbot Labs graphical interface. Similar to controlling a Linkbot-I, a Ch program can be developed to control the Linkbot-L. Program 10.4 contains the code typically used for controlling a Linkbot-L.

```
/* File: linkbotL.ch
   Move joints relative to the current positions for Linkbot-L. */
#include <linkbot.h>
CLinkbotL robot;

/* move joint 1 by 360 degrees and joint 2 by 360 degrees */
robot.move(360, 360, NaN);
```

Program 10.4: A program to control a Linkbot-L.

Like all Linkbot programs, a Linkbot-L program begins with the line

```
#include <linkbot.h>
```

to include the contents of the header file **linkbot.h**. This header file defines both classes **CLinkbotI** and **CLinkbotL** for creating Linkbot-I and Linkbot-L objects, respectively. The line

```
CLinkbotL robot;
```

creates the variable `robot` for controlling a Linkbot-L and connects the variable `robot` to a Linkbot-L that has been previously configured through Linkbot Labs.

The general syntax of the **CLinkbotL** member function **move**() to move joints 1 and 2 of a Linkbot-L relative to their current positions is as follows.

```
robot.move(angle1, angle2, NaN);
```

It has three arguments for three joints, each representing a joint angle relative to its current position. Since joint 3 cannot be moved, the argument **NaN** which stands for Not-a-Number is used. The next line

```
robot.move(360, 360, NaN);
```

moves both joints 1 and 2 by 360 degrees relative to their current positions.

Most member functions for **CLinkbotI** are available for **CLinkbotL**. They are handled similarly for both Linkbot-I and Linkbot-L. The major difference is that the Linkbot-I uses joints 1 and 3 whereas the Linkbot-L uses joints 1 and 2. In general, all ideas and concepts for programming Linkbot-I are applicable to Linkbot-L. Therefore, in the remaining part of this manuscript, the presentation will focus on the Linkbot-I.

However, since the Linkbot-I can be treated as a two-wheel vehicle, member functions **driveAngle**(), **driveDistance**(), **turnLeft**(), and **turnRight**() are available for the class **CLinkbotI** only. We will learn the member function **driveDistance**() in section 3.4, and **turnLeft**() and **turnRight**() in section 5.2.

Do Exercises 1 and 2 on page 174.

10.4.1 Summary

1. Include the header file **linkbot.h** and use the class **CLinkbotL** to declare a variable `robot` by the following two statements

```
#include <linkbot.h>
CLinkbotL robot;
```

   for controlling a Linkbot-L.

2. Call the **CLinkbotL** member function

```
robot.move(angle1, angle2, NaN);
```

   to move joints 1 and 2 of a Linkbot-L relative to their current positions specified in its first and second arguments.

10.4.2 Terminology

CLinkbotL.

10.4.3 Exercises

1. Write a program `linkbotL2.ch` to move the two rotating joints of a Linkbot-L two full rotations of 720 degrees.
2. Write a program `linkbotL3.ch` to rotate joint 2 by -360 degrees for a Linkbot-L.

10.5 Get Joint Angles

10.5.1 Get a Joint Angle

After a Linkbot is moved through several motion statements, you may want to know the current angular position of a joint of the Linkbot. The member function **getJointAngle**() can be used to obtain a joint angle of a Linkbot. The syntax of the member function **getJointAngle**() is as follows.

```
robot.getJointAngle(id, angle);
```

The first argument `id` is for a joint number. It should be one of the enumerated values defined in Table 10.1. These enumerated values are also used in several other member functions to identify joint numbers. The second argument `angle` should be a variable of **double** type. When the function is called, it will read the joint angle of a Linkbot 10 times and store the average of 10 joint angles in degrees in the variable `angle`.

Table 10.1: The enumerated values for joint number.

Enumerated Value	Joint Number
JOINT1	Joint 1
JOINT2	Joint 2
JOINT3	Joint 3

```
/* File: getjointangle.ch
   Get a joint angle. */
#include <linkbot.h>
CLinkbotI robot;
double angle;  // declare the variable 'angle' to hold a joint angle

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

/* move the joint 1 by 90 degrees and joint 3 by -90 degrees */
robot.move(90, NaN, -90);
/* obtain the angle for joint 1 */
robot.getJointAngle(JOINT1, angle);
printf("Joint1 angle = %.2lf degrees.\n", angle);

/* move the joint 1 by 360 degrees and joint 3 by -360 degrees */
robot.moveTo(360, NaN, -360);
robot.getJointAngle(JOINT1, angle);
printf("Joint1 angle = %.2lf degrees.\n", angle);
```

Program 10.5: Getting a joint angle using **getJointAngle()**.

For example, Program 10.5 uses the member function **getJointAngle**() to obtain joint angles. When Program 10.5 is executed, the following output will be displayed in the input/output pane.

```
Joint1 angle = 90.68 degrees.
Joint1 angle = 360.96 degrees.
```

The resolution for joint angle is about 0.6 degrees. In addition, the robotic system involves many components, most of which are not high precision. Therefore, the function **getJointAngle**() does not give an exact angle. If you run the program in your computer, the displayed joint angles will be different.

As mentioned in section 8.1.3, before getting a joint angle using the member function **getJointAngle**(), the function call

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

sets all joints to their zero positions.

After the first movement statement

```
robot.move(90, NaN, -90);
```

the function call

```
robot.getJointAngle(JOINT1, angle);
```

obtains the joint angle 90.68 degrees for joint 1 and store the result in the variable `angle`. After the second movement statement

```
robot.moveTo(360, NaN, -360);
```

the joint angle for joint 1 from the function **getJointAngle**() is 360.96 degrees.

E Do Exercises 1, 2, and 3 on page 177.

10.5.2 Get Multiple Joint Angles

The member function **getJointAngle**() can be used to obtain one joint angle at a time. You can call the function multiple times to obtain joint angles for different joints. However, you can call the member function

getJointAngles() once to obtain joint angles for both joints 1 and 3 of a Linkbot-I. The syntax of the member function **getJointAngles**().

```
robot.getJointAngles(angle1, NaN, angle3);
```

The function uses arguments `angle1` and `angle3` to store the joint angles for joints 1 and 3, respectively. Similar to the member function **getJointAngle**(), each joint angle is the average of 10 joint angles read from the Linkbot-I.

```
/* File: getjointangles.ch
   Get joint angles. */
#include <linkbot.h>
CLinkbotI robot;
// declare the variables to hold joint angles
double angle1, angle3;

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

/* move the joint 1 to 360 degrees and joint 3 to -180degrees */
robot.moveTo(360, NaN, -180);
/* obtain joint angles for all four joints */
robot.getJointAngles(angle1, NaN, angle3);
printf("Joint1 angle = %.2lf degrees.\n", angle1);
printf("Joint3 angle = %.2lf degrees.\n", angle3);
```

Program 10.6: Getting joint angles using **getJointAngles**().

Similarly to calling **getJointAngle**(), before getting joint angles using the member function **getJointAngles**(), the function call

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

sets all joints to their zero positions. For example, Program 10.6 uses the member function **getJointAngles**() to get the two joint angles of a Linkbot-I. When the program is executed, the following output will be displayed in the input/output pane.

```
Joint1 angle = 361.02 degrees.
Joint3 angle = -180.29 degrees.
```

Do Exercise 4 on page 177.

10.5.3 Summary

1. Use the enumerated values **JOINT1**, **JOINT2**, and **JOINT3** to specify joints 1, 2, and 3, respectively.
2. Call the **CLinkbotI** member function

   ```
   robot.getJointAngle(JOINT1, angle);
   ```

   to get the joint angle in the second argument for a specified joint in the first argument.
3. Call the **CLinkbotI** member function

   ```
   robot.getJointAngles(angle1, NaN, angle3);
   ```

   to get the two joint angles of a Linkbot-I.

10.5.4 Terminology

JOINT1, **JOINT2**, **JOINT3**, **robot.getJointAngle()**, **robot.getJointAngles()**.

10.5.5 Exercises

1. Watch the video tutorial "R13. Get a Joint Angle: getJointAngle()" in http://roboblockly.ucdavis.edu/videos.
2. What do you think the `average` means for the function **getJointAngle**()? Can you think of a situation where using the function **getJointAngle**() is necessary.
3. Write a program `getjointangle2.ch` to move joints 1 and 3 to the absolute positions of 180 and -360 degrees, respectively. Then, get the joint angles for each joint by the function **getJointAngle**() and display the values of these joint angles.
4. Write a program `getjointangles2.ch` to move joints 1 and 3 to the absolute positions of 180 and -360 degrees, respectively. Then, get the joint angles for two joints 1 and 3 by the function **getJointAngles**() and display the values of these joint angles.

10.6 Move a Single Joint

The member functions **moveTo**() and **move**() can be used to move two joints of a Linkbot to its specified absolute positions and new positions relative to its current positions. The member functions **moveJointTo**() and **moveJoint**() can be used to move a single joint. The member function **moveJointTo**(), can be used to move a joint of a Linkbot-I to its specified absolute position. The member function **moveJoint**() can move a joint to a new position relative to its current position.

10.6.1 Move a Single Joint to the Absolute Position

The syntax of the member function **moveJointTo**() is as follows.

```
robot.moveJointTo(id, angle);
```

The function has two arguments. The first argument `id` is for a joint number. It should be one of the three enumerated values defined in Table 10.1 for joint numbers. The second argument `angle` is the absolute position of the joint angle.

```
/* File: movejointto.ch
   Move a joint to the absolute position. */
#include <linkbot.h>
CLinkbotI robot;
double angle;

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

/* move joint 1 to the position of 180 degrees. */
robot.moveJointTo(JOINT1, 180);

/* move joint 1 to the position of 360 degrees. */
robot.moveJointTo(JOINT1, 360);

/* move joint 1 to the position of 720 degrees. */
robot.moveJointTo(JOINT1, 720);

robot.getJointAngle(JOINT1, angle);
printf("Joint1 angle = %.2lf degrees.\n", angle);
```

Program 10.7: Moving a joint to the absolution position using **moveJointTo**().

For example, Program 10.7 calls the member function **moveJointTo**() by

```
robot.moveJointTo(JOINT1, 180);
```

to move joint 1 to the position of 180 degrees first. Next, it calls the member function **moveJointTo**() twice again to move joint 1 to 360 degrees and 720 degrees. Then, it calls the member function **getJointAngle**() to obtain the joint angle for joint 1. Finally, it displays the joint angle on the screen. When the program is executed, the output from the following output will be displayed in the input/output pane.

```
Joint1 angle = 720.99 degrees.
```


Do Exercise 1 on page 180.

10.6.2 Move a Single Joint Relative to the Current Position

The syntax of the member function **moveJoint**(), similar to that of **moveJointTo**(), is as follows.

```
robot.moveJoint(id, angle);
```

The function has two arguments. The first argument `id` is for a joint number. It should be one of the three enumerated values defined in Table 10.1 for joint numbers. The second argument `angle` is the position of the joint angle relative to its current position. An application example using the member function **moveJoint**() will be presented in the next section.

10.6.3 Delay the Motion of a Linkbot Using the Member Function delaySeconds()

In some applications, you may want to pause a program for a specified time. For example, a robot may pose for a few seconds for choreography in a robot dance. You may also pause a program so that the motion of a Linkbot can be settled down. The delay of a program execution can be accomplished by the member function **delaySeconds**() with the syntax as follows.

```
robot.delaySeconds(seconds);
```

The argument `seconds` is the pause time in seconds. The pause time can be less than one second. A sample application of the member function **delaySeconds**() is shown in Program 10.8.

```
/* File: movejoint.ch
   Move a joint relative to its current position. */
#include <linkbot.h>
CLinkbotI robot;
double angle;

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

/* move joint 1 by 180 degrees. */
robot.moveJoint(JOINT1, 180);

robot.delaySeconds(0.5); //  delay 0.5 second

/* move joint 1 by 180 degrees. */
robot.moveJoint(JOINT1, 180);

robot.delaySeconds(0.5); //  delay 0.5 second

/* move joint 1 by 360 degrees. */
robot.moveJoint(JOINT1, 360);

robot.delaySeconds(0.5); //  delay 0.5 second

robot.getJointAngle(JOINT1, angle);
printf("Joint1 angle = %.2lf degrees.\n", angle);
```

Program 10.8: Moving a joint relative to the current position using **moveJoint**().

Program 10.8 calls the member function **moveJoint**() to move joint 1 multiple times. The member function call

```
    robot.delaySeconds(0.5); //  delay 0.5 second
```

pauses the program between each call of the member function **moveJoint**(). At the end of the program, the joint angle for joint 1 is obtained by the member function **getJointAngle**().

When the program is executed, the following output will be displayed in the input/output pane.

```
    Joint1 angle = 720.86 degrees.
```

If you remove the statement

```
    robot.delaySeconds(0.5); //  delay 0.5 second
```

in the program, the joint angle obtained by the function **getJointAngle**() will deviate more from the desired angle 720 degrees. This is because the movement created by the function **moveJoint**() will start before the previous movement is fully settled down. These errors will be accumulated at each subsequent call of the function **moveJoint**(). This accumulation error also occurs for the function **move**(), which moves joints relative to its current positions.

Because the functions **moveJointTo**() and **moveTo**() use absolute positions, there is no accumulated error each time they are called. Therefore, for the accuracy with multiple movements, functions **moveJointTo**() and **moveTo**() shall be used.

10.6.4 Summary

1. Call the member function

   ```
   robot.moveJoint(id, angle);
   ```

   to move a joint with the specified angle relative to its current position.

2. Call the member function

   ```
   robot.moveJointTo(id, angle);
   ```

   to move a joint with the specified angle to its absolute position.

3. Call the member function

   ```
   robot.delaySeconds(seconds);
   ```

   to pause a program for `seconds`.

4. There are accumulated errors for each movement relative to its current position.

10.6.5 Terminology

robot.delaySeconds(), **robot.moveJoint()**, **robot.moveJointTo()**, accumulated error, relative position, absolute position.

E Do Exercise 2 on page 180.

10.6.6 Exercises

1. Write a program `movejointto2.ch` to move joint 1 to 45 degrees first, then move joint 3 to 120 degrees. Then get and display the joint angles for joints 1 and 3.

2. Write a program `movejoint2.ch` to move joint 3 to the position of 90 degrees, delay for two seconds, next move joint 3 again by 180 degrees relative to its current position, then get and display the joint angle for joint 3.

10.7 Get and Set Joint Speeds

By default, each Linkbot joint rotates at the angular speed of 90 degrees per second. You can change a joint speed and obtain the joint speed through some member functions of the **CLinkbotI** class.

10.7.1 Get and Set a Joint Speed

The member function **getJointSpeed**() can be used to obtain a joint speed of a Linkbot. The syntax of the member function **getJointSpeed**() is as follows.

```
robot.getJointSpeed(id, speed);
```

The first argument `id` is for a joint number. It should be one of the enumerated values defined in Table 10.1. The second argument `speed` should be a variable of **double** type. When the function is called, it will store the joint speed in degrees per second in the variable `speed`.

The member function **setJointSpeed**() can be used to set the angular speed of of a joint for a Linkbot. The syntax of the member function **setJointSpeed**() is as follows.

```
robot.setJointSpeed(id, speed);
```

The first argument `id` is for a joint number. It should be one of the enumerated values defined in Table 10.1. The second argument `speed` is the angular speed to be set for the joint. The minimum joint speed for a Linkbot is -200 degrees per second. The maximum joint speed for a Linkbot is 200 degrees per second. When the speed is positive, a joint moves in count clockwise direction as shown in Figure 2.2. When the speed is negative, a joint moves in clockwise direction.

```
/* File: setjointspeed.ch
   Get and set a joint speed. */
#include <linkbot.h>
CLinkbotI robot;
double speed;  // the joint angular speed in degrees per second

/* get the joint speed for joint 1 */
robot.getJointSpeed(JOINT1, speed);
printf("Joint1 speed = %.2lf degrees per second\n", speed);

/* set the joint angular speed for joint 1 to 45 degrees per second */
speed = 45;
robot.setJointSpeed(JOINT1, speed);

/* rotate joint 1 by 360 degrees */
robot.moveJoint(JOINT1, 360);

/* get the joint angular speed for joint 1 */
robot.getJointSpeed(JOINT1, speed);
printf("Joint1 angular speed = %.2lf degrees per second\n", speed);
```

Program 10.9: Getting and setting a joint speed in degrees per second using **getJointSpeed**() and **setJointSpeed**().

For example, when Program 10.9 is executed, the following output will be displayed in the input/output pane.

```
Joint1 angular speed = 90.00 degrees per second
Joint1 angular speed = 45.00 degrees per second
```

The first function call

```
robot.getJointSpeed(JOINT1, speed);
```

obtains the default angular speed of 90 degrees per second for joint 1. When the function **getJointSpeed**() is called the second time, the joint angular speed of 45 degrees per second, set by the function **setJointSpeed**(), will be obtained.

E Do Exercise 1 on page 186.

10.7.2 Get and Set Joint Speeds

The member functions **getJointSpeed**() and **setJointSpeed**() can be used to get and set one joint angular speed at a time, respectively. You can call the function multiple times to set and get joint speeds for different joints.

However, you can call the member function **getJointSpeeds**() once to obtain angular speeds for both joints of a Linkbot. The syntax of the member function **getJointSpeeds**() is as follows.

```
robot.getJointSpeeds(speed1, NaN, speed3);
```

The function uses two arguments `speed1` and `speed3` to store the joint angular speeds for joints 1 and 3, respectively.

Similarly, you can call the member function **setJointSpeeds**() once to set angular speeds for two joints of a Linkbot-I. The syntax of the member function **setJointSpeeds**() is as follows.

```
robot.setJointSpeeds(speed1, NaN, speed3);
```

The function has two arguments `speed1` and `speed3` to set the angular speeds for joints 1 and 3, respectively.

```
/* File: setjointspeeds.ch
   Get and set joint speeds. */
#include <linkbot.h>
CLinkbotI robot;
double speed1, speed3;  // the joint speeds

/* set the joint angular speed 75 degrees/second for all joints */
speed1 = 75;
speed3 = 75;
robot.setJointSpeeds(speed1, NaN, speed3);

/* move the joint 1 by 180 degrees and joint 3 by -180 degrees */
robot.move(180, NaN, -180);

/* get the joint angular speed for all joints */
robot.getJointSpeeds(speed1, NaN, speed3);
printf("Joint1 angular speed = %.2lf degrees per second\n", speed1);
printf("Joint3 angular speed = %.2lf degrees per second\n", speed3);
```

Program 10.10: Getting and setting joint speeds in degrees per second using **getJointSpeeds**() and **setJointSpeeds**().

For example, when Program 10.10 is executed, the following output will be displayed in the input/output pane.

```
Joint1 angular speed = 75.00 degrees per second
Joint3 angular speed = 75.00 degrees per second
```


Do Exercises 2 and 3 on page 186.

Problem Statement:
A Linkbot-I has two-wheels of different sizes. The radius for joint 1 is 1.75 inches whereas the radius for joint 3 is 2 inches. Write a program `twodiffwheels.ch` to move the robot for 4 seconds. The speed for joint 1 is 100 degrees per second and the speed for joint 3 is 87.5 degrees per second.

You may verify that with different joint angular speeds in the above problem statement, the robot will move in a straight line even though the radii of two wheels are different.

As the member function **driveTime**() uses the angular joint speed in positive values, Program 10.11 uses the statements below to drive the robot

```
    robot.setJointSpeeds(100, NaN, 87.5);
    robot.driveTime(4);
```

The program `twodiffwheels2.ch` use the statements

```
    robot.setJointSpeeds(100, NaN, -87,5);
    robot.moveTime(4);
```

to move the robot using the member function **moveTime**().

```
/* File: twodiffwheels.ch
   Drive a robot with different wheel sizes in a straight line */
#include <linkbot.h>
CLinkbotI robot;
double speed1, speed3;  // the joint speeds

/* set joint speeds and drive the robot with a specified time  */
speed1 = 100;
speed3 = 87.5;
robot.setJointSpeeds(speed1, NaN, speed3);
robot.driveTime(4);
```

Program 10.11: Drive a robot with wheels of different sizes in a straight line.

In section 13.4, we will learn how to control individual joints to move a robot with two wheels of different sizes in a straight line. We will also record and plot joint or arc length versus time.

E Do Exercise 4 on page 186.

10.7.3 Get and Set a Joint Angular Speed Ratio

The joint angular speed of a Linkbot can be specified not only in degrees per second, but also in a ratio. A ***speed ratio*** of a joint is the percentage of the maximum joint speed. Its value ranges from -1 to 1. In other words, if the ratio is set to 0.5, the joint will turn at 50% of its maximum angular velocity while moving continuously or moving to a new goal position. The member function **getJointSpeedRatio**() can be used to obtain the speed ratio of a joint of a Linkbot-I. The syntax of the member function **getJointSpeedRatio**() is as follows.

```
    robot.getJointSpeedRatio(id, ratio);
```

The first argument `id` is for a joint number. It should be one of the enumerated values defined in Table 10.1. The second argument `ratio` should be a variable of **double** type. When the function is called, it will store the speed ratio of the joint in the range of -1 to 1.

The member function **setJointSpeedRatio**() can be used to set the speed ratio of a joint for a Linkbot. The syntax of the member function **setJointSpeedRatio**() is as follows.

```
    robot.setJointSpeedRatio(id, ratio);
```

The first argument `id` is for a joint number. It should be one of the enumerated values defined in Table 10.1. The second argument `ratio` is the speed ratio to be set for the joint. The minimum speed ratio for a joint of a Linkbot is -1. The maximum speed ratio for a joint of a Linkbot is 1.

```
/* File: setjointspeedratio.ch
   Get and set joint speed ratio. */
#include <linkbot.h>
CLinkbotI robot;
double ratio;     // joint angular speed ratio
double speed;  // the joint angular speed in degrees per second

/* get the joint speed ratio for joint 1 */
robot.getJointSpeedRatio(JOINT1, ratio);
printf("Joint1 angular speed ratio = %lf\n", ratio);

/* set the joint speed ratio for joint 1 to 0.75 (75% of the max speed).  */
ratio = 0.75;
robot.setJointSpeedRatio(JOINT1, ratio);

/* rotate joint 1 by 180 degrees */
robot.moveJoint(JOINT1, 180);

/* get the joint speed ratio for joint 1 */
robot.getJointSpeedRatio(JOINT1, ratio);
printf("Joint1 angular speed ratio = %lf\n", ratio);

/* get the joint speed for joint 1 */
robot.getJointSpeed(JOINT1, speed);
printf("Joint1 angular speed = %lf degrees per second\n", speed);
```

Program 10.12: Getting and setting a joint speed ratio using **getJointSpeedRatio()** and **setJointSpeedRatio()**.

For example, when Program 10.12 is executed, the following output will be displayed in the input/output pane.

```
Joint1 angular speed ratio = 0.375000
Joint1 angular speed ratio = 0.750000
Joint1 angular speed = 90.000000 degrees per second
```

E Do Exercise 5 on page 187.

10.7.4 Get and Set Joint Speed Ratios

The member functions **getJointSpeedRatio**() and **setJointSpeedRatio**() can be used to get and set one speed ratio for a joint at a time, respectively. You can call the function multiple times to set and get speed ratios for different joints.

However, you can call the member function **getJointSpeedRatios**() once to obtain speed ratios for two joints of a Linkbot-I. The syntax of the member function **getJointSpeedRatios**() is as follows.

```
robot.getJointSpeedRatios(ratio1, NaN, ratio3);
```

The function uses two arguments `ratio1` and `ratio3` to store the joint speeds for joints 1 and 3, respectively.

Similarly, you can call the member function **setJointSpeedRatios**() once to set speed ratios for two joints of a Linkbot. The syntax of the member function **setJointSpeedRatios**() is as follows.

```
robot.setJointSpeedRatios(ratio1, NaN, ratio3);
```

The function uses two arguments `ratio1` and `ratio3` to set the angular speeds for joints 1 and 3, respectively.

```
/* File: setjointspeedratios.ch
   Get and set joint speed ratios for all joints. */
#include <linkbot.h>
CLinkbotI robot;
double ratio1, ratio3;

/* set the joint angular speed ratio for all joints to 0.75 */
ratio1 = 0.75;
ratio3 = 0.75;
robot.setJointSpeedRatios(ratio1, NaN, ratio3);

/* rotate joints 1 and 3 by 180 degrees */
robot.move(180, NaN, 180);

/* get the joint speed ratio for all joints */
robot.getJointSpeedRatios(ratio1, NaN, ratio3);
printf("Joint1 angular speed ratio = %.2lf\n", ratio1);
printf("Joint3 angular speed ratio = %.2lf\n", ratio3);
```

Program 10.13: Getting and setting joint speed ratios using **getJointSpeedRatios()** and **setJointSpeedRatios()**.

For example, when Program 10.13 is executed, the following output will be displayed in the input/output pane.

```
    Joint1 speed ratio = 0.75
    Joint3 speed ratio = 0.75
```

E Do Exercises 6 and 7 on page 187.

10.7.5 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.getJointSpeed(JOINT1, speed);
   ```

   to get the joint angular speed in the second argument for a specified joint in the first argument.

2. Call the **CLinkbotI** member function

   ```
   robot.setJointSpeed(JOINT1, speed);
   ```

   to set the joint angular speed in the second argument for a specified joint in the first argument.

3. Call the **CLinkbotI** member function

   ```
   robot.getJointSpeeds(speed1, NaN, speed3);
   ```

   to get the joint angular speeds for two joints of a Linkbot-I.

4. Call the **CLinkbotI** member function

   ```
   robot.setJointSpeeds(speed1, NaN, speed3);
   ```

to set the joint angular speeds for two joints of a Linkbot-I.

5. Call the **CLinkbotI** member function

```
robot.getJointSpeedRatio(JOINT1, ratio);
```

to get the speed ratio in the second argument for a specified joint in the first argument.

6. Call the **CLinkbotI** member function

```
robot.setJointSpeedRatio(JOINT1, ratio);
```

to set the speed ratio in the second argument for a specified joint in the first argument.

7. Call the **CLinkbotI** member function

```
robot.getJointSpeedRatios(ratio1, NaN, ratio3);
```

to get the speed ratios for two joints of a Linkbot-I.

8. Call the **CLinkbotI** member function

```
robot.setJointSpeedRatios(ratio1, NaN, ratio3);
```

to set the speed ratios for two joints of a Linkbot-I.

10.7.6 Terminology

angular speed, speed ratio, **robot.getJointSpeed**(), **robot.setJointSpeed**(), **robot.getJointSpeeds**(), **robot.setJointSpeeds**(), **robot.getJointSpeedRatio**(), **robot.setJointSpeedRatio**(), **robot.getJointSpeedRatios**(), **robot.setJointSpeedRatios**().

10.7.7 Exercises

1. Write a program `setjointspeed2.ch` to move joint 1 by 360 degrees at the speed of 45 degrees per second, then move joint 1 by another 360 degrees at the speed of 90 degrees per second. At the end of the program, get the joint speed for joint 1 using the member function **getJointSpeed**().

2. Write a program `setjointspeeds2.ch` to move joint 1 by 360 degrees and joint 3 by -360 degrees at the speed of 45 degrees per second. then move joint 1 by 360 degrees and joint 3 by -360 degrees at the speed of 90 degrees per second. At the end of the program, get the joint speeds for joints 1 and 3 using the member function **getJointSpeed**().

3. Write a program `setjointspeeds3.ch` to set the joint speed for joint 1 to 45 degrees per second and the joint speed for joint 3 to 90 degrees per second. Then, call the function **moveTo**() as follows:

```
robot.moveTo(360, NaN, -360);
```

What motion would occur? Why?

4. A Linkbot-I has two-wheels of different sizes. The radus for joint 1 is 1.75 inches whereas the radus for joint 3 is 2 inches. Write a program `twodiffwheels.ch` to move the robot for 12 seconds. The speed for joint 1 is 50 degrees per second and the speed for joint 3 is 43.75 degrees per second.

5. Write a program `setjointspeedratio2.ch` to move joint 1 with two full rotations at the maximum speed set by the member function **setJointSpeedRatio()**. Then move joint 1 backward one full rotation with a speed ratio of 0.25. At the end of the program, get the joint speed ratio for joint 1 using the member function **getJointSpeedRatio**(), and get the joint speed for joint 1 using the member function **getJointSpeed**().

6. Write a program `setjointspeedratios2.ch` to move the robot forward with two full rotations for both joints 1 and 3 at the maximum speed set by the member function **setJointSpeedRatios()**. Then move the robot backward one full rotation for joints 1 and 3 with a speed ratio of 0.25. At the end of the program, get the joint speed ratios for joints 1 and 3 using the member function **getJointSpeedRatios**().

7. What are the four different ways to set joint speeds? Which functions would you use to change speeds for multiple joints at once?

10.8 ‡ Convert Units of Angles between Degrees and Radians

In some applications, joint angles are specified in radians, instead of degrees. However, the member functions for movements such as **move()** expect input angles in degrees. In this case, angles in radians must first be converted to degrees, then passed to the movement functions. Similarly, a joint speed might be specified in radian per second, instead of degrees per second. The angular speed needs to be converted to degrees per second before it is used in the function **setJointSpeed**().

The function **radian2degree**() or **rad2deg**() can be used to convert an angle from radians to degrees with the following syntax.

```
degree = radian2degree(radian);
```

The function **radian2degree**() or **rad2deg**() takes an angle in radians as its argument and returns the angle in degrees. The function can also be used to convert a joint speed from radians per second to degrees per second. The function is implemented in Ch with the code

```
double radian2degree(double radian)
{
    return radian * 180 / M_PI;
}
```

The symbol **M_PI** has the value of π. How to write a function is beyond the scope of this book.

If desired, values in radians can be converted to degrees using the counterpart function, **degree2radian**() or **deg2rad**()

```
radian = degree2radian(degree);
```

The function is implemented as follows.

```
double degree2radian(double degree)
{
    return degree * M_PI / 180;
}
```

```
/* File: radian.ch
   Move the robot joint 1 with speed specified in radians per second. */
#include <linkbot.h>
CLinkbotI robot;
double angle;
double speed;

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

/* set the speed specified in radians per second */
speed = 1.5;            // speed in 1.5 radians per second
speed = radian2degree(speed); // convert the speed to degrees per second
robot.setJointSpeed(JOINT1, speed);

/* rotate joint 1 by 90 degrees */
robot.moveJoint(JOINT1, 90);

/* get the joint angle in degrees, display in both degrees and radians */
robot.getJointAngle(JOINT1, angle);
printf("Joint1 angle = %.2lf degrees.\n", angle);
angle = degree2radian(angle);  // convert the angle in degrees to radian
printf("Joint1 angle = %.2lf radians.\n", angle);
```

Program 10.14: Converting units of angles between degrees and radians using **degree2radian**() and **radian2degree**().

For example, Program 10.14 moves 90 degrees for joint 1 at the speed of 1.5 radian per second. After joint 1 finishes its movement, the joint angle for joint 1 is obtained and displayed in both degrees and radians. The function calls

```
    /* set the speed specified in radians per second */
    speed = 1.5;                              // speed in 1.5 radians per second
    speed = radian2degree(speed); // convert the speed to degrees per second
    robot.setJointSpeed(JOINT1, speed); // set joint in degree per second
```

set the joint speed for joint 1 to 1.5 radians per second. The function calls

```
    /* get the joint angle in degrees, display in both degrees and radians */
    robot.getJointAngle(JOINT1, angle);
    printf("Joint1 angle = %.2lf degrees.\n", angle);
    angle = degree2radian(angle);  // convert the angle in degrees to radian
    printf("Joint1 angle = %.2lf radians.\n", angle);
```

get the joint angle for joint 1 in degrees, display it in both degrees and radians. When Program 10.14 is executed, the following output will be displayed in the input/output pane.

```
    Joint1 angle = 89.88 degrees.
    Joint1 angle = 1.56 radians.
```

E Do Exercise 1 on page 189.

10.8.1 Summary

1. Call the function **radian2degree**() to convert an angle from radians to degrees.

```
    degree = radian2degree(radian);
```

2. Call the function **degree2radian**() to convert an angle from degrees to radians.

```
radian = degree2radian(degree);
```

10.8.2 Terminology

radian, **degree2radian**(), **radian2degree**().

10.8.3 Exercises

1. Write a program `radian2.ch` to set the joint speeds of both faceplates of a Linkbot to 0.5 radian per second and move the Linkbot's two faceplates one full rotation. Then, use the member function **getJointAngle**() to obtain the joint angle for joint 1. Display the joint angle in degrees and radians.

CHAPTER 11

Writing Advanced Programs to Control a Single Linkbot

In the previous chapters, we learned the member functions for controlling a Linkbot with different characteristics. In this chapter, we will learn advanced features of the Linkbot and additional member functions to control the Linkbot for various other applications.

11.1 Plot Recorded Joint Angles and Time

As we learned in Section 8.3.1, we can record and plot distance values versus time for a Linkbot-I. We can also record and plot joint angle values versus time for a Linkbot. To record joint angle data, we use the two **CLinkbotI** member functions **recordAngleBegin()** and **recordAngleEnd()**.

Program 11.1 demonstrates how to record and plot real time joint angle data for a Linkbot-I. This plot can be used to show the angle of a Linkbot-I joint at a specific point in time.

Problem Statement:
Write a program `recordanglescattern.ch` to rotate joint 1 of a Linkbot for 360 degrees at 90 degrees per second. The radius of each wheel is 1.75 inches. Record the joint angle for joint 1 with a time interval of 0.1 second. Plot the joint angle versus time.

```
/* File: recordanglescatten.ch
   Record a joint angle and time, plot the acquired data */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed = 90;          // speed in 90 degrees/seconds
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, angledata; // recorded time and angles for joint 1
CPlot plot;                 // plotting class

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

/* set the joint 1 speed */
robot.setJointSpeed(JOINT1, speed);

/* begin recording time and angle */
robot.recordAngleBegin(JOINT1, timedata, angledata, timeInterval);

/* move joint 1 for 360 degrees */
robot.moveJoint(JOINT1, 360);

/* end recording time and angle */
robot.recordAngleEnd(JOINT1, numDataPoints);

/* plot the data */
plot.title("Angles for joint 1 versus time");
plot.label(PLOT_AXIS_X, "time (seconds)");
plot.label(PLOT_AXIS_Y, "angle for joint1 (degrees)");
plot.scattern(timedata, angledata, numDataPoints);
plot.plotting();
```

Program 11.1: Recording a joint angle using the function **recordAngleBegin**() and **recordAngleEnd**() to generate a scatter plot.

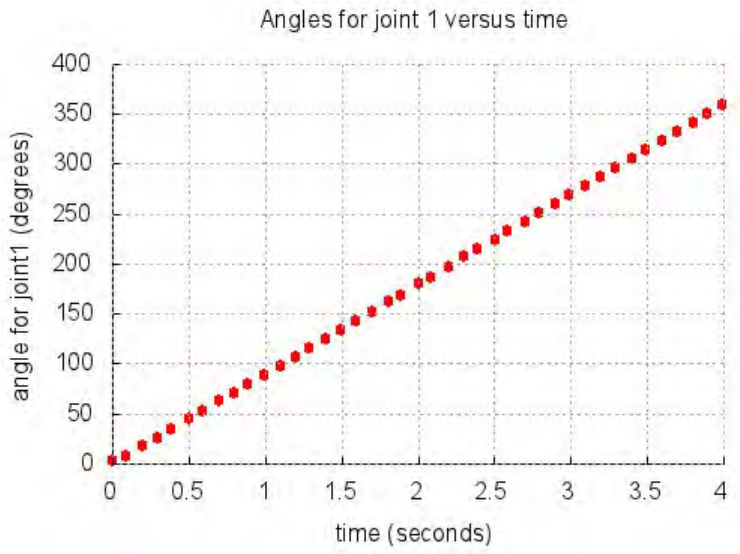


Figure 11.1: The scatter plot for the joint angle versus time from Program 11.1.

Program 11.1 records the angle of joint 1 of a Linkbot-I every 0.1 seconds. It then uses the **CPlot** member functions demonstrated in Program 8.7 to plot the recorded joint angles versus time for this Linkbot. In order to collect and plot the angle data from the Linkbot-I, we need to declare a few extra variables in the program. The line

```
double timeInterval = 0.1;  //time interval in 0.1 second
```

specifies that we want to measure the angle every 0.1 second after the robot starts to move. The next line

```
int numDataPoints;  //number of data points recorded
```

declares an **int** variable to keep hold the number of data points the program has collected. The following line

```
robotRecordData_t timedata, angledata;   //recorded time and angles for joint 1
```

creates two variables of type **robotRecordData_t**. This is a special type of array defined in the header file **linkbot.h** that can be used when we don't know ahead of time how many values we will be recording. Variables of type **robotRecordData_t** dynamically grow to the size that you need while the program is running. These two variables, `timedata` and `angledata`, will be used to keep track of the data points while the Linkbot-I is in motion.

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.

To record angle data while the robot is moving, we need to use two new **CLinkbotI** member functions **recordAngleBegin**() and **recordAngleEnd**(). The general syntax of the **CLinkbotI** member function **recordAngleBegin**() is as follows

```
robot.recordAngleBegin(id, timedata, angledata, timeInterval);
```

The argument `id` specifies the joint from which you wish to record data. The argument `timedata` is used to record time values in seconds elapsed since the Linkbot-I starts to move, and argument `angledata` is used to record the angle values in degrees. The argument `timeInterval` specifies the amount of time between each recorded measurement. In Program 11.1, the line

```
robot.recordAngleBegin(JOINT1, timedata, angledata, timeInterval);
```

starts the process of recording the angle of joint 1 every 0.1 seconds. It is important to include this statement before the Linkbot-I starts moving, so that the program does not miss any data points.

After the Linkbot-I has stopped moving, we must use **recordAngleEnd()** to stop the recording of data. The general syntax of the function **recordAngleEnd()** is as follows

```
robot.recordAngleEnd(id, numDataPoints);
```

The argument `id` is the joint you wish to stop recording, and the argument `numDataPoints` keeps track of the total number of data points that were collected while the Linkbot-I was moving.

In Program 11.1, after the Linkbot-I has finished moving forward 360 degrees, the line

```
robot.recordAngleEnd(JOINT1, numDataPoints);
```

stops the recording of data after the Linkbot-I has stopped moving forward. `numDataPoints` will now hold the total number of data points that were collected while the Linkbot-I was in motion. In addition, `angledata` will hold all the angle values that were recorded while the Linkbot-I was in motion, and `timedata` will hold all the corresponding time values.

The aquired data are graphed as a scatter plot as shown in Figure 11.1 by the function call

```
plot.scattern(timedata, angledata, numDataPoints);
```

If we change the above statement in Program 11.1 to

```
plot.data2DCurve(timedata, angledata, numDataPoints);
```

for a line plot. The plot generated by such a program `recordangle.ch` is shown in Figure 11.2.

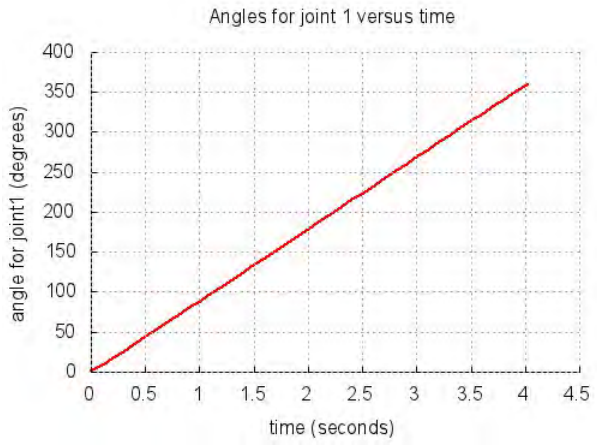


Figure 11.2: The line plot for the joint angle versus time from the program `recordangle.ch`.

When a joint rotates at 90 degrees per second, the relation between the joint angle ($angle$) and time (t) in Figures 11.1 and 11.2 can be formulated by the following linear equation.

$$angle = 90t \tag{11.1}$$

If we change the statement

```
robot.moveJoint(JOINT1, 360);
```

in Program 11.1 to

```
robot.moveJoint(JOINT1, -360);
```

we rotate the joint 1 in the opposite direction. The plot generated by such a program `recordangleneg.ch` is shown in Figure 11.3. The relation between the joint angle (s) and time (t) in Figure 11.3 can be formulated by the following linear equation.

$$angle = -90t \tag{11.2}$$

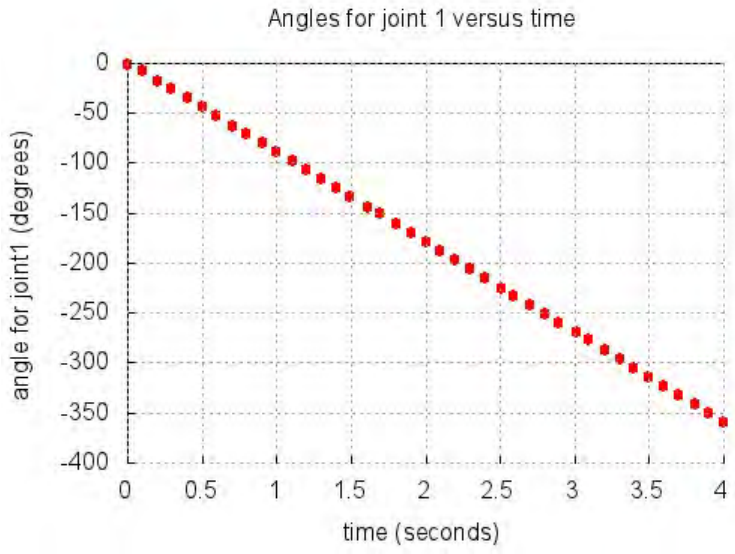


Figure 11.3: The scatter plot for the joint angle versus time for a Linkbot moving in the negative direction.

E Do Exercises 2 and 3 on page 195.

11.1.1 Summary

1. Use the data type **robotRecordData_t** to record data points when you don't know ahead of time how many points you will get.

2. Call the **CLinkbotI** member function

```
robot.recordAngleBegin(id, timedata, angledata, timeInterval);
```

when you want to begin recording joint angle data from a Linkbot-I.

3. Call the **CLinkbotI** member function

```
robot.recordAngleEnd(id, numDataPoints);
```

when you want to stop recording joint angle data from a Linkbot-I.

11.1.2 Terminology

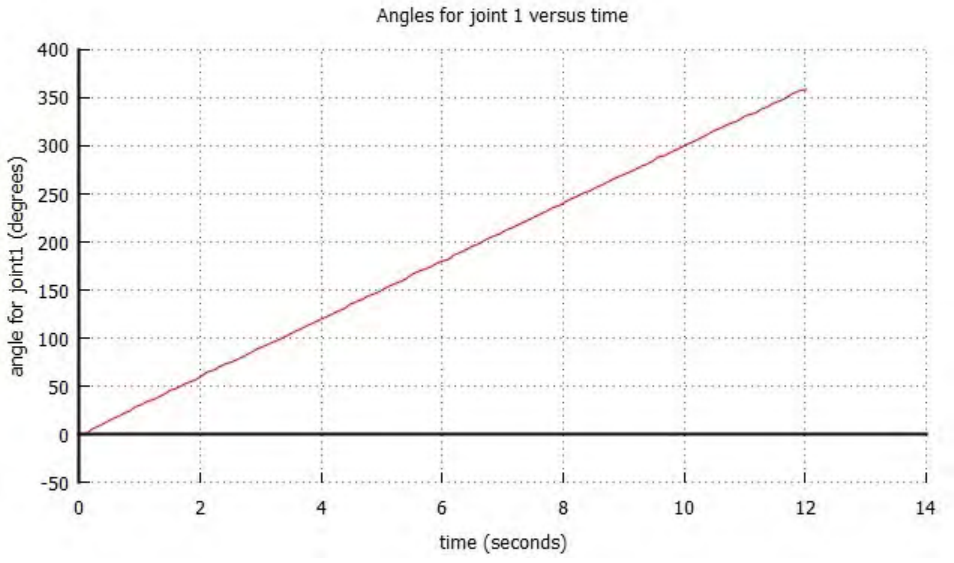
robotRecordData_t, **robot.recordAngleBegin()**, **robot.recordAngleEnd()**.

11.1.3 Exercises

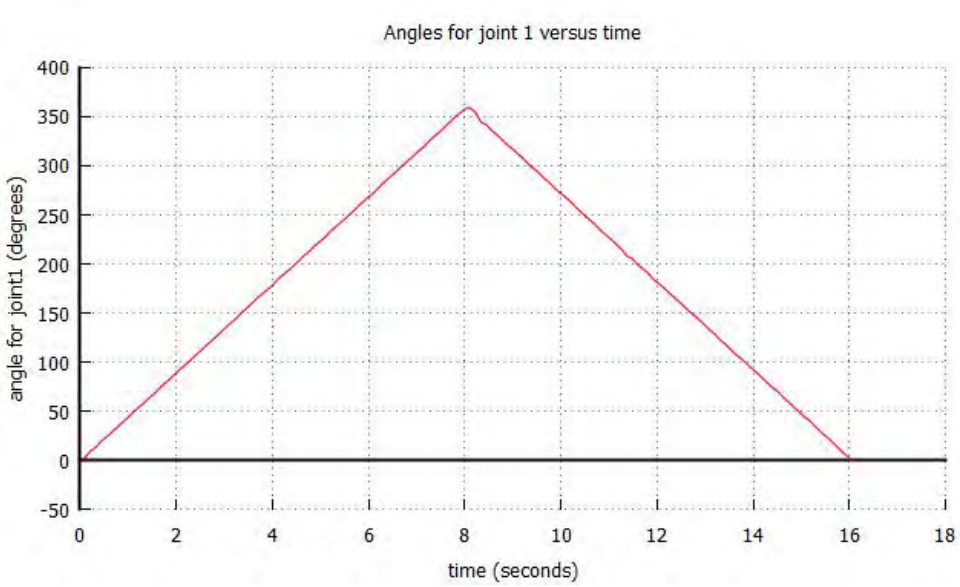
1. Watch the following video tutorial for Ch Linkbot Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Control a Robot with Different Wheels

2. Write a program `recordangle2.ch` to record the joint angle for joint 1 of a Linkbot with a time interval of 0.1 second while the Linkbot joint 1 rolls forward for 360 degrees at 30 degrees per second. Plot the joint angle versus time as shown in the figure below in a line plot. What is the equation for the linear relation shown in the figure? Also write a program `recordanglescattern2.ch` to plot the joint angle versus time in a scatter plot.



3. Write a program `recordangle3.ch` to record the joint angle for joint 1 of a Linkbot-I with a time interval of 0.1 second when the robot drives forward by rotating the joint 1 by 360 degrees and drives backward by 360 degrees. Plot the joint angle versus time as shown in the figure below.



11.2 Move Joints with a Specified Time

Sometimes it may be necessary to move joints of a robot for a specified time. The member function **moveTime**() can be used for such a purpose. The general syntax of this function is

```
    robot.moveTime(seconds);
```

The argument, `seconds`, defines how long each joint will be moved in seconds. The direction of the motion for each joint is determined by the speed of the joint. If the speed is positive, the joint will move in the counterclockwise direction. If the speed is negative, the joint will move in the clockwise direction.

```
/* File: movetime.ch
   Move a robot for a specified time. */
#include <linkbot.h>
CLinkbotI robot;
double time1=5;   // five seconds
double speed=90; // 90 degrees per second

/* set the joint speeds for all joints to 90 degrees per second */
robot.setJointSpeeds(speed, NaN, speed);

/* rotate joints 1 and 3 for the specified 'time1', based on the speed */
robot.moveTime(time1);
```

Program 11.2: Moving a joint with a specified time using **moveTime**().

Program 11.2 sets the speeds of joints 1 and 3 to 90 degrees per second using the function **setJointSpeeds**(), which was introduced in Program 10.10.

The line

```
    robot.moveTime(time);
```

moves a Linkbot-I for 5 seconds at the speed set by **setJointSpeeds**(). Note that if the speeds for both joints are positive, the robot will spin and not drive forward.

 Do Exercise 1 on page 198.

It is also possible to move only one joint, using the **CLinkbotI** member function **moveJointTime()**. The general syntax of this function is

```
robot.moveJointTime(id, seconds);
```

The first argument `id` specifies which joint you want to move. The second argument, `seconds`, indicates how long the joint will move in seconds.

```
/* File: movejointtime.ch
   Move a joint in a specified time. */
#include <linkbot.h>
CLinkbotI robot;
double time1=5;   // five seconds
double speed=90; // 90 degrees per second

/* set the joint speed for joint 1 to 90 degrees per second */
robot.setJointSpeed(JOINT1, speed);

/* rotate joint 1 for the specified 'time1' */
robot.moveJointTime(JOINT1, time1);
```

Program 11.3: Moving a joint with a specified time using **moveJointTime()**.

Program 11.3 works the same way as Program 11.2 except that the last line

```
robot.moveJointTime(JOINT1, time);
```

moves only joint 1 for 5 seconds, at 90 degrees per second.

In the program `movejointtimeneg.ch` distributed in C-STEM Studio, the joint speed is changed to -90 degrees per second by the statement

```
robot.setJointSpeed(JOINT1, -speed);
```

joint 1 of the robot will move in clockwise direction instead of counterclockwise direction.

 Do Exercise 2 3 on page 198.

11.2.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.moveTime(seconds);
   ```

   to move joints for a number of seconds indicated by the argument `seconds`.

2. Call the **CLinkbotI** member function

   ```
   robot.moveJointTime(id, seconds);
   ```

   to move a joint specified by the argument `id` for the number of seconds indicated by the argument `seconds`.

11.2.2 Terminology

robot.moveTime(), **robot.moveJointTime()**.

11.2.3 Exercises

1. Write a program `movetime2.ch` to rotate joints 1 and 3 of a Linkbot-I in opposite directions at the speed of 45 degrees per second for 8 seconds. What is the resulting motion?

2. Write a program `movejointtime2.ch` to rotate joint 1 in the positive direction (counter clockwise) at the speed of 45 degrees per second for 8 seconds.

3. Write a program `movejointtimeneg2.ch` to rotate joint 1 in the negative direction (clockwise) at the speed of 45 degrees per second for 8 seconds.

11.3 Hold the Joints At Exit

By default, when the program finishes, all joints are relaxed. When joints are relaxed, depending on how the Linkbot is positioned, the joint angles can be different after the program finishes its execution. In some applications, especially for a robot configured with multiple Linkbots, you may want to hold the pose of the robot after the program exits. You can hold all joint angles of a Linkbot at the exit of a program by calling the function **holdJointsAtExit()**. The general syntax of this function is

```
robot.holdJointsAtExit();
```

```
/* File: holdjointsatexit.ch
   Hold the joints at the exit. */
#include <linkbot.h>
CLinkbotI robot;
double angle=360;    // angle for rotation

/* drive forward for 'angle' */
robot.driveAngle(angle);

/* Hold the joints at exit .*/
robot.holdJointsAtExit();
```

Program 11.4: Holding the joints after the exit using the function **holdJointsAtExit()**.

Program 11.4 moves a Linkbot-I forward 360 degrees. The last line

```
robot.holdJointsAtExit();
```

holds the position of the Linkbot's motors in its final position.

E Do Exercises 1 and 2 on page 199.

11.3.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.holdJointsAtExit();
   ```

   to hold the final position of a Linkbot-I.

11.3.2 Terminology

robot.holdJointsAtExit().

11.3.3 Exercises

1. Write a program `holdjointsatexit2.ch` to rotate joint 3 to 45 degrees (without holding the joints at the exit). After the program is run, try to rotate joint 3 manually.

2. Modify the program `holdjointsatexit2.ch` in Exercise 1 as a new program `holdjointsatexit3.ch`, which holds all the joint at exit. After the program is run, try to rotate joint 3 manually.

11.4 Measure the Clock Time for Moving a Joint Using the Member Function systemTime()

In Program 8.5 we measured how long it took for a Linkbot-I to complete a motion of a specified distance . The member function **systemTime**() can be used in many other applications. One example is measuring the time it takes for a joint to complete its movement with a specified joint speed and angle.

Program 11.5 gives an example of using **systemTime**() to time a Linkbot's motion for a specified joint angle and joint speed.

> **Problem Statement:**
> The joint 1 of a Linkbot rotates two full rotations for 720 degrees at the speed of 90 degrees per second. Write a program `time.ch` to measure the time for the Linkbot to complete the motion.

```
/* File: time.ch
   Get the time for a robot to finish its motion */
#include <linkbot.h>
CLinkbotI robot;
double speed = 90;          // speed in degrees/second
double angle = 720;         // joint angle
double time1, time2, elapsedtime;    // system time and elapsed time

/*  set the joint speed for joint 1  */
robot.setJointSpeed(JOINT1, speed);

robot.systemTime(time1);        // get the system time since the system starts
robot.moveJoint(JOINT1, angle); // rotate joint 1 by 'angle'
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", angle/speed);
```

Program 11.5: Get the time for a Linkbot to complete a motion with specified joint angle and joint speed using the member function **systemTime**().

Program 11.5 declares three variables of type **double**: `time1`, `time2`, and `elapsedtime`, which will be used to time the motion of the Linkbot-I. The line

```
robot.systemTime(time1);     // the system time since the system starts
```

records the system time in seconds immediately before joint 1 of the Linkbot-I starts moving. Then the line

```
robot.systemTime(time2);     // the system time since the system starts
```

records the system time in seconds immediately after joint 1 of the Linkbot-I stops moving. The next line

```
elapsedtime = time2 - time1; // Calculate the time for the motion.
```

calculates the difference between the end time and the start time. This difference gives the total actual time joint 1 of the Linkbot-I was in motion.

The last two lines of the program display the actual `elapsedtime` that was calculated using **systemTime**(), as well as the theoretical time. The theoretical time of a Linkbot-I's motion as a function of joint angle and joint speed is defined as

$$theoretical\ time = \frac{(joint\ angle\ in\ degrees)}{(joint\ speed\ in\ degrees\ per\ second)}$$

When Program 11.5 is executed, the following output will be displayed in the input/output pane

```
The motion for the Linkbot took 8.14 seconds.
The motion should take 8.00 seconds in theory.
```


Do Exercises 1 and 2 on page 200.

11.4.1 Summary

1. Measure the clock time taking to move a joint.

11.4.2 Terminology

actual time, theoretical time.

11.4.3 Exercises

1. Write a program `time2.ch` to drive forward for 450 degrees at the speed of 45 degrees per second. Use the member function **systemTime**() to measure the time for the Linkbot to complete its motion.

2. Based on program `movejointtime2.ch` developed in Exercise 2 on page 198, write a program `time3.ch` to drive joints 1 and 3 forward at the speed of 45 degrees per second for 8 seconds using the function **moveTime()**. Use the member function **systemTime**() to measure the time for the Linkbot to complete its motion.

CHAPTER 12

Sensory Information for a Linkbot

In this chapter, we will learn how to control the settings of various sensors on a Linkbot. We will also learn how to retrieve sensory input from a single Linkbot. The types of sensory information available to a Linkbot pertain to the LED, buzzer, accelerometer, and remaining battery charge. As was mentioned in Section 11.1 the ability to acquire sensory information in real time makes it possible for a robot to interact with its environment or with human users.

12.1 Set and Get the LED Color by Name Using the Data Type string_t

In Section 3.5 we learned how to set the LED color using a color name. In this section we will learn how to also get the LED color of a Linkbot using a color name. The ability to change the LED color with a computer program opens up the possibility of adding visual effects to a Linkbot choreography. Program 12.1 gives an example of how to set and get the LED color of a Linkbot using the member functions **setLEDColor()** and **getLEDColor()**.

```
/* File: color.ch
   Set and get the LED color by name */
#include <linkbot.h>
CLinkbotI robot;
string_t color;

robot.setLEDColor("magenta");
robot.delaySeconds(2);
robot.setLEDColor("green");
robot.delaySeconds(2);
robot.setLEDColor("blue");
robot.delaySeconds(2);
robot.setLEDColor("purple");
robot.delaySeconds(2);
robot.setLEDColor("aqua");
robot.delaySeconds(2);
robot.setLEDColor("orange");
robot.delaySeconds(2);
robot.setLEDColor("pink");     // pink
robot.delaySeconds(2);
robot.setLEDColor("hotPink");  // hot pink
robot.delaySeconds(2);
robot.setLEDColor("deepPink"); // deep pink

robot.getLEDColor(color);
printf("The current LED color: %s\n", color);
```

Program 12.1: Setting and getting the LED color for a Linkbot-I using a color name.

Program 12.1 uses the CLinkbotI member function **setLEDColor()**, which was introduced in Section 3.5, to change the LED color using a color name. The line

```
    string_t color;
```

declares a variable of string_t type, which is used to retrieve the name of the LED color. string_t is a special data type used in Ch for strings. A string_t data type consists of any combination of letters or numbers, surrounded by quotation marks. For example, the line

```
    robot.setLEDColor("red");
```

uses a string_t data type with the value "red" as the argument to set the LED color to red. The next line

```
    robot.delaySeconds(2);
```

pauses the program in order to allow the LED to shine red for two seconds. The following lines

```
    robot.setLEDColor("green");
    robot.delaySeconds(2);
```

changes the LED color to green for a duration of two seconds. Program 12.1 continues to use **setLEDColor()** to change the LED color to blue, purple, aqua, orange, then three different shades of pink. The first seven colors match the default colors for lines and points in the Ch plot. As was mentioned in Section 3.5, Appendix C lists the 139 color names available for changing the LED color of a Linkbot. If a name other than those listed in Appendix C is used, **setLEDColor()** will print an error message to the input/output panel of ChIDE.

It is also possible to get the name of the LED color. The CLinkbotI member function **getLEDColor()** is used to retrieve the name of the current LED color. Only 137 color names are available for use with **getLEDColor()**, and there are many more than 137 colors in existence. Therefore whenever the current

LED color is not a color that is listed in Appendix C, the member function **getLEDColor()** finds and returns the name of the color closest to it. In Program 12.1, the line

```
robot.getLEDColor(color);
```

passes the name of the most recent LED color through its argument of the member function **getLEDColor**(). Since the most recent color set by **setLEDColor()** is deep pink, the value of the argument `color` will be "deepPink" after **getLEDColor()** is called. The final line

```
printf("The current LED color: %s\n", color);
```

prints the following message to the input/output pane:

```
The current LED color: deepPink
```

The conversion specifier `"%s"` is used to print the value of the variable `color` of the string type.


Do Exercise 1 on page 203.

12.1.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.getLEDColor(color);
   ```

   to get the LED color for a Linkbot.

12.1.2 Terminology

robot.getLEDColor().

12.1.3 Exercises

1. Write a program `color2.ch` to command a Linkbot-I in the following sequence. Change the LED color to green, move the Linkbot forward 360 degrees, change the LED color to yellow, wait for 1 second, move the Linkbot backward 360 degrees, and finally change the LED color to red. Obtain the current color of the LED.

12.2 Set and Get the LED Color by RGB Values

In Section 12.1 we learned how to set and get the LED color of a Linkbot by using a color name. In this section we will learn how to change the LED color of a Linkbot using RGB values. We will also learn how to receive data from the Linkbot about the red, green, and blue values of the LED color currently displayed.

A Linkbot uses the RGB color model to control the color of its LED. The ***RGB color model*** is an additive color model, in which the three primary light colors -red, green, and blue- are added together in various ways to produce a broad array of colors. The RGB color model is commonly used in cameras, television, and computer monitors to detect, store, and display images. Generally, each of the three primary colors, red, green, and blue, can vary in intensity. The possible values for the intensity of each color range from 0, which is an absence of a primary color, to 255, which is the full intensity of a primary color. Figure 12.1 shows the different ways the primary colors can combine to make the secondary colors as well as white. The values of the intensities of the red, green, and blue color components is listed by each color. A few of the rules for RGB values are as follows

1. If all RGB values are equal, then the color is a gray tone.
2. If all RGB values are 0, the color is black (an absence of light).
3. If all RGB values are 255, the color is white.

The **CLinkbotI** class uses the member function **setLEDColorRGB()** to change the LED color of a Linkbot. This member function has the following general syntax

```
robot.setLEDColorRGB(r, g, b);
```

The argument `r` indicates the amount of red color you want the LED to have, and can be any integer value from 0 to 255. This range of integer values also applies to the arguments `g` for green and `b` for blue. The **CLinkbotI** class uses the member function **getLEDColorRGB()** to retrieve the RGB values of the current LED color. This function has the general syntax

```
robot.getLEDColorRGB(r, g, b);
```

where the arguments `r`, `g`, and `b` return the current intensity values of the primary color components. Program 12.2 shows how to set and get the LED color of a Linkbot-I using RGB values.

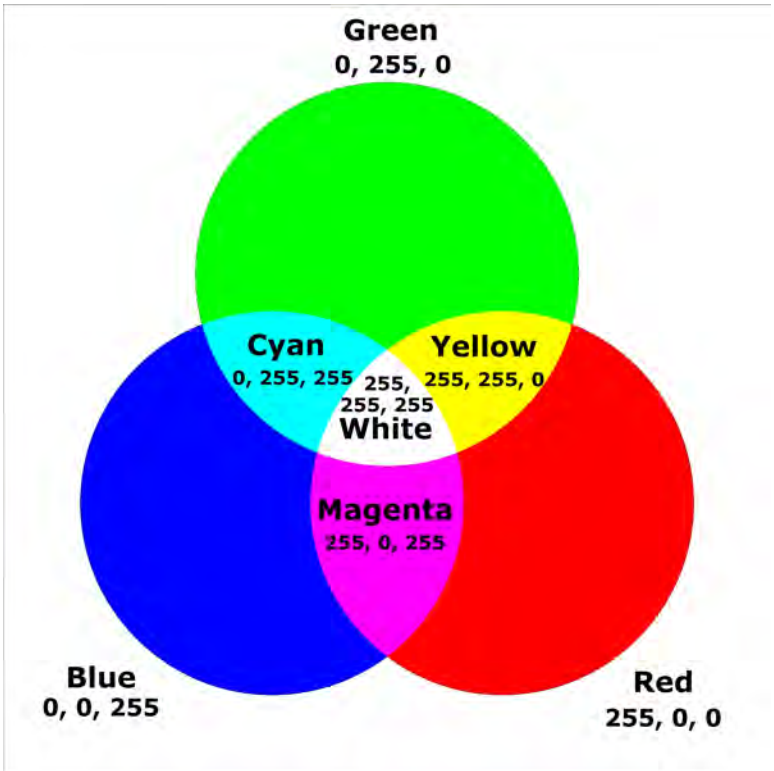


Figure 12.1: The relationship between colors in the RGB color model.

```
/* File: colorrgb.ch
   Set and get the LED color by RGB */
#include <linkbot.h>
CLinkbotI robot;
int r, g, b;

robot.setLEDColorRGB(255, 0, 0);       // red
robot.delaySeconds(2);
robot.setLEDColorRGB(0, 255, 0);       // green
robot.delaySeconds(2);
robot.setLEDColorRGB(0, 0, 255);       // blue
robot.delaySeconds(2);
robot.setLEDColorRGB(255, 255, 0);     // yellow
robot.delaySeconds(2);
robot.setLEDColorRGB(160, 32, 240);    // purple
robot.delaySeconds(2);
robot.setLEDColorRGB(255, 192, 203);   // pink
robot.delaySeconds(2);
robot.setLEDColorRGB(255, 105, 180);   // hot pink
robot.delaySeconds(2);
robot.setLEDColorRGB(255, 20, 147);    // deep pink

robot.getLEDColorRGB(r, g, b);
printf("The RGB value for the current LED color: %d %d %d\n", r, g, b);
```

Program 12.2: Setting and getting the LED color for a Linkbot-I using RGB values.

Program 12.2 is similar to Program 12.1 in that it shows how to set and get the LED color of a Linkbot-I. In Program 12.2, however, setting and getting the LED color is done using the RGB color model. The first line of the program

```
    int r, g, b;
```

declares three **int** type variables `r`, `g`, and `b` to hold the red, green, and blue color values of the LED. These variables will be used to retrieve the LED color values from the Linkbot-I. After the variable `robot` is connected to a Linkbot-I, the following line

```
    robot.setLEDColorRGB(255, 0, 0);    // red
```

sets the LED color to red using the **CLinkbotI** member function **setLEDColorRGB()**. Since this function is first used in Program 12.2 to set the LED to red, the value of the argument `r` is 255. This gives the full intensity of the red color component. Since no green or blue is needed in the color mix, the arguments `g` and `b` are both 0, specifying an absence of these colors. After the LED is set to red, the next line

```
    robot.delaySeconds(2);
```

pauses the program for 2 seconds. This controls the length of time that the LED continues to shine red. The following line

```
    robot.setLEDColorRGB(0, 255, 0);    // green
```

changes the LED to green by setting the argument `g` to 255 while setting both the arguments `r` and `b` to 0. This gives the green color component its full intensity while red and blue have zero intensity. The program is then paused for 2 seconds using the member function **delaySeconds**(), to allow the LED to shine green until the next color change. The line

```
    robot.setLEDColorRGB(0, 0, 255);    // blue
```

Changes the LED to blue by setting the argument `b` to full intensity while setting the arguments `r` and `g` to zero intensity. After the LED is changed to blue, the program is paused once again for 2 seconds before the next color change. The LED color is not limited to the primary colors red, green, and blue. For instance, the line

```
robot.setLEDColorRGB(255, 255, 0);  // yellow
```

mixes red and green to make yellow by setting the values of both `r` and `g` to their full intensity while setting the value of `b` to zero intensity. In addition to mixing primary colors at full intensity, it is possible to mix the primary color components at various intermediate values. For example, the line

```
robot.setLEDColorRGB(255, 192, 203);        // pink
```

mixes all three primary colors at intermedate intensities necessary to make the LED pink. Furthermore, by changing the intensities of some of the primary components it is possible to make different shades of pink. For example, Program 12.2 proceeds to change the LED to hot pink and then to deep pink by changing the values of `g` and `b` while keeping `r` at full intensity. The LED is also changed to purple by increasing the intensity of blue while decreasing the intensity of red. The following line

```
robot.getLEDColorRGB(r, g, b);
```

uses the **CLinkbotI** member function **getLEDColorRGB()** to retrieve the red, green, and blue components of the current LED color. Since the last color the LED is set to is deep pink, the values returned in this case are 255 for `r`, 20 for `g`, and 147 for `b`. The final **printf**() statement outputs the following content to the console.

```
The RGB value for the current LED color: 255, 20, 147
```

E Do Exercise 1 on page 206.

12.2.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.setLEDColorRGB(r, g, b);
   ```

   to set the color of the LED on a Linkbot.

2. Call the **CLinkbotI** member function

   ```
   robot.getLEDColorRGB(r, g, b);
   ```

   to get the red, green, and blue values of the current LED color.

12.2.2 Terminology

robot.setLEDColorRGB(), **robot.getLEDColorRGB()**, RGB color model.

12.2.3 Exercises

1. Write a program `colorrgb2.ch` that takes user input for the R, G, and B color values of the LED. Then set the LED of a Linkbot-I using these user specified values.

12.3 Set the Buzzer Frequency of a Robot

In this section, we will learn how to set the buzzer frequency of a Linkbot. We will also learn how to start the buzzer, stop the buzzer, and make the buzzer sound for a specified length of time. A Linkbot's buzzer can be set to an integer value, which indicates the frequency of the buzzer in hertz (Hz). In terms of sound, hertz is defined as

$$1\ Hertz = \frac{1\ vibration}{second}$$

The frequency of a sound determines its pitch. The more vibrations per second, the higher the frequency. And the higher the frequency, the higher the buzzer will sound. The lower the frequency, the lower the buzzer will sound. Although it is possible to use any integer value to set the frequency, generally only frequencies between 20 Hz and 20,000 Hz can be heard by the human ear. Learning how to control the buzzer of a Linkbot enables the user to program a Linkbot to make music.

12.3.1 Set the Buzzer to a Specified Frequency

Program 12.3 demonstrates the basics of how to turn the buzzer on at a specified frequency and how to turn it off.

```
/* File: buzzer.ch
   Make a buzz with the frequency 450 for 1 second */
#include <linkbot.h>
CLinkbotI robot;
int freq = 450;              // frequency for the buzz.
double time1 = 1;            // time in seconds

// set the buzzer with "freq" for "time1" seconds
robot.setBuzzerFrequency(freq, time1);
```

Program 12.3: Setting the buzzer frequency for a Linkbot-I.

The line

```
    int freq = 450;      // frequency for the buzz.
```

declares a variable `freq` of **int** type and assigns it the value 450. This will be the frequency of the buzz in hertz. The line

```
    double time1 = 1;    // time in seconds
```

declares a variable `time1` of **double** type and assigns it the value 1. This will be the time duration of the buzz. the line

```
    robot.setBuzzerFrequency(freq, time1);
```

sets the buzzer frequency to `freq` for `time1` seconds. The above function call is equivalent to

```
   robot.setBuzzerFrequencyOn(freq);
   robot.delaySeconds(time);
   robot.setBuzzerFrequencyOff();
```

The line

```
    robot.setBuzzerFrequencyOn(freq);
```

sets the buzzer of the Linkbot to a frequency and then turns it on. This is done using the **CLinkbotI** member function **setBuzzerFrequencyOn()**. The argument `freq` indicates the desired buzzer frequency. The next line

```
    robot.delaySeconds(time);
```

pauses the program to allow the buzzer to sound for the specified time in its argument. The statement

```
    robot.setBuzzerFrequencyOff();
```

uses the **CLinkbotI** member function **setBuzzerFrequencyOff()** to turn off the buzzer. These member functions can be useful when the during time of the buzzer is not specified.

Do Exercise 1 on page 211.

12.3.2 Set the Buzzer Multiple Times Using a while Loop with a User Specified Frequency

It is also possible to change the frequency of the buzzer to different values, as specified by user input. The frequency can be changed as many times as a user wants within a single program. To do this, we will use a control structure called a loop to control the flow of the program.

```
/* File: buzzer2.ch
   Make a buzz with the frequency specified by the user  */
#include <linkbot.h>
CLinkbotI robot;
int freq;
string_t answer = "yes";    /* "yes" to continue, others to quit */

while(answer == "yes") {
   printf("Please input the frequency for the buzzer\n");
   scanf("%d", &freq);
   robot.setBuzzerFrequency(freq, 1);
   printf("Would you like to enter another frequency? Type yes or no.\n");
   scanf("%s", &answer);
}
```

Program 12.4: Using the function **scanf()** and a **while** loop to set the buzzer frequency multiple times.

Program 12.4 asks a user to input a frequency, sounds the buzzer at that frequency, and repeats this until the user is done entering frequencies. The line

```
    int freq;
```

declares a variable `freq` of type **int** to hold the user entered frequency. The next line

```
    string_t answer = "yes";    /* "yes" to continue, others to quit */
```

declares a variable `answer` of type **string_t** and initializes it with the string "yes". **string_t** is a standard data type used for strings in Ch. The variable `answer` will be used to determine whether or not to continue asking the user for frequency values. The commands in Program 12.4 that take in user input and control the buzzer are repeated using a **while** loop. A ***loop*** is a sequence of statements which is specified once but which may be carried out multiple times in succession. A loop can be an effective way of determining at run time how many times to repeat a sequence of statements. A **while** loop repeats this sequence until the loop condition is no longer true. The general syntax of a **while** loop is

```
    while(condition){
        /* code */
    }
```

where the argument `condition` specifies the loop condition. A ***loop condition*** expression must be true in order to enter and then repeat the **while** loop. In the case of Program 12.4 the expression `answer == "yes"` is used as the loop condition. Each time the sequence of statements is about to execute, the value of `answer` is compared to the value specified in the loop condition. When the variable `answer` has the value `"yes"`, the loop condition evaluates as true and the program can enter and repeat the loop. ***Entering*** a **while** loop refers to the first time the code sequence inside the loop is executed. If the loop condition is false before the program gets to the loop, then the loop will not be entered. For instance, if the variable `answer` in Program 12.4 initially had the value `"no"` then the program would not enter the loop at all. After entering a loop, the loop repeats until the loop condition is false. This is called ***exiting*** the **while** loop. In the case of Program 12.4, when the variable `answer` has any value other than `"yes"` the loop condition expression evaluates as false and the program will exit the loop. In general, the argument `condition` can have the value of `true` or 1, `false` or 0, or it can be a mathematical expression that evaluates as `true` or `false`.

Inside the **while** loop of program 12.4, the lines

```
    printf("Please input the frequency for the buzzer\n");
    scanf("%d", &freq);
```

prompt the user to input a frequency, and stores the input in the variable `freq`. The next lines

```
    robot.setBuzzerFrequency(freq, 1);
```

sound the buzzer at the frequency specified by the user input for one second, then the lines

```
    printf("Would you like to enter another frequency? Type yes or no.\n");
    scanf("%s", &answer);
```

asks the user if they want to enter another frequency. The user's answer is stored in the variable `answer`. As long as the user enters `yes`, then the loop condition `answer == "yes"` is true and the loop will repeat. Once the user enters `no` then the loop condition will be false and the loop will not repeat. The program will then finish executing. When program 12.4 is run, the buzzer can be set at a new frequency as many times as the user wants. The program also finishes when the user decides to quit.

E Do Exercise 2 on page 211.

12.3.3 Change the Buzzer Frequency for a Predetermined Number of Times

In Section 12.3.2 we learned that **while** loops are valuable for deciding how many times to repeat a sequence of statements at runtime. In this section we will learn that **while** loops are also useful for repeating a sequence of statements for the number of times determined ahead of time by the programmer. Program 12.5 demonstrates how to use a **while** loop to increase the buzzer frequency for a predetermined number of times.

```
/* File: buzzer3.ch
   Make buzzes with the frequencies from 1 to 10000 */
#include <linkbot.h>
CLinkbotI robot;
int freq = 1;

while (freq<10000) {
   printf("frequency is %d\n", freq);
   robot.setBuzzerFrequency(freq, 0.05);
   freq = freq + 1;
}
```

Program 12.5: Gradually increasing the buzzer frequency using a **while** loop.

Program 12.5 increases the buzzer frequency of a Linkbot-I by 1 Hz every 50 milliseconds until the frequency of the buzzer reaches 9,999 Hz. The line

```
int freq = 1;
```

declares a variable of **int** type and assigns it a value of 1. This is the starting frequency of the Linkbot's buzzer. The statement

```
while (freq<10000)
```

establishes a **while** loop with `freq<10000` as the loop condition. This loop condition will be true as long as the variable `freq` has a value less than 10,000 Hz. The loop will repeat until the variable `freq` is equal to or greater than 10,000 Hz, which will make the loop condition false. Inside the **while** loop, the line

```
printf("frequency is %d\n", freq);
```

prints the current frequency of the buzzer to the console. The following lines

```
robot.setBuzzerFrequency(freq, 0.05);
```

sets the buzzer to the current frequency value, sounds the buzzer for 0.05 seconds, and then turns it off. The final line

```
freq = freq + 1;
```

increases the value of the buzzer frequency by 1 Hz. Overall, the **while** loop in this program will repeat 9,999 times, each time buzzing at the current frequency for 50 milliseconds and then increasing the frequency by 1 Hz. Even though the buzzer is turned off during each repetition, a very small amount of time passes before the buzzer turns on again. The time elapsed can be measured in microseconds. Because this amount of time is so small, the human ear does not detect it. This makes the buzzer sound like it is on the entire time that the frequency is increasing.

E Do Exercise 3 on page 211.

12.3.4 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.setBuzzerFrequency(freq, time);
   ```

   to turn on the buzzer of a Linkbot at a specified frequency for a specified time duration.

2. Call the **CLinkbotI** member function

   ```
   robot.setBuzzerFrequencyOn(freq);
   ```

   to turn on the buzzer of a Linkbot at a specified frequency.

3. Call the **CLinkbotI** member function

   ```
   robot.setBuzzerFrequencyOff();
   ```

   to turn off the buzzer of a Linkbot.

4. Use a **while** loop to repeat a sequence of statements multiple times in succession while the loop condition is true.

12.3.5 Terminology

robot.setBuzzerFrequency(), **robot.setBuzzerFrequencyOn**(), **robot.setBuzzerFrequencyOff**(), loop, **while** loop, loop condition, entering a loop, exiting a loop.

12.3.6 Exercises

1. Write a program `buzzer4.ch`, which will be based off of program `color2.ch` from Exercise 1 on page 203. Make the following change to the code: after changing the LED color to yellow, make the Linkbot buzz at a frequency of 350 Hz for one second. Then move the Linkbot-I backward 360 degrees.

2. Write a program `buzzer5.ch` using a **while** loop that repeats the following sequence of commands. Prompt the user to enter a frequency value. Set the buzzer to that user specified value for 1 second, then set the buzzer to half of that user specified frequency value for another second. Then ask the user if they want to enter another value.

3. Write a program `buzzer6.ch` that plays all the odd-valued frequencies from 1 Hz to 4999 Hz on the buzzer of a Linkbot-I. Use a **while** loop to accomplish this.

12.4 Set the Buzzers to Specified Frequencies for Multiple Robots

In this section, we will learn how to set the buzzer frequencies for multiple robots.

```
/* File: buzzers.ch
   Two robot buzzers sound together at the same time */
#include <linkbot.h>
CLinkbotI robot1, robot2;
int freq1 = 450, freq2 = 900;

/* set the buzzers on at different frequencies */
robot1.setBuzzerFrequencyOn(freq1);
robot2.setBuzzerFrequencyOn(freq2);

/* Two buzzers sound for 5 seconds */
robot1.delaySeconds(5);

/* set the buzzers off */
robot1.setBuzzerFrequencyOff();
robot2.setBuzzerFrequencyOff();
```

Program 12.6: Setting the buzzer frequencies for two robots.

Program 12.6 sets buzzers of two robots at different frequencies for a specified time. The statements

```
    robot1.setBuzzerFrequencyOn(freq1);
    robot2.setBuzzerFrequencyOn(freq2);
```

set the first robot buzzer at 450 hertz and the second robot buzzer at 900 hertz. using the variables `freq1` and `freq2`. The statement

```
    robot1.delaySeconds(5);
```

allows two buzzers to sound for 5 seconds. The last two statements

```
    robot1.setBuzzerFrequencyOff();
    robot2.setBuzzerFrequencyOff();
```

turn off both buzzers before the program finishes.

Do Exercise 1 on page 212.

12.4.1 Summary

1. Call the **CLinkbotI** member function **setBuzzerFrequencyOn**(freq) to turn on the buzzers of multiple robots.
2. Call the **CLinkbotI** member function **setBuzzerFrequencyOff**() to turn off the buzzers of multiple robots.

12.4.2 Exercises

1. Write a program `buzzers2.ch` to turn on two robot buzzers for 10 seconds, one at the frequency 600 and the other at 1,200 hertz.

12.5 Play Melody and Drive a Robot at the Same Time

We have learned how to play a melody by the member function **playMelody**() in section 3.6. Program 3.5 in section 3.6 plays the melody Jingle Bells first, then drives the robot forward a distance of 12 inches. In this section, you will learn how to play a melody and drive a robot at the same time.

Program 12.7 plays the melody Jingle Bells and drives a robot for a distance of 12 inches at the same time. The program 12.7, uses the **CLinkbotI** member functions **playMelodyNB()** and **playNotesWait()** to synchronize the playing melody and motion of the robot.

```
/* File: playmelodynb.ch
   Play a melody and drive a distance at the same time */
#include <linkbot.h>
CLinkbotI robot;
double radius = 1.75;
double distance = 12;

/* play "Jingle Bells".
   do not block the execution of the next line of the code */
robot.playMelodyNB(JingleBells, 1);

robot.driveDistance(distance, radius);

/* wait for playMelodyNB() to finish */
robot.playNotesWait();

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

Program 12.7: Playing the JingleBells and driving the robot a distance at the same time using **playMelodyNB**().

The general syntax of the member function **playMelodyNB()** is as follows.

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

Similar to the member function **playMelody()**, the first argument `melody` specifies the melody to be played. The suffix "NB" indicates that this member function is non-blocking. A ***non-blocking function*** does not prevent the execution of the next line of code. In other words, the next line of code

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

can begin execution before the non-blocking function **playMelodyNB()** has finished its execution. This way both lines of code can execute simultaneously.

The general syntax of the member function **playNotesWait()** is as follows.

```
robot.playNotesWait();
```

If the member function **driveDistance**() is finished before the melody is completed, the program will wait at the statement

```
robot.playNotesWait();
```

until the melody is finished. Then, the next statement

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

is executed.

E Do Exercise 1 on page 215.

In Program 12.7, the melody is played in the background by the non-blocking member function **playMelodyNB** while the program is executing the member function **driveDistance()**. Similar to Program 12.7, Program 12.8 can also play the melody and drive the robot forward at the same time. However, in Program 12.8, the motion of the robot is started by the non-blocking member function **driveDistanceNB**() first by the statement

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

Then the program executing the statement

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

to play the melody Jingle Bells while the robot is driving forward. If the melody is finished before the motion of robot is completed, the program will wait at the statement

```
    robot.moveWait();
```

until the motion is finished. Then, the next statement

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

will be executed.

```
/* File: driveplaymelody.ch
   Drive a distance and play a melody at the same time */
#include <linkbot.h>
CLinkbotI robot;
double radius = 1.75;
double distance = 12;

/* do not block the execution of the next line of the code */
robot.driveDistanceNB(distance, radius);

/* play "Jingle Bells" */
robot.playMelody(JingleBells, 1);

/* wait for driveDistanceNB() to finish */
robot.moveWait() */

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

Program 12.8: Playing the JingleBells and driving the robot a distance at the same time using **driveDistanceNB**().

The syntax of the non-blocking member function **driveDistanceNB**() is as follows.

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

Similar to the member function **driveDistance**(), the first argument `distance` specifies the distance to be driven. The radius of the two wheels is specified by the second argument `radius`. The syntax of the member function **moveWait**() is as follows.

```
    robot.moveWait();
```

In section 13.6, you will learn how the member function **driveDistanceNB()** can be used to drive multiple robots at the same time.

E Do Exercise 2 on page 215.

12.5.1 Summary

1. Call the non-blocking **CLinkbotI**() member function

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

   to play a melody in background. The lines of code following the non-blocking member function will begin execution before the melody is finished.

2. Call the **CLinkbotI**() member function

   ```
       robot.playNotesWait();
   ```

to pause a program until the melody or music notes notes have finished.

3. Call the non-blocking **CLinkbotI** member function

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

to drive a Linkbot-I for the specified distance. The lines of code following the non-blocking member function will begin execution before the specified distance has reached.

4. Call the non-blocking **CLinkbotI** member function

```
robot.moveWait();
```

to pause a program until all joints of a robot have finished moving.

12.5.2 Terminology

robot.playMelodyNB(), **robot.playNotesWait()**, **robot.driveDistanceNB()**, **robot.moveWait()**.

12.5.3 Exercises

1. Write a program `playmelodynb3.ch`, based on Program 12.7, to play the melody "Do Re Mi" and drive the robot forward 10 inches at the same time. Afterwards, drive the robot back 10 inches.

2. Write a program `driveplaymelody2.ch`, based on Program 12.8, to play the melody "Do Re Mi" and drive the robot forward 10 inches at the same time. Afterwards, drive the robot back 10 inches.

12.6 Play Melodies with Multiple Robots

The non-blocking member function **playMelodyNB()** can also be used to play multiple melodies at the same using multiple robots. As an example, in Program 12.9, two robots play the JingleBells at the same time by the following statements.

```
robot1.playMelodyNB(JingleBells, 1);
robot2.playMelody(JingleBells, 1);
robot1.playNotesWait();
```

```
/* File: playmelodynb2.ch
   Two robot play Jingle Bells melody at the same time */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* play Jingle Bells melody */
robot1.playMelodyNB(JingleBells, 1);
robot2.playMelody(JingleBells, 1);
robot1.playNotesWait();
```

Program 12.9: Two robots playing the JingleBells at the same time.

In Program 12.10, two robots play "The Ants Go Marching" at the same time. Robot 2 plays a higher pitch song.

```
    robot1.playMelodyNB(AntsGoMarching, 1);
    robot2.playMelody(AntsGoMarchingHighPitch, 1);
    robot1.playNotesWait();
```

```
/* File: playmelodyTwoPitches.ch
   Two robots play "The Ants Go Marching" melody at the same time,
   one with high pitch */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* play "The Ants Go Marching" melody at the same time, one with high pitch */
robot1.playMelodyNB(AntsGoMarching, 1);
robot2.playMelody(AntsGoMarchingHighPitch, 1);
robot1.playNotesWait();
```

Program 12.10: Two robots playing "The Ants Go Marching" at the same time with two different pitches.

Do Exercise 1 on page 216.

12.6.1 Summary

1. Call the non-blocking **CLinkbotI**() member function **playMelodyNB**() to play melodies with multiple robots at the same time.

12.6.2 Exercises

1. Write a program `playmelodynb4.ch`, based on Program 12.9, for two robots to play the melody "Do Re Mi" at the same time.

12.7 Play Music Notes and Create Melody Files

Have you ever listened to a song and wondered how its made? How did the musician string a bunch of sounds together, while following a beat? Let's do this with robots in Ch. In this section, we will learn how to create our own melodies in Ch.

12.7.1 Music Basics

Let's get started with some basic music theory. Actually, all the sounds we hear during a song can be represented by notes played for a certain duration.

Notes are the building blocks of music. Each note represents a certain tone that corresponds to a certain frequency. Remember, frequency is measured in Hertz (Hz) which means the number of vibrations per second. Sound waves cause vibrations in the air which our ears detect and we hear as sounds. For example, a note of 100Hz means the air vibrates 100 times per second. A human ear can typically hear from 20 Hz 20,000 Hz. So, a 100Hz note would sound fairly low pitched while a 10,000 one would be sound much sharper and higher pitched.

Musicians don't talk about notes in Hertz, as you can imagine it would get complicated rather quickly. Instead, they use letters to describe notes, and each letter corresponds to a certain frequency. These letters are listed in Table 12.1.

Table 12.1: Music notes and their corresponding frequencies.

Letter	Frequency (Hz)
C1	33
D1	37
E1	41
F1	44
G1	49
A1	55
B1	62

You may notice that a lot of frequencies are not listed. This is because, in music, musicians use scales. Each letter can have a number after it indicating its scale, the lower the number, the lower the frequency, and vice verse. The notes in Table 12.1 are in the first scale. For example, the note C2 has a frequency of 65Hz and thus is higher pitched than the note C1. Scales make the biggest difference in pitch, a note with a lower scale will always be a lower frequency than any note in a higher scale. In addition to scales every C, D, F, G, and A notes can also be "sharp" (denoted by a # in music notation, by S in Ch) which increases their frequencies, but not as much going from one letter to another letter. If you look at C# (35Hz), it has a higher frequency than C but is still lower pitched than a D note.

You may notice that the difference in frequency between notes is not linear. C1 and D1 differ by 4Hz while A1 and B1 differ by 7Hz. This is not a mistake but is actual because the human ear senses change in pitch differently. If a note is very high pitched its hard for humans to notice a change in frequency from one note to another. This is why we have to make the difference in frequency between notes larger when the note is higher pitched. Other senses work this way too. For example, if a light is off and it turns on a little bit the human eye sees this as a big change. But if the light is already really bright, the eye would not be able to detect small changes in its brightness.

Luckily musicians don't worry about frequencies and Hertz when designing a song. Instead, they'll string a bunch of notes together at different scales and duration to make a tune. Next, we'll learn how to program our robots to do this.

12.7.2 Defining a Music Note in Ch

Remember, a note corresponds to a certain frequency which can be represented by a letter, and we can also play a note for a certain duration. Creating a note in Ch is simple. Check out the example below.

```
note_t note = {NOTE_C5, 4};
```

In Ch, notes are a structure of type **note_t** containing two values of {frequency, duration}. Frequency is in Hertz and duration is in length of notes. A duration of 1 would mean the length is one full note, while a duration of 4 means the length is $\frac{1}{4}$ of a full note. This relationship means the bigger the number, the smaller amount of time your note will play for. Typically, we say the length of one full note is 1 second, so a note with a duration of 4 will only play for $\frac{1}{4}$ the length of a full note, or 0.25 second. Therefore, the duration of 2 is for an half note with 0.5 second.

If we want to create a note we need to declare an object of **note_t** type. In the above example, we declare `note` similar to how we would create a robot object or any other variable, and initialize it with the note's frequency and duration. We can access these values by the notation `note.frequency` and `note.duration`.

In Ch, notes are defined as macros starting with `NOTE_` followed by the letter and scale. All valid notes in Ch are listed in Table D.2 in Appendix D on page 363. In the above example, we initialize `note` with the frequency of C in the number 5 scale. We could have also used an actual frequency 523Hz. The duration of 4 for one quarter of a full note time is also specified.

Remember a song is just a sequence of notes. We can represent a sequence of notes in Ch by using an array as shown below.

```
note_t song[] = {{NOTE_C5, 4}, {NOTE_G4, 8}, {NOTE_G4, 8},
      {NOTE_A4, 4}, {NOTE_G4, 4}, {0, 4}, {NOTE_B4, 4}, {NOTE_C5, 4}};
```

In the above code, we declare the variable `song` as an array of **note_t**. It is the same syntax just like declaring an array of **int** or **double**. And since it is an array of notes we have to specify what each note sounds like. When we declare an array like this we separate each note by a comma and enclose the whole thing in curly braces.

12.7.3 Playing an Array of Music Notes

It is easy to play an array of notes in Ch by the member function **playNotes**().

The general syntax of this member function is as follows.

```
    robot.playNotes(song, speedFactor);
```

The first argument `song` specifies the notes of array **note_t** type to be played. The second argument `speedFactor` is the speed factor, which determines how fast to play the notes. The value 1 is for the normal speed. If the value for the speed factor is larger than 1, the notes will be played faster than the normal speed. If the value for the speed factor is less than 1 but larger than 0, the notes will be played slower. For example, the value 2 doubles the speed whereas the value 0.5 is an half of the normal speed.

For example, Program 12.11 plays the Happy Birthday song. In Program 12.11, the variable `song` is declared as an array of **note_t** and initialized with the notes for the Happy Birthday song. This array of notes is played twice by the statements

```
    robot.playNotes(song, 1); // play notes at the normal speed
    robot.playNotes(song, 2); // play notes twice faster
```

It is first played at the normal speed, then twice faster than the normal. We can easily change the notes for the array `song` to play different songs.

```
/* File: playnotes.ch
   Play music notes for Happy Birthday song */
#include <linkbot.h>
CLinkbotI robot;

/* Happy Birthday song */
note_t song[] = {{NOTE_C4, 8}, {NOTE_C4, 8},
                 {NOTE_D4, 4}, {NOTE_C4, 4}, {NOTE_F4, 4},
                 {NOTE_E4, 4}, {0, 4}, {NOTE_C4, 8},
                 {NOTE_C4, 8}, {NOTE_D4, 4}, {NOTE_C4, 4},
                 {NOTE_G4, 4}, {NOTE_F4, 4}, {0, 4},
                 {NOTE_C4, 8}, {NOTE_C4, 8}, {NOTE_C5, 4},
                 {NOTE_A4, 4}, {NOTE_F4, 8}, {NOTE_F4, 8},
                 {NOTE_E4, 4}, {NOTE_D4, 4}, {0, 4},
                 {NOTE_AS4, 8}, {NOTE_AS4, 8}, {NOTE_A4, 4},
                 {NOTE_F4, 4}, {NOTE_G4, 4}, {NOTE_F4, 4}};

robot.playNotes(song, 1);  // play notes at the normal speed
robot.playNotes(song, 2);  // play notes twice faster
```

Program 12.11: Playing the notes for Happy Birthday song.

12.7.4 Write and Play Customized Melody Files

We have learned how to declare a song as an array of **note_t** and play the song using the member function **playNotes**(). However, you may want to play the same song in different Ch programs or share it with others. It is better to store the song in a Ch function file as a melody file. In this section, we will learn how to create customized melody files.

Remember that we use the member function **playMelody**() to play a pre-defined melody, JingleBells, in Program 3.5 in section 3.6. The melody is distributed along with other program a Ch function file. Now, let's start to create our own Happy Birthday melody file.

Program 12.12 contains the source code of the the Happy Birthday melody file, or a Ch function file. The function name myHappyBirthday() defined inside the function file shall be the same as the file name. The file name must has a file extension .chf.

```
/* File: myHappyBirthday.chf
   The function file myHappyBirthDay.chf for melody myHappyBirthday */
#include <linkbot.h>
note_t myHappyBirthday(int i) {
    int len;
    note_t note;
    note_t song[] = {{NOTE_C4, 8}, {NOTE_C4, 8},
        {NOTE_D4, 4}, {NOTE_C4, 4}, {NOTE_F4, 4},
        {NOTE_E4, 4}, {0, 4}, {NOTE_C4, 8},
        {NOTE_C4, 8}, {NOTE_D4, 4}, {NOTE_C4, 4},
        {NOTE_G4, 4}, {NOTE_F4, 4}, {0, 4},
        {NOTE_C4, 8}, {NOTE_C4, 8}, {NOTE_C5, 4},
        {NOTE_A4, 4}, {NOTE_F4, 8}, {NOTE_F4, 8},
        {NOTE_E4, 4}, {NOTE_D4, 4}, {0, 4},
        {NOTE_AS4, 8}, {NOTE_AS4, 8}, {NOTE_A4, 4},
        {NOTE_F4, 4}, {NOTE_G4, 4}, {NOTE_F4, 4}
    };

    len = sizeof(song) / sizeof(note_t);
    if (i < len) {
        note.frequency = song[i].frequency;
        note.duration = song[i].duration;
    } else {
        note.frequency = -1;
        note.duration = -1;
    }

    return note;
}
```

Program 12.12: The function file myHappyBirthDay.chf for Happy Birthday melody.

In this function, we defined three variables `len`, `note` and `song`. The array `song` of **note_t** stores all the notes of the song which is the same as we introduced in the previous section. The variable `len` is used to store the number of notes in `song`. The ith note stored in `song` will be retrieved and assigned to the variable `note`.

```
    note.frequency = song[i].frequency;
    note.duration = song[i].duration;
```

This note will be returned.

An **if** statement is used to check if the argument i is over the length of notes stored in `song`. If it passed the length, a spell value of -1 would be assigned to the frequency and duration fields of the `note`.

```
    note.frequency = -1;
    note.duration = -1;
```

This special note allows the member function **playNotes**() to stop playing the notes.

To play the melody file, simply call the member function **playMelody**() by passing the melody function we just created as shown in Program 12.13. However, make sure you will add the function prototype

```
    note_t myHappyBirthday(int i);
```

before the melody file is used by the member function **playNotes**() as shown below.

```
    /* drive the robot 5 inches forward, then playing the melody */
    robot.driveDistance(5, 1.75);
    robot.playMelody(myHappyBirthday, 1);
```

```
/* File: birthday.ch
   Play the melody myHappyBirthday */
#include <linkbot.h>
CLinkbotI robot;
note_t myHappyBirthday(int i);

/* drive the robot 5 inches forward, then playing the melody */
robot.driveDistance(5, 1.75);
robot.playMelody(myHappyBirthday, 1);
```

Program 12.13: Playing the melody file myHappyBirthDay.chf for Happy Birthday melody.

One thing needs to be noted that the melody file with the file extension .chf need to be in the same folder where your main program is or in a folder for function files where Ch can find it. Otherwise, Ch will not be able to find where the function is. More information about Ch function files can be found in *Ch User's Guide* available through the C-STEM Studio.


Do Exercises 1 and 2 on page 221.

12.7.5 Summary

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

   ```
   robot.playNotes(note_t song[], speedFactor);
   ```

   play an array of notes.

2. Create Melody files with a file extension .chf.

12.7.6 Terminology

Ch function file, melody file, notes, **robot.playNotes()**.

12.7.7 Exercises

1. The "Do Re Mi" notes are given below.

   ```
   note_t song[] = {{NOTE_C4, 2}, {NOTE_D4, 2}, {NOTE_E4, 2}, {NOTE_F4, 2},
                    {NOTE_G4, 2}, {NOTE_A4, 2}, {NOTE_B4, 2}, {NOTE_C5, 2}};
   ```

   Write a program `playnotes2.ch`, based on Program 12.11, to first play the notes at the normal speed. at three times of the normal speed, and at the half of the normal speed.

2. Write a Do Re Mi melody file myDoReMi.chf based on the notes in Exercise 1. Then write a program `doremi.ch` to play this melody using the member function **playMelody**().

12.8 Play Music Notes with Multiple Robots

In this section, we will learn how to play music notes for multiple robots.

```
/* File: playnoteswait.ch
   Play two music notes at the same time for Happy Birthday song */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* Happy Birthday song */
note_t song1[] = {{NOTE_C4, 8}, {NOTE_C4, 8},
                  {NOTE_D4, 4}, {NOTE_C4, 4}, {NOTE_F4, 4},
                  {NOTE_E4, 4}, {0, 4}, {NOTE_C4, 8},
                  {NOTE_C4, 8}, {NOTE_D4, 4}, {NOTE_C4, 4},
                  {NOTE_G4, 4}, {NOTE_F4, 4}, {0, 4},
                  {NOTE_C4, 8}, {NOTE_C4, 8}, {NOTE_C5, 4},
                  {NOTE_A4, 4}, {NOTE_F4, 8}, {NOTE_F4, 8},
                  {NOTE_E4, 4}, {NOTE_D4, 4}, {0, 4},
                  {NOTE_AS4, 8}, {NOTE_AS4, 8}, {NOTE_A4, 4},
                  {NOTE_F4, 4}, {NOTE_G4, 4}, {NOTE_F4, 4}};

/* Happy Birthday song with higher pitch */
note_t song2[] = {{NOTE_C5, 8}, {NOTE_C5, 8},
                  {NOTE_D5, 4}, {NOTE_C5, 4}, {NOTE_F5, 4},
                  {NOTE_E5, 4}, {0, 4}, {NOTE_C5, 8},
                  {NOTE_C5, 8}, {NOTE_D5, 4}, {NOTE_C5, 4},
                  {NOTE_G5, 4}, {NOTE_F5, 4}, {0, 4},
                  {NOTE_C5, 8}, {NOTE_C5, 8}, {NOTE_C6, 4},
                  {NOTE_A5, 4}, {NOTE_F5, 8}, {NOTE_F5, 8},
                  {NOTE_E5, 4}, {NOTE_D5, 4}, {0, 4},
                  {NOTE_AS5, 8}, {NOTE_AS5, 8}, {NOTE_A5, 4},
                  {NOTE_F5, 4}, {NOTE_G5, 4}, {NOTE_F5, 4}};

robot1.playNotesNB(song1, 1);
robot2.playNotes(song2, 1);
robot1.playNotesWait();
```

Program 12.14: Playing the Happy Birthday song at two different pitches with two robots.

The statements

```
robot1.playNotesNB(song1, 1);
robot2.playNotes(song2, 1);
robot1.playNotesWait();
```

in Program 12.14 play the Happy Birthday song at two different pitches with two robots. The `robot1` plays the music notes `song1` by the non-blocking member fuction **playNotesNB**() using the first robot. The `robot2` plays the music notes `song2` by the member fuction **playNotes**() using the second robot. The member function **playNotesWait**() pauses the program till the first robot finishes playing the music notes.

The general syntax of the non-blocking member function **playNotesNB**() is as follows.

```
robot.playNotesNB(song, speedFactor);
```

This is the non-blocking version of the member function **playNotes**(). Similar to **playNotes**(), the first argument `song` specifies the notes of array **note_t** type to be played. The second argument `speedFactor` is the speed factor, which determines how fast to play the notes.

E Do Exercise 1 on page 223.

12.8.1 Summary

1. Call the **CLinkbotI** member function **playNotesNB**() to play music notes in a background.

2. Call the **CLinkbotI** member function **playNotesWait**() to pause the program till the music notes are finished.

12.8.2 Exercises

1. The "Do Re Mi" notes are given below.

```
note_t song[] = {{NOTE_C4, 2}, {NOTE_D4, 2}, {NOTE_E4, 2}, {NOTE_F4, 2},
                 {NOTE_G4, 2}, {NOTE_A4, 2}, {NOTE_B4, 2}, {NOTE_C5, 2}};
```

   Write a program `playnoteswait2.ch`, based on Program 12.11, to play the "Happy Birthday" and "Do Re Mi" notes at the same time using two robots.

2. Write a program `playnoteswait3.ch`, based on Program 12.11, to play the "Happy Birthday" notes and Jingle Bells melody at the same time using two robots.

12.9 Get the Accelerometer Data

In this section we will learn how to read the accelerometer data of a Linkbot-I. An ***accelerometer*** is a type of sensor that measures the acceleration, or g-force, experienced by an object such as a robot. Accelerometers have many important applications in robotics. Because an accelerometer can sense g-forces in real time, it is useful in analyzing the way a robot moves. For example, an accelerometer can be used in a guidance system to help a robot determine the ideal path to perform a task. Accelerometers are commonly used to help robots to navigate different types of terrain and aid robots in collision detection. Consequently, it is a useful skill to be able to program a robot to receive, process, and respond to accelerometer data in real time.

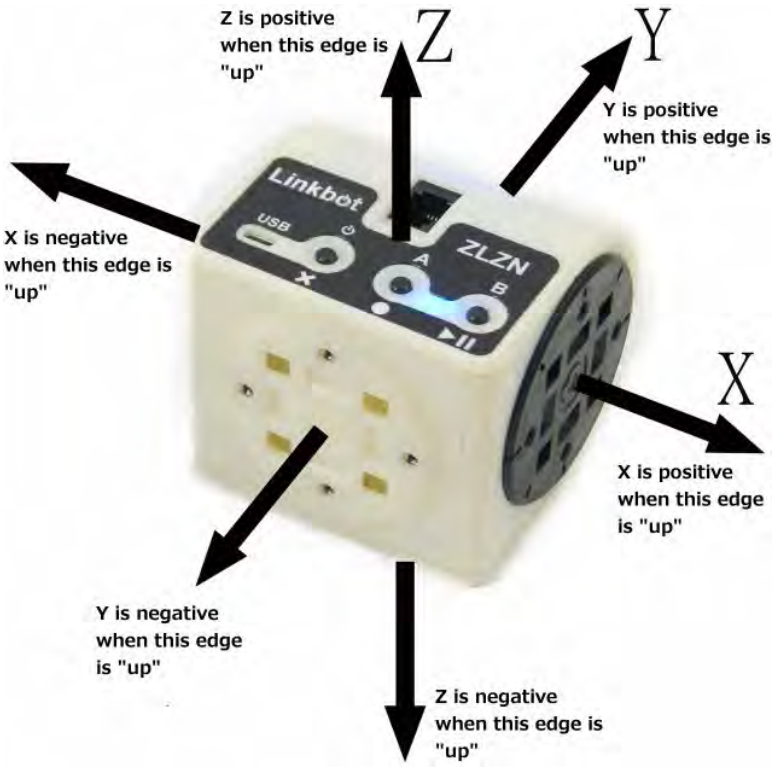


Figure 12.2: The X, Y, and Z directions of the accelerometer.

The Linkbot accelerometer measures the magnitude and direction of g-forces along the X, Y, and Z axes. Figure 12.2 shows the positive and negative directions of the X, Y, and Z axes as detected by the accelerometer of a Linkbot-I. These axes are the same for a Linkbot-L. This accelerometer can detect all the free falls, bumps, and tilt angles of a Linkbot. The accelerometer helps to connect two Linkbots through BumpConnect, by detecting the collision of the Linkbots. The accelerometer can also be used to control a Linkbot using the accelerometer readings of another Linkbot.

The values of the X, Y, and Z components and magnitude of the accelerometer for a Linkbot can be viewed via the Robot Control Panel as shown in Figure 2.8 on page 18. Program 12.15 shows how to program a Linkbot-I to retrieve the X, Y, and Z components of the data from its accelerometer. While running Program 12.15, be sure to hold the Linkbot and move or tilt it in different directions.

```
/* File: getaccelerometer.ch
   get the accelerometer data */
#include <linkbot.h>
CLinkbotI robot;
double x, y, z;

while(1) {
  robot.getAccelerometerData(x, y, z);
  printf("Accelerometer: x: %lf y: %lf z: %lf\n", x, y, z);
  robot.delaySeconds(1);
}
```

Program 12.15: Getting the accelerometer data of a Linkbot-I using the function **getAccelerometerData()**.

The line

```
    double x, y, z;
```

declares three variables of type **double**, `x`, `y`, and `z`, to hold the values of the X, Y, and Z components of the accelerometer data. After connecting to the Linkbot, the statement

```
    while(1)
```

starts a **while** loop with a loop condition of 1. This condition will always be true, so the loop will repeat until the Linkbot is turned off or the user presses the `Stop` button on the debug bar in ChIDE. Inside the **while** loop, the line

```
    robot.getAccelerometerData(x, y, z);
```

retrieves the current X, Y, and Z components of the accelerometer data using the **CLinkbotI** member function **getAccelerometerData()**. After this member function is called, the arguments `x`, `y`, and `z` will hold the values of the X, Y, and Z components of acceleration. The next two lines

```
    printf("Accelerometer: x: %lf y: %lf z: %lf\n", x, y, z);
    robot.delaySeconds(1);
```

print the X, Y, and Z values of acceleration to the console, and delay the program for one second before repeating the **while** loop. The sequence of statements inside the loop will repeatedly get accelerometer data from the Linkbot-I and print this data to the console as long as Program 12.15 continues running.

An application of the accelerometer data will be illustrated by a robot relay race in section 13.9.1 on page 263.

E Do Exercises 1 and 2 on page 225.

12.9.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.getAccelerometerData(x, y, z);
   ```

   to get the X, Y, and Z component values from the accelerometer of a Linkbot-I.

12.9.2 Terminology

robot.getAccelerometerData(), accelerometer.

12.9.3 Exercises

1. Connect a Linkbot-I to Linkbot Labs. Click on the "Sensors" tab and look at sliders under "Accelerometer Data". Move the Linkbot into a position where X equals 1, while Y and Z are both approximately zero. Then, move the Linkbot into a position where Y equals 1 while X and Z are both approximately zero. Finally, move the Linkbot into a position where Z equals 1 while X and Y are both approximately zero.
2. Run Program 12.15 and move the Linkbot-I into the positions described in Exercise 1.

12.10 Get the Battery Voltage

In this section, we will learn how to read the battery voltage of a Linkbot-I. It is useful to have the ability to monitor how much battery life is left on a Linkbot before it needs to be recharged. Program 12.16 shows how to check the remaining battery voltage of a Linkbot-I.

```
/* File: getbattery.ch
   get the battery voltage */
#include <linkbot.h>
CLinkbotI robot;
double v;

robot.getBatteryVoltage(v);
printf("Battery voltage is %lf\n", v);
```

Program 12.16: Getting the battery voltage of a Linkbot-I using the function **getBatteryVoltage()**.

Program 12.16 retrieves the battery voltage of a Linkbot-I. The line

```
double v;
```

declares a variable `v` of type **double**. This variable will be used to hold the voltage value. After the program connects to a Linkbot, the line

```
robot.getBatteryVoltage(v);
```

gets the voltage of the battery using the **CLinkbotI** member function **getBatteryVoltage()**. After this function is called, the argument `v` will have the voltage value of the Linkbot's battery. This value is then printed to the console by the statement below.

```
    printf("Battery voltage is %lf\n", v);
```

The maximum voltage of a Linkbot's lithium-ion battery pack is 4.2 volts. The battery dies when it reaches a voltage of about 3.3 volts, and the Linkbot needs to be recharged. Figure 12.3 shows how a fully charged battery of a Linkbot discharges over time. This graph was obtained by running Program 12.17. Some of the programming techniques used in Program 12.17 are beyond the scope of this book. Just keep in mind that Program 12.17 measures the battery voltage of a Linkbot-I every 5 minutes until the battery dies. The data is stored in a separate file, which is used to generate the graph after all measurements are taken. Between measurements, joints 1 and 3 of the Linkbot were rotated forward at the maximum speed of 200 degrees per second. This rotation of joints 1 and 3 simulates a constant workload, in order to give a better idea of how the battery will discharge under heavy use.

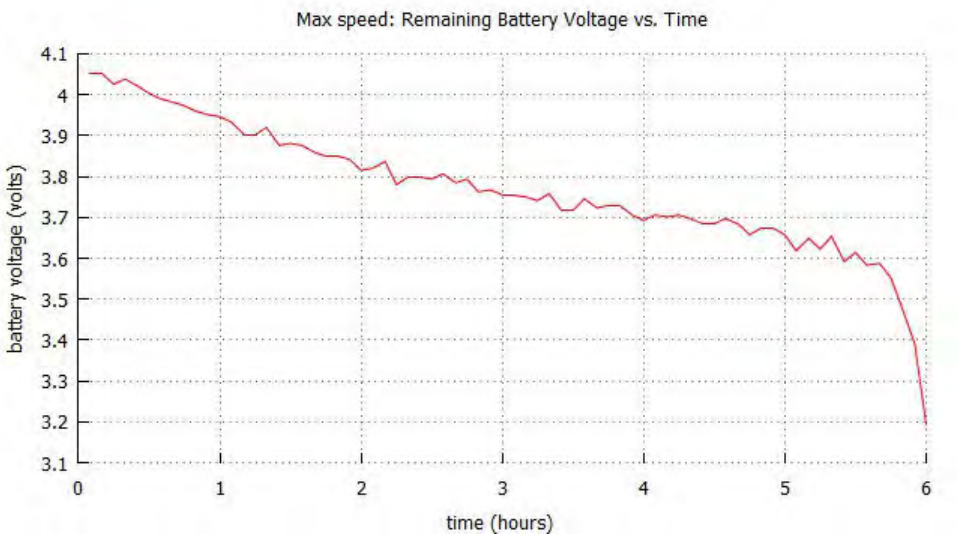


Figure 12.3: The discharge profile of a Linkbot-I battery over time.

Under this workload, the Linkbot-I battery lasts about 6 hours. Notice that the discharging pattern is not linear. The battery discharges slowly over a period of about 5.7 hours, then the voltage drops off quickly in the remaining half hour.

```
/* File: batterytest.ch
   Get the battery voltage every 5 minutes and store it in a file until the battery
   dies. Then plot voltage vs. time from the file. */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
CPlot plot;
double time1 = 300;   // 5 minutes == 5 * 60 seconds == 300 seconds.
double hour = 3600.0; //1 hour == 1 * 60 minutes * 60 seconds == 3600 seconds.
double totalTime = 0; // a variable to keep track of the total time in seconds.
double voltage = 4.2; // a starting value for v, so that the program enters the
                      //while loop.
double speed=200; // max joint speed(deg/s).
string_t filename = "voltage.txt"; //The file used to store the data.
FILE * stream;  // this will be the file stream.

//Set joints to maximum speed
robot.setJointSpeeds(speed,NaN,speed);

stream = fopen("voltage.txt", "w"); //Open the file stream
if(stream == NULL) // If file stream does not open successfully,
{                  // then exit the program.
    printf("Error: cannot open 'voltage.txt'\n");
    exit(-1);
}

// repeat the following until v < 3.3, when battery should die.
while (voltage >= 3.3){

    // moveTime() for 5 minutes
    robot.moveTime(time1);

    // add 5 minutes to total time in hours
    totalTime = totalTime + (time1/hour);

    // get the battery voltage
    robot.getBatteryVoltage(voltage);

    // store time and voltage values in a file
    fprintf(stream, "%lf %lf\n", totalTime, voltage);

}
fclose(stream);

plot.title("Max speed: Remaining Battery Voltage vs. Time");
plot.label(PLOT_AXIS_X, "time (hours)");
plot.label(PLOT_AXIS_Y, "battery voltage (volts)");
plot.dataFile("voltage.txt");
plot.plotting();
```

Program 12.17: Plotting the discharge of a Linkbot battery over time.

Do Exercise 1 and 2 on page 228.

12.10.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.getBatteryVoltage(v);
   ```

   to get the battery voltage of a Linkbot-I.

12.10.2 Terminology

robot.getBatteryVoltage().

12.10.3 Exercises

1. Write a program `getbattery2.ch` to get the battery voltage of a Linkbot-I, and then calculate the percent of the battery voltage remaining. Print both the voltage and the voltage percent to the console.
2. Write a program `getbattery3.ch` using a **while** loop to get the battery voltage of a Linkbot-I every 5 minutes until the battery is fully discharged. Each time you get the voltage, print it to the console, along with the amount of time in seconds that has passed since the program started. In between voltage measurements, move joints 1 and 3 of the Linkbot-I for 5 minutes.

CHAPTER 13

Writing Programs to Control Multiple Individual Linkbots

13.1 Control Multiple Linkbots Using the Robot Control Panel

A computer can connect to multiple Linkbots as described in section 2.2 in Chapter 2. The Robot Control Panel can be used to control multiple Linkbots, one at a time. If a computer is connected to multiple Linkbots, an individual Linkbot can be selected to be controlled by picking the Linkbot with its ID on the Robot Control Panel as shown in Figure 2.8 on page 18.

E Do Exercise 1 on page 229.

13.1.1 Summary

1. Connect to multiple Linkbots from a computer through Linkbot Labs and control one Linkbot at a time through the Robot Control Panel.

13.1.2 Terminology

Robot Control Panel.

13.1.3 Exercises

1. Work with your partner to connect a computer to two Linkbots. Control the connected two Linkbots using the Robot Control Panel of Linkbot Labs on the computer.

13.2 Control Multiple Linkbots Using a Program

In Chapter 7 we learned how to control multiple Linkbots in a group with identical movements simultaneously. In this chapter we will learn how to control multiple Linkbots in the same program, but they will have different movements at different times. Although it is efficient to control multiple robots using a single group, there will be times when it is preferable for robots to move independently within the same program.

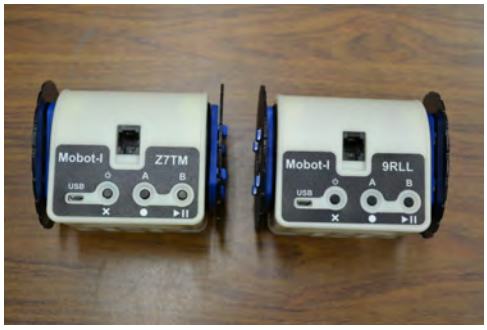


Figure 13.1: Control two modules of Linkbot-I.

```
/* File: twomodules.ch
   Control two robots sequentially. */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* move joints 1 and 3 for robot1 */
robot1.move(360, NaN, -360);

/* move joints 1 and 3 for robot2 */
robot2.move(360, NaN, -360);
```

Program 13.1: Controlling two modules of Linkbot-I in a program.

Recall that Program 10.1 on page 169 moves a single Linkbot-I forward by 360 degrees. Program 13.1 moves two Linkbots forward by 360 degrees, in the same manner as Program 10.1. But each Linkbot-I moves at a different time. The line

```
    CLinkbotI robot1, robot2;
```

declares two variables and connect them to two Linkbot-Is sequentially. The first of the next two lines

```
    robot1.move(360, NaN, -360);
    robot2.move(360, NaN, -360);
```

moves `robot1` forward 360 degrees first. Only when `robot1` has finished moving does `robot2` start moving forward 360 degrees.

You can run the program `twomodules.ch` in Program 13.1 to control two Linkbot-Is and monitor their joint angles on the Robot Control Panel, showing one Linkbot-I at a time.

E Do Exercise 1 on page 231.

13.2.1 Exercises

1. Write a program `twomodules2.ch` to control two Linkbot-Is. The program will move joints 1 and 3 of the first Linkbot-I to 90 degrees. Then, it will move the joints of the second Linkbot-I the same amount as the first one.

13.2.2 Summary

1. The commands

```
robot1.move(360, NaN, -360);
robot2.move(360, NaN, -360);
```

   move `robot1` forward 360 degrees first. When `robot1` has finished moving, then `robot2` starts moving.

13.2.3 Terminology

independent movements, sequential commands.

13.3 Blocking versus Non-Blocking Functions

All of the functions that have been introduced so far have been blocking functions. A ***blocking function*** is a function that must complete its execution before the next line of code can start executing. The previous program demonstrated how blocking functions work. The use of blocking functions is fine if the programmer intends for independent motions to execute sequentially. But a programmer may want to command Linkbots in independent motions simultaneously. In this section we will learn how to achieve this effect using non-blocking functions.

Run the programs in these sections in ChIDE in debug mode using the command `Next`.

```
/* File: block.ch
   Use the blocking function moveJoint(). */
#include <linkbot.h>
CLinkbotI robot;

/* rotate joint 1 by 360 degrees first, then joint 2 by 30 degrees */
robot.moveJoint(JOINT1, 360);
robot.moveJoint(JOINT3, 30);
```

Program 13.2: A program using all blocking functions.

Program 13.3 demonstrates the use of blocking functions on the same Linkbot. The line

```
robot.moveJoint(JOINT1, 360);
```

rotates joint 1 of the Linkbot-I by 360 degrees first. Only after joint 1 has completed its rotation does the line

```
robot.moveJoint(JOINT3, 30);
```

rotate joint 3 by 30 degrees. The execution of these two functions is sequential.

```
/* File: nonblocka.ch
   Use the non-blocking function moveJointNB(). */
#include <linkbot.h>
CLinkbotI robot;

/* rotate joint 1 by 360 degrees and joint 3 by 180 degrees at the same time
   till the movement for joint 3 is finished. */
robot.moveJointNB(JOINT1, 360);
robot.moveJoint(JOINT3, 180);
```

Program 13.3: Using the non-blocking function **moveJointNB**().

Program 13.3 demonstrates non-blocking functions. A ***non-blocking function*** does not prevent the execution of the next line of code. In other words, the next line of code can begin execution before the non-blocking function has finished its execution. This way both lines of code can execute simultaneously, as long as their actions do not overlap. The non-blocking function introduced in Program 13.3 is the **CLinkbotI** member function **moveJointNB()**. The general syntax of this function is

```
robot.moveJointNB(id, angle);
```

The argument `id` specifies the joint to move. The argument `angle` is the final position of the joint angle relative to its current position. The function **moveJointNB()** moves a Linkbot-I in the same way as the function **moveJoint()** does. But when **moveJointNB()** is used, the next line of code starts its execution before **moveJointNB()** has finished executing. A non-blocking version of a member function for motion has the suffix **NB**. In Program 13.3 the line

```
robot.moveJointNB(JOINT1, 360);
```

starts rotating joint 1 by 360 degrees relative to its current position. Immediately the next line

```
robot.moveJoint(JOINT3, 180);
```

begins rotating joint 3 by 180 degrees relative to its current position. As a result, joints 1 and 3 will rotate simultaneously until joint 3 has finished rotating.

In Program 13.3, Joint 1 will not be able to complete the full rotation of 360 degrees before the program finishes its execution.

Since the code will continue to run after calling a non-blocking function, it is sometimes desirable to wait until certain Linkbot motions have finished. For example, a program may command joint 1 to rotate 360 degrees. If that rotation has not yet completed, and the next line of code commands the same joint in a different motion, then that next line would need to be delayed. Program 13.4 demonstrates how to do this.

```
/* File: nonblockb.ch
   Use the function moveJointWait() to wait for the completion of
   a joint movement. */
#include <linkbot.h>
CLinkbotI robot;

/* rotate joint 1 for 360 degrees and joint 3 for 720 degrees at the same time.
   till the movement for  joint 1 is finished, then move joint 1 by 30 degrees */
robot.moveJointNB(JOINT1, 360);
robot.moveJointNB(JOINT3, 720);
robot.moveJointWait(JOINT1);
robot.moveJoint(JOINT1, 30);
robot.moveJointWait(JOINT3);
```

Program 13.4: Using the function **moveJointWait**() to wait for the completion of a joint movement from **moveJoint**().

Program 13.4 makes use of the **CLinkbotI** member function **moveJointWait()**. This function pauses the program until the specified joint has finished its motion. The syntax of the function **moveJointWait()** is

```
robot.moveJointWait(id);
```

The argument `id` specifies the joint for which you want to pause the program until its current motion is completed. This command only applies to the joint number for the specific Linkbot-I it is used on. In Program 13.4 the lines

```
robot.moveJointNB(JOINT1, 360);
robot.moveJointNB(JOINT3, 720);
```

command the Linkbot-I to move joint 1 by 360 degrees and joint 3 by 720 degrees simultaneously. The next line

```
robot.moveJointWait(JOINT1);
```

commands the program to pause until joint 1 has finished moving 360 degrees relative to its starting position. The next line of the program will not begin execution until joint 1 has finished moving. Note, however, that joint 3 will still be in the process of rotating 720 degrees when the program resumes execution. The next line

```
robot.moveJoint(JOINT1, 30);
```

will rotate joint 1 by 30 degrees at the same time that joint 3 continues its 720 degree rotation. After joint 1 finishes its 30 degree rotation, the final line

```
robot.moveJointWait(JOINT3);
```

pauses the program until joint 3 has finished rotating 720 degrees. This last line of the code ensures that joint 3 will be able to complete its full 720 degree rotation before the program finishes execution.

E Do Exercises 1 and 2 on page 235.

There are times when it may be desirable to wait for all joints of a specific Linkbot to finish moving. Program 13.5 shows how this can be accomplished.

```
/* File: nonblockc.ch
   Use the function moveWait(). */
#include <linkbot.h>
CLinkbotI robot;

/* rotate joint 1 for 360 degrees and joint 3 for 720 degrees at the same time.
   till the movement for all joints is finished, then move joint 1 by 30 degrees */
robot.moveJointNB(JOINT1, 360);
robot.moveJointNB(JOINT3, 720);
robot.moveWait();
robot.moveJoint(JOINT1, 30);
```

Program 13.5: Using the function **moveWait**() to wait for the completion of all joint movement.

The **CLinkbotI** member function that will pause the program until certain motions are complete is **moveWait()**. The syntax of this function is

```
robot.moveWait();
```

The function **moveWait()** has no argument. It pauses the program until all joints of a specific Linkbot-I have finished rotating. As in the previous program, the statements

```
robot.moveJointNB(JOINT1, 360);
robot.moveJointNB(JOINT3, 720);
```

rotate joint 1 by 360 degrees and joint 3 by 720 degrees simultaneously. The next statement

```
robot.moveWait();
```

pauses the program until both joint 1 and joint 3 have finished rotating. The last statement

```
robot.moveJoint(JOINT1, 30);
```

rotates joint 1 by 30 degrees. Since this last statement uses a blocking function, joint 1 is able to complete its 30 degree rotation before the program finishes execution.

E Do Exercise 3 on page 235.

13.3.1 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveJointNB(id, angle);
   ```

   to move a joint with the specified angle relative to its current position. The next line of code will begin execution before the joint has finished its rotation.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveJointWait(id);
   ```

   to pause the program until a specific joint of a specific Linkbot has finished rotating.

3. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveWait();
   ```

   to pause a program until all joints of a specific Linkbot have finished rotating.

13.3.2 Terminology

robot.moveJointNB(), **robot.moveJointWait()**, **robot.moveWait()**, blocking function, non-blocking function.

13.3.3 Exercises

1. In Program 13.3, the motion for the rotation of 360 degrees for joint 1 will not be able to complete before the program exits. Modify Program 13.3 as a new program `nonblocka2.ch` by adding the following function call at the end of the program.

   ```
   robot.moveJointWait(JOINT1);
   ```

   Run programs `nonblocka.ch` and `nonblocka2.ch` while monitoring the joint angle for joint 1 on the Robot Control Panel in Linkbot Labs to see the different motions.

2. Write a program `nonblockb2.ch` to move joint 1 of a Linkbot-I to 360 degrees and joint 3 to 180 degrees at the same time using the member function **moveJointNB()**. Once both joints 1 and 3 finish the motion, move joint 3 to 45 degrees.

3. Write a program `nonblockc2.ch` to move joint 1 of a Linkbot-I by 360 degrees and joint 3 by 180 degrees at the same time using the member function **moveJointNB()**. Once joint 3 finishes the motion, move joint 3 by another 45 degrees. Make sure all motions complete before the program ends (Hints: need **moveWait**() at the end of the program).

13.4 Record and Plot Two Joint Angles versus Time

In Section 10.7.2, we learn to control a robot with different wheel sizes to move in a straight line using the member functions **driveTime**() and **moveTime**(). In this section, we will learn how to control two joints of a Linkbot individually by a sample application. This will be useful if you would like to control a Linkbot-I as a two-wheel vehicle with two wheels of different radius. We will also learn how to record two joints of a Linkbot at the same time.

> **Problem Statement:**
> Write a program `recordangles.ch` to rotate joint 1 of a Linkbot-I for 720 degrees at 120 degrees per second and joint 3 for 360 degrees at -90 degrees per second, starting at the same time. The radius of each wheel is 1.75 inches. Plot the joint angles for joints 1 and 3 versus time.

We can solve this problem conveniently using Ch Linkbot Controller (CLC) with the setup shown in Figure 13.2. For control individual joints separately, we need to use "General Vehicle Control" in CLC.

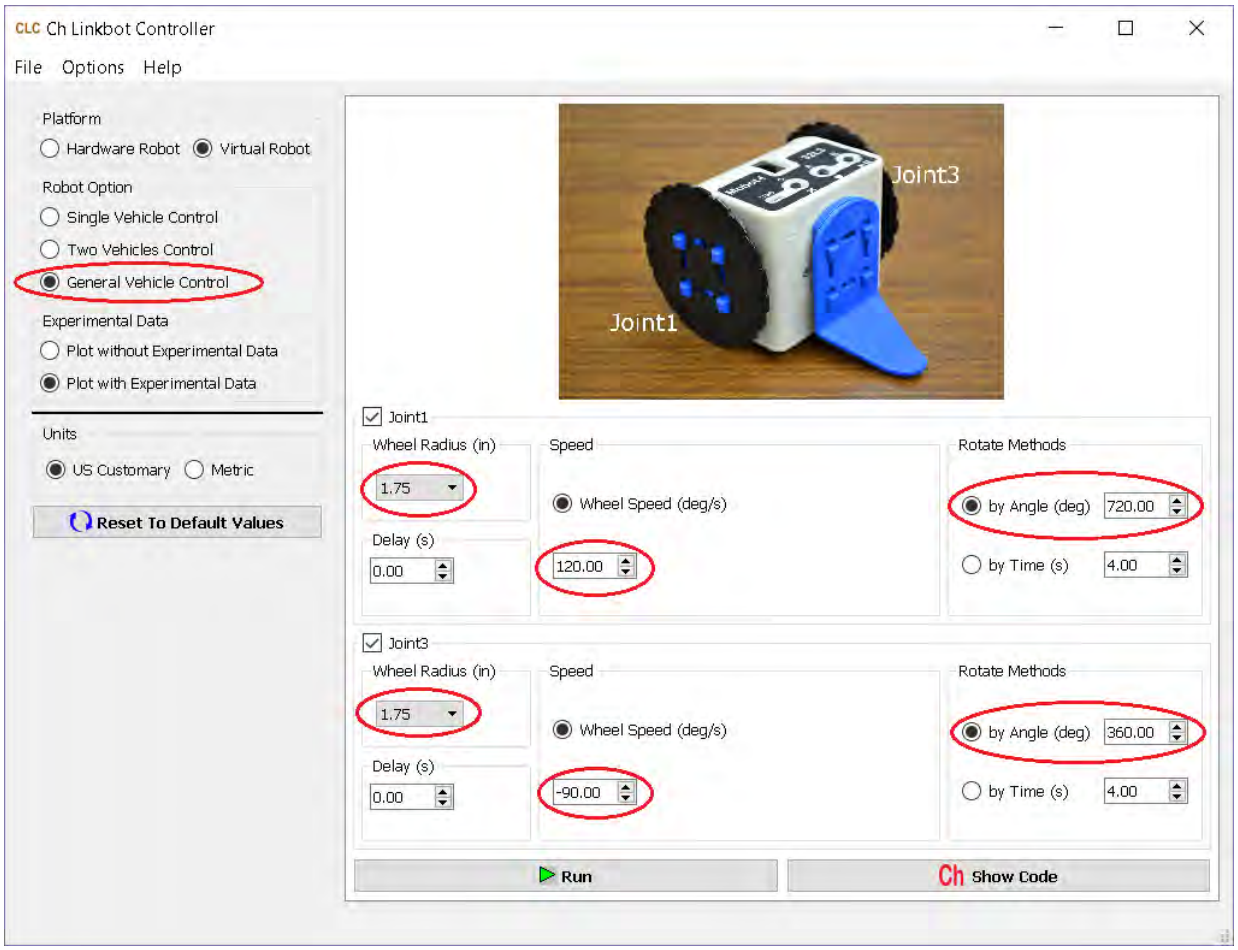


Figure 13.2: The setup of Ch Linkbot Controller for solving the problem.

E Do Exercise 1 on page 239.

```
/* File: recordangles.ch
   Record joint angles and time, plot the acquired data */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot;
double speed1 = 120;         // speed for joint 1 in 120 degrees/sec nds
double speed2 = -90;         // speed for joint 3 in -90 degrees/seconds
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints;          // number of data points recorded
robotRecordData_t timedata, angledata1; // recorded time and angles for joint 1
robotRecordData_t angledata3; // recorded angles for joint 3
CPlot plot;                 // plotting class

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

/* set the joints 1 and 3 speed */
robot.setJointSpeeds(speed1, NaN, speed2);

/* begin recording time and angles */
robot.recordAnglesBegin(timedata, angledata1, angledata3, timeInterval);

/* move joint 1 for 720 degrees */
robot.moveJointNB(JOINT1, 720);
/* move joint 3 for 360 degrees */
robot.moveJoint(JOINT3, 360);
robot.moveJointWait(JOINT1);

/* end recording time and angle */
robot.recordAnglesEnd(numDataPoints);

/* plot the data */
plot.title("Angles for joint 1 versus time");
plot.label(PLOT_AXIS_X, "time (seconds)");
plot.label(PLOT_AXIS_Y, "angles (degrees)");
plot.scattern(timedata, angledata1, numDataPoints);
plot.legend("Angle for joint 1");
plot.scattern(timedata, angledata3, numDataPoints);
plot.legend("Angle for joint 3");
plot.plotting();
```

Program 13.6: Move and record two joint angles at the same time using the function **recordAnglesBegin**() and **recordAnglesEnd**() to generate a scatter plot.

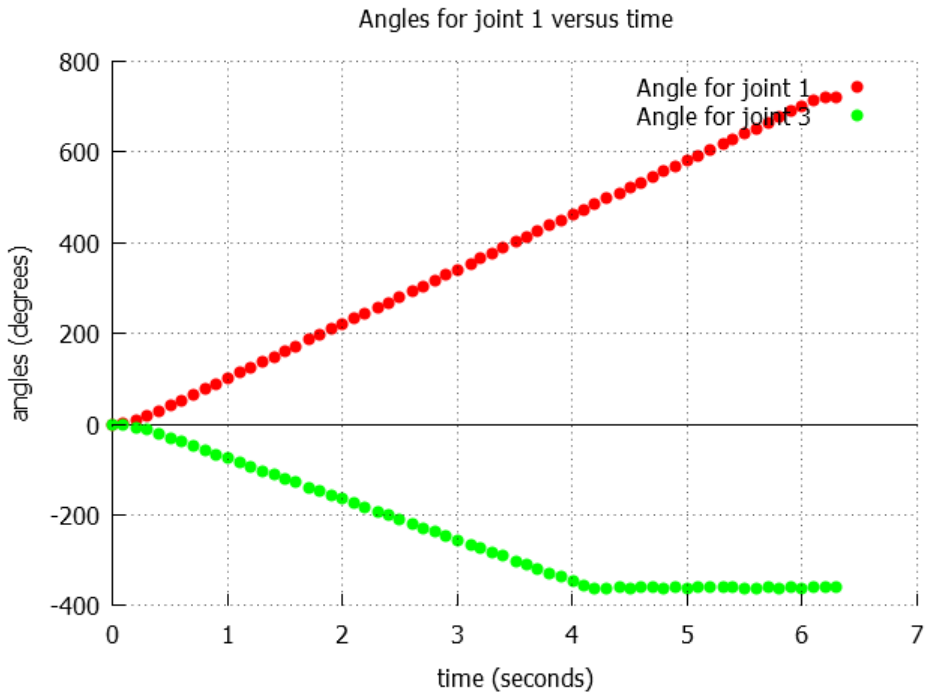


Figure 13.3: The scatter plot for joint angles versus time from Program 13.6.

Program 13.6 records the angles for both joints 1 and 2 of a Linkbot-I every 0.1 seconds. It then uses the **CPlot** member functions, as shown in Program 11.1, to plot the recorded joint angles versus time for this Linkbot.

Similar to the member functions **recordAngleBegin**() and **recordAngleEnd**(), the member functions **recordAnglesBegin**() and **recordAnglesEnd**() can be used to record joint angles for both joints 1 and 3 for Linkbot-I, or joints 1 and 2 for Linkbot-L, at the same time. The general syntax of the **CLinkbotI** member function **recordAnglesBegin**() is as follows To record angle data while the robot is moving, we need to use two new **CLinkbotI** member functions **recordAngleBegin**() and **recordAngleEnd**(). The general syntax of the **CLinkbotI** member function **recordAnglesBegin**() is as follows

```
robot.recordAnglesBegin(timedata, angledata1, adngledata3, timeInterval);
```

The argument `timedata` is used to record time values in seconds elapsed since the Linkbot-I starts to move, argument `angledata1` is used to record the angle values in degrees for joint 1, and argument `angledata3` is used to record the angle values in degrees for joint 3. The argument `timeInterval` specifies the amount of time between each recorded measurement. In Program 13.6, the line

```
robot.recordAnglesBegin(timedata, angledata1, angledata3, timeInterval);
```

starts the process of recording the angles of joints 1 and 3 every 0.1 seconds.

The motion of two wheels are accomplished by the following statements.

```
/* move joint 1 for 720 degrees */
robot.moveJointNB(JOINT1, 720);
/* move joint 3 for 360 degrees */
robot.moveJoint(JOINT3, 360);
robot.moveJointWait(JOINT1);
```

It is important to include this statement before the Linkbot-I starts moving, so that the program does not miss any data points.

After the Linkbot-I has stopped moving, we must use **recordAnglesEnd()** to stop the recording of data. The general syntax of the function **recordAngleEnd()** is as follows

```
robot.recordAnglesEnd(numDataPoints);
```

The argument `numDataPoints` keeps track of the total number of data points that were collected while the Linkbot-I was moving.

In Program 13.6, after the Linkbot-I has finished moving forward 360 degrees, the line

```
robot.recordAnglesEnd(numDataPoints);
```

stops the recording of data after the Linkbot-I has stopped moving forward. `numDataPoints` will now hold the total number of data points that were collected while the Linkbot-I was in motion. In addition, `angledata1` and `angledata3` will hold all the angle values that were recorded while the Linkbot-I was in motion, and `timedata` will hold all the corresponding time values.

The aquired data and legends are graphed as a scatter plot as shown in Figure 11.1 by the function call

```
plot.scattern(timedata, angledata1, numDataPoints);
plot.legend("Angle for joint 1");
plot.scattern(timedata, angledata3, numDataPoints);
plot.legend("Angle for joint 3");
```

E Do Exercise 2 on page 239.

13.4.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.recordAnglesBegin(timedata, angledata1, angledata3, timeInterval);
   ```

   when you want to begin recording joint angles for both joints of a Linkbot-I.

2. Call the **CLinkbotI** member function

   ```
   robot.recordAnglesEnd(numDataPoints);
   ```

   when you want to stop recording joint angle data from a Linkbot-I.

13.4.2 Terminology

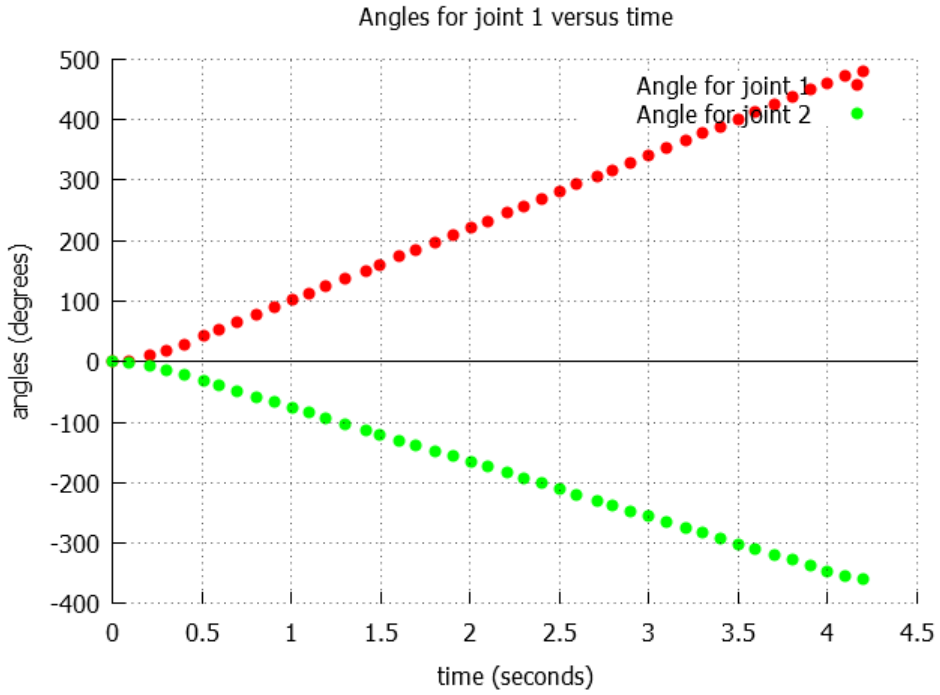
robot.recordAnglesBegin(), **robot.recordAnglesEnd()**.

13.4.3 Exercises

1. Watch the following video tutorial for Ch Linkbot Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Control Two Robots

2. Write a program `recordangles2.ch` to rotate joint 1 of a Linkbot-I for 520 degrees at 120 degrees per second and joint 3 for 400 degrees at -90 degrees per second, starting at the same time. The radius for joint 1 is 1.75 inches. The radius for joint 3 is 2 inches. Plot the joint angles for joints 1 and 3 versus time as shown below.



13.5 Synchronize the Motion of Multiple Linkbots

In the previous section we learned how to rotate joints 1 and 3 on a single Linkbot in different ways simultaneously. It is also possible to command multiple Linkbots in independent motions simultaneously using non-blocking functions. The motions of two Linkbot-Is are considered to be synchronized if they are distinct but simultaneous.

```
/* File: movenb.ch
   Control two robots with motions simultaneously using non-blocking functions. */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* move joints 1 and 3 for both robots at the same time using moveNB() */
robot1.moveNB(180, NaN, -180);
robot2.moveNB(360, NaN, -360);
/* wait for both robots to finish their movements through moveNB() */
robot1.moveWait();
robot2.moveWait();

/* move joints 1 and 3 for both robots at the same time using moveNB() */
robot1.moveNB(-360, NaN, 360);
robot2.moveNB(-360, NaN, 360);
/* wait for both robots to finish their movements through moveNB() */
robot1.moveWait();
robot2.moveWait();

/* hold the position after the program exits */
robot1.holdJointsAtExit();
robot2.holdJointsAtExit();
```

Program 13.7: Synchronize the motion of multiple Linkbots using non-blocking functions **moveNB**() and **moveWait**().

In Program 13.7 multiple Linkbots are moved simultaneously using the non-blocking **CLinkbotI** member function **moveNB()**. The lines

```
robot1.moveNB(180, NaN, -180);
robot2.moveNB(360, NaN, -360);
```

move `robot1` forward by 180 degrees and `robot2` forward by 360 degrees at the same time using **moveNB()**. The syntax of this function is

```
robot.moveNB(angle1, NaN, angle3);
```

The function **moveNB()** moves a Linkbot in the same way that **move()** does, except that it is non-blocking. This allows two Linkbot-Is to move both joints 1 and 3 relative to their current positions simultaneously. The arguments `angle1` and `angle3` correspond to joint 1 and 3 and specify the joint angle to move both joints relative to their current positions. Since joint 2 of a Linkbot-I cannot be moved, the second argument has the value NaN.

The next two lines call **moveWait()** on `robot1` and `robot2` to allow both to finish their motions before continuing. After that, the lines

```
robot1.moveNB(-360, NaN, 360);
robot2.moveNB(-360, NaN, 360);
```

move both `robot1` and `robot2` backward by 360 degrees simultaneously. Then **moveWait()** is called on both Linkbots to allow them to finish moving backward. The final lines

```
robot1.holdJointsAtExit();
robot2.holdJointsAtExit();
```

command `robot1` and `robot2` to hold their poses when the program exits.

E Do Exercises 1 and 2 on page 244.

So far we used non-blocking functions to move joints 1 and 3 of multiple Linkbots relative to their current positions. We can also use non-blocking functions to move the joints of multiple Linkbots to their absolute positions.

Problem Statement:
Write a program `movetonb.ch` to start the motions of two Linkbot-Is at the same time. It rotates joint 1 to 180 degrees and joint 3 to -180 degrees for the first Linkbot-I, joint 1 to 360 degrees and joint 3 to -360 degrees for the second Linkbot-I. Then, it moves joint 1 of both Linkbot-Is by 90 degrees. Next it moves joint 1 of both Linkbot-Is to 720 degrees.

```
/* File: movetonb.ch
   Control two robots with motions simultaneously using non-blocking functions. */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* move to the zero position at the same time. */
robot1.resetToZeroNB();
robot2.resetToZeroNB();

/* wait for both robots to finish their movements through resetToZeroNB() */
robot1.moveWait();
robot2.moveWait();

/* move joints 1 and 3 for both robots at the same time using moveNB() */
robot1.moveToNB(180, NaN, -180);
robot2.moveToNB(360, NaN, -360);
/* wait for both robots to finish their movements through moveNB() */
robot1.moveWait();
robot2.moveWait();

/* move joint 1 for both robots at the same time using moveJointNB() */
robot1.moveJointNB(JOINT1, 90);
robot2.moveJointNB(JOINT1, 90);
/* wait for both robots to finish their movements through moveJointNB() */
robot1.moveJointWait(JOINT1);
robot2.moveJointWait(JOINT1);

/* move joint 1 for both robots at the same time using moveJointToNB() */
robot1.moveJointToNB(JOINT1, 720);
robot2.moveJointToNB(JOINT1, 720);
/* wait for both robots to finish their movements through moveJointNB() */
robot1.moveJointWait(JOINT1);
robot2.moveJointWait(JOINT1);
```

Program 13.8: Synchronize the motion of multiple Linkbots using non-blocking functions **resetToZeroNB**(), **moveToNB**(), **moveJointToNB**(), and **moveJointNB**().

Program 13.8 moves joints 1 and 3 of multiple Linkbot-Is to different absolute positions simultaneously. This is accomplished using the non-blocking **CLinkbotI** member functions **moveToNB()** and **moveJointToNB()**.

As mentioned in section 10.3, moving a joint to an absolute position, the function call

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

for a single robot, or

```
robot.resetToZeroNB();
```

for multiple robots shall be used to set all joints to their zero positions. The general syntax of the function **resetToZeroNB()** is

```
robot.resetToZeroNB();
```

The function **resetToZeroNB()** has no argument. This function allows multiple Linkbots to reset their joints to the zero position at the same time instead of one at a time.

The general syntax of the function **moveToNB()** is

```
robot.moveToNB(angle1, NaN, angle3);
```

This function uses arguments `angle1` and `angle3` to store the joint angle values for joints 1 and 3, respectively. The function **moveToNB()** moves joints 1 and 3 of a Linkbot-I to their absolute positions. It is the non-blocking counterpart of the function **moveTo()** that was introduced in Section 10.3. In Program 13.8, the lines

```
robot1.moveToNB(180, NaN, -180);
robot2.moveToNB(360, NaN, -360);
```

move `robot1` forward to an absolute position of 180 degrees and move `robot2` forward to an absolute joint position of 360 degrees. Both Linkbot-Is perform these motions simultaneously. The next two lines call **moveWait()** to pause the program until `robot1` and `robot2` finish their motions. The next lines

```
robot1.moveJointNB(JOINT1, 90);
robot2.moveJointNB(JOINT1, 90);
```

move joint 1 of `robot1` and `robot2` simultaneously 90 degrees relative to their current positions. The next two lines use **moveJointWait()** on both Linkbots to pause the program until both have finished their motions.

The next lines

```
robot1.moveJointToNB(JOINT1, 720);
robot2.moveJointToNB(JOINT1, 720);
```

move joint 1 of both `robot1` and `robot2` simultaneously to the absolute position of 720 degrees using the function **moveJointToNB()**. The syntax of this function is

```
robot.moveJointToNB(id, angle);
```

The argument `id` specifies the joint angle to move. The argument `angle` specifies the absolute position to move the joint. The function **moveJointToNB()** is the non-blocking counterpart of the function **moveJointTo()**, which was introduced in section 10.6. It rotates a single joint of a Linkbot to its absolute position and allows the next line of code to execute before the joint finishes its rotation.

Finally, **moveJointWait()** is used on both Linkbots to pause the program until both have finished their 720 degree rotations.

E Do Exercise 3 on page 244.

13.5.1 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.resetToZeroNB();
   ```

   on multiple Linkbot-Is to simultaneously reset their joints to the zero position.

2. Call the non-blocking **CLinkbotI** member function

```
robot.moveNB(angle1, NaN, angle3);
```

on multiple Linkbot-Is to simultaneously move them by different degrees relative to their current joint angle positions.

3. Call the non-blocking **CLinkbotI** member function

```
robot.moveToNB(angle1, NaN, angle3);
```

on multiple Linkbot-Is to simultaneously move them to by different degrees to their absolute joint angle positions.

4. Call the non-blocking **CLinkbotI** member function

```
robot.moveJointToNB(id, angle);
```

on multiple Linkbots to simultaneously move single joints by different degrees to their absolute joint angle positions.

13.5.2 Terminology

robot.resetToZeroNB(), **robot.moveNB()**, **robot.moveToNB()**, **robot.moveJointToNB()**.

13.5.3 Exercises

1. Modify the program `movenb.ch` in Program 13.7 as a new program `nomovewait.ch` by removing the last two **moveWait()** statements. How will the motion change?

2. Write a program `movenb2.ch` using member functions **moveJoint**() and **moveJointNB**() to move joints 1 and 3 by 360 degrees at the same time.

3. Write a program `movetonb2.ch` with the synchronized motions for both Linkbot-Is. It first rotates joints 1 and 3 to 360 degrees. Then, rotate joint 1 by 90 degrees. Finally, rotate joint 3 to 720 degrees.

13.6 Move Multiple Linkbots with Specified Distances or Joint Angles

In section 8.1, we learned how to drive a Linkbot-I with the specified speed and distance using the member function **driveDistance**(). In section 12.5, we learned how to write a program to drive a robot and play a melody at the same time using the non-blocking member function **driveDistanceNB**(). The non-blocking member function **driveDistanceNB**() can also be used to drive multiple Linkbot-Is at the same time.

Special consideration may need to be given to when it is more appropriate to use a blocking or a non-blocking function to drive a particular Linkbot. In addition, the programmer must pay attention to whether some of the final functions are time based. If this is the case, then either the last function must be blocking or a waiting function such as moveWait() must be added to wait for the completion of the motion for a non-blocking function. How to drive two Linkbots simultaneously will be illustrated by solving the following problem and corresponding Program 13.9.

Problem Statement:
Two Linkbot-Is are configured as two-wheel robots, as shown in Figure 13.4. The radii of the wheels are 1.75 inches. Write a program `drivedistancenb.ch` for two Linkbots racing in the following manner. Both Linkbots will drive from a starting line at different times. The first Linkbot drives for 24 inches at the speed of 1.5 inches per second. Then, 8 seconds after the first Linkbot has left the starting line, the second Linkbot races for 24 inches at the speed of 3 inches per second.

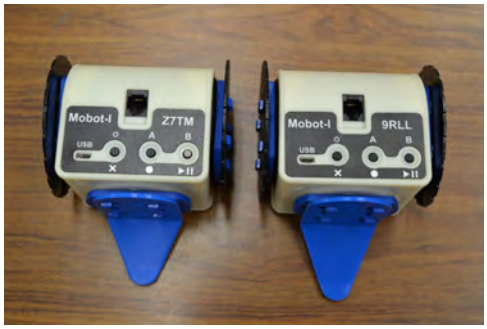


Figure 13.4: Two two-wheel drive vehicles.

When two Linkbot-Is are placed on a starting line, make sure to press both buttons A and B together for each Linkbot to reset the wheels to the zero positions before running the program, as described in Section 2.1.3. Therefore, two Linkbots will drive at the same starting position.

```
/* File: drivedistancenb.ch
   Drive two two-wheeled robots with different speeds continuously
   with specified distances. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double radius1=1.75; // the radius of the two wheels of robot1 in inches
double radius2=1.75; // the radius of the two wheels of robot2 in inches
double speed1=1.5, speed2=3; // speed of robots in inches per second
double distance1=24, distance2=24; // the traveled distances for robot1 and robot2
double delaytime=8;            // delay time in seconds for robot2

/* set the speed for robot1 */
robot1.setSpeed(speed1, radius1);
/* set the speed for robot2 */
robot2.setSpeed(speed2, radius2);

/* robot1 drives for 'distances1'
 'delaytime' seconds later, robot2 drives for 'distance2' while robot1 also drives */
robot1.driveDistanceNB(distance1, radius1);
robot2.delaySeconds(delaytime);
robot2.driveDistance(distance2, radius2);
robot1.moveWait();  // wait till robot1 drove 'distance1'
```

Program 13.9: Driving multiple Linkbots with specified distances using **driveDistanceNB**().

Program 13.9 drives two Linkbots by specified distances simultaneously. The lines

```
    double radius1=1.75; // the radius of the two wheels of robot1 in inches
    double radius2=1.75; // the radius of the two wheels of robot2 in inches
```

declare two variables of **double** type and assign them both the wheel radius value of 1.75 inches. The next line

```
    double speed1=1.5, speed2=3; // speed of robots in inches per second
```

declares two variables of type **double** and assigns them two different speeds in inches per second. Note that the speed of the second Linkbot-I is twice the speed of the first. Then the next two lines

```
    double distance1=24, distance2=24; // the traveled distances for robot1 and robot2
    double delaytime=8; // delay time in seconds for robot2
```

declare two variables `distance1` and`distance2` of type **double** and assign them with distances of 24 inches. Note that the distances for both robots are the same. Another variable `delaytime` of type **double** is also declared, and assigned a value of 8 seconds. This will be used to delay the start of `robot2` for 8 seconds after `robot1` starts moving.

The next six lines of code connect both `robot1` and `robot2` to Linkbots and reset both to the zero position. The program is paused by calling **moveWait()** for both `robot1` and `robot2` until both have finished resetting. Then the next two lines

```
    robot1.setSpeed(speed1, radius1);
    robot2.setSpeed(speed2, radius2);
```

sequentially sets the speeds of `robot1` and `robot2`. The following line

```
     robot1.driveDistanceNB(distance1, radius1);
```

drives the `robot1` for a specific distance `distance1` with the radius of `radius1`. This is done using the **CLinkbotI** member function **driveDistanceNB()**. The syntax of the member function **driveDistanceNB**() is as follows.

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

The line

```
    robot2.delaySeconds(delaytime);
```

delays the start of `robot2` for 8 seconds after `robot1` starts moving. Since `robot1` has a total movement distance of 24 inches, however, it will only be half way through its movement when `robot2` starts to move. This makes the choice of **driveDistanceNB()** appropriate for `robot1`. The next line

```
   robot2.driveDistance(distance2, radius2);
```

drives `robot2` forward for 24 inches at 3 inches per second. `robot2` is the last Linkbot in the program to begin moving, and its motion depends on a specified distance and speed. Thus, the choice of a blocking function is appropriate for `robot2`. After `robot2` has completed its motion, the line

```
    robot1.moveWait(); // wait until robot1 moved 'distance1'
```

is included to make sure that `robot1` has completed its motion before the program finishes execution. In this case, since both robots drive the same distance of 24 inches. This synchronization statement may not be critical. However, if the traveling `distance2` for `robot2` is less than that for `robot1`, say 16 inches, without the last statement `robot1.moveWait()`, when `robot2` finishes its motion and the program exits, `robot1` would drive only 16 inches, instead of 24 inches.

The distance traveled by two Linkbot-Is is the same. Since the speed of the second Linkbot-I is twice as fast as the first one, both Linkbot-Is shall stop at the same finishing line at the same time. For the first robot, the time to drive 24 inches can be calculated as follows.

$$t = \frac{distance}{v} = \frac{24}{1.5} = 16\ seconds$$

The time for the second robot to drive 24 inches can be calculated as follows.

$$t = \frac{distance}{v} = \frac{24}{3} = 8\ seconds$$

Do Exercises 1, 2, and 3 and on page 247.

There are also similar non-blocking member functions

```
robot.driveAngleNB(angle);
robot.driveAngleNB(-angle);
robot.turnLeftNB(angle, radius, trackwidth);
robot.turnRightNB(angle, radius, trackwidth);
```

to roll forward, roll backward, turn left, and turn right a Linkbot-I, respectively. Programs `driveanglenb.ch` and `turnleftnb.ch` illustrate how to use these non-blocking member functions. These two programs are distributed along with other sample programs.

Do Exercises 4 and 5 and on page 248.

13.6.1 Summary

1. Call the non-blocking **CLinkbotI** member function

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

   to drive a Linkbot-I for the specified distance. The lines of code following the non-blocking member function will begin execution before the specified distance has reached.

2. Call the non-blocking **CLinkbotI** member functions **driveAngleNB**(), **turnLeftNB**(), and **turnRightNB**() to roll forward or backward, turn left, and turn right a Linkbot-I, respectively. The lines of code following the non-blocking member function will begin execution before the specified angle has reached.

13.6.2 Terminology

robot.driveDistanceNB(), **robot.driveAngleNB**(), **robot.turnLeftNB**(), and **robot.turnRightNB**().

13.6.3 Exercises

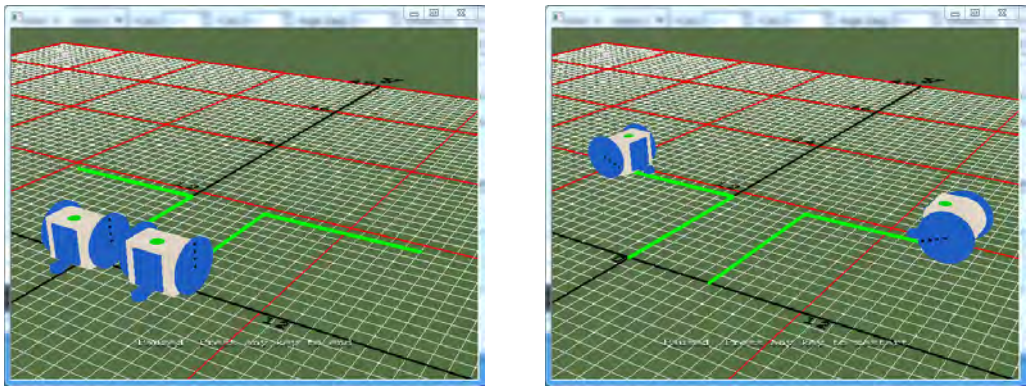
1. Watch the following video tutorials for RoboBlockly in http://roboblockly.ucdavis.edu/videos/.

   (a) R8. Program Multiple Robots of Linkbot and Lego Mindstorms

   (b) R9. Move Multiple Robots Simultaneously: driveDistanceNB() and moveWait()

   (c) R10. Robots Chase Each Other: delaySeconds()

2. Two Linkbot-Is are configured as two-wheel robots, as shown in Figure 13.4. The radii of the wheels are 1.75 inches. Write a program `drivedistancenb2.ch` for two Linkbots racing in the following manner. Both Linkbots will drive from a starting line at different times. The first Linkbot drives for 12 inches at the speed of 1.2 inches per second. Then 5 seconds after the first Linkbot has left the starting line, the second Linkbot races for 12 inches at the speed of 2.4 inches per second.



(a) The starting and ending positions.

(b) The position reached by two Linkbot-Is.

Figure 13.5: The trajectories for two Linkbot-Is.

3. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels for the first Linkbot-I are 1.75 inches. The radii of the wheels for the second Linkbot-I are 2 inches. Write a program `drivedistancenb3.ch` for two Linkbot-Is racing in the following manner. Two Linkbots will drive from a starting line at the same time at the same speed of 1.5 inches per second for 20 inches.

4. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels for the first Linkbot-I are 1.75 inches. The radii of the wheels for the second Linkbot-I are 2 inches. Write a program `driveanglenb2.ch` for two Linkbot-Is racing in the following manner. Two Linkbots will roll forward for 720 degrees from a starting line at the same time at the same joint speed of 45 degrees per second. You can set the joint speed of a Linkbot-I by the member function **setJointSpeeds**() and using the member functions **driveAngleNB**() and **driveAngle**() to roll two Linkbots at the same time in RoboSim. What is the distance traveled by each Linkbot-I.

5. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels for both Linkbot-Is are 1.75 inches. The track width of the Linkbot-I is 3.69 inches. Write a program `turnleftnb2.ch` for two Linkbot-Is moving in the following manner simultaneously. The joint speeds for Linkbot-Is are set to 45 degrees per second. Two Linkbots will roll forward for 360 degrees from a starting line at the same time. Next, the first robot turns left 90 degrees and the second robot turns right 90 degrees. Then, both robots roll forward for 360 degrees. Finally, both robots drive back to the originally positions along the same paths as shown in Figure 13.5.

13.7 Plot Recorded Distances and Time for Multiple Linkbots

In Section 8.3.1 we learned how to record and plot distance versus time for a single Linkbot-I with a specified speed and distance. In this section we will learn how to do this for two Linkbot-Is in synchronous motion with a specified speed and distance or joint angle. The ability to receive and process real time data from multiple sources is valuable for comparing the performance of multiple robots. Program 13.10 extends the concepts learned from Program 8.9, except that plotting is done for multiple Linkbot-Is and non-blocking functions are used.

Problem Statement:
Two Linkbot-Is are configured as two-wheel robots, as shown in Figure 13.4 on page 245. The radii of the wheels are 1.75 inches. Write a program `recorddistancesdist.ch` to drive the first robot at the speed of 1.5 inches per second for 24 inches. After 8 seconds delay, the second robot drives at the speed of 3 inches per second for 24 inches. Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances for both robots versus time.

We can solve this problem conveniently using Ch Linkbot Controller (CLC) with the setup shown in Figure 13.6. To control two robots, we need to "Two Vehicle Control" in CLC.

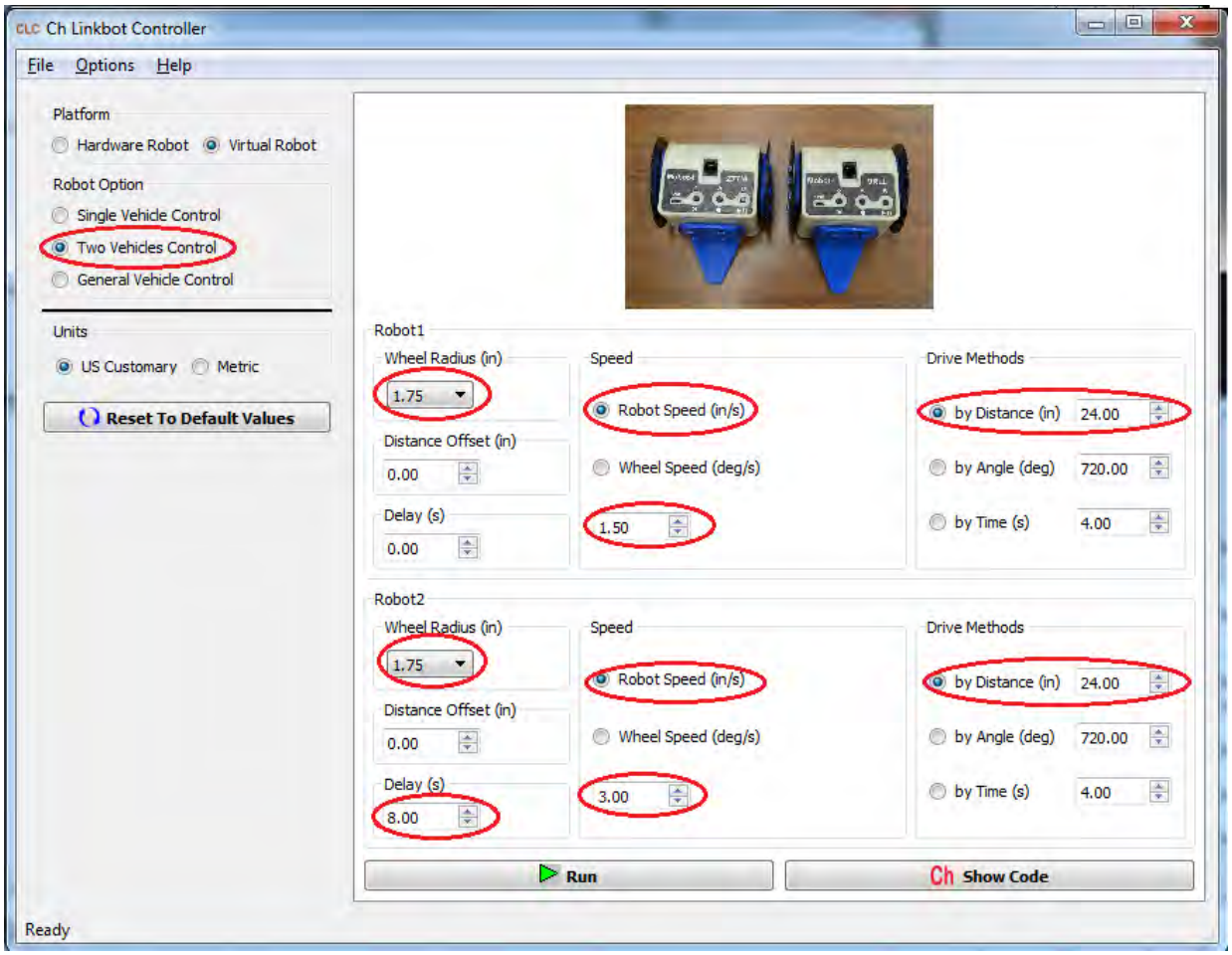


Figure 13.6: The setup of Ch Linkbot Controller for solving the problem.

E Do Exercise 1 on page 253.

```
/* File: recorddistancesdist.ch
   Record time and distances using driveDistance(),
   Delay 8 seconds for robot2. The equations of motions are
          d = 1.5t
          d = 3(t-8)
   Plot the acquired data for two robots */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot1, robot2;
double speed1=1.5, speed2=3; // speeds of robots in inches per second
double radius1=1.75, radius2=1.75; // the radii of two wheels of robot1 and 2
double distance1=24, distance2=24; // the traveled distances for robot1 and robot2
double delaytime=8;          // delay time for robot2
double timeInterval = 0.1;   // time interval in 0.1 second
int numDataPoints1, numDataPoints2; // number of data points recorded
robotRecordData_t timedata1, distances1; // recorded time and distances for robot1
robotRecordData_t timedata2, distances2; // recorded time and distances for robot2
CPlot plot;                  // plotting class

/* move to the zero position at the same time. */
robot1.resetToZeroNB();  robot2.resetToZeroNB();
robot1.moveWait();       robot2.moveWait();

/* set the speeds for robot1 and robot2 */
robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

/* disable record data shift and begin recording time and distance */
robot1.recordNoDataShift();
robot2.recordNoDataShift();
robot1.recordDistanceBegin(timedata1, distances1, radius1, timeInterval);
robot2.recordDistanceBegin(timedata2, distances2, radius2, timeInterval);

/* robot1 drives for 'distances1'
 'delaytime' seconds later, robot2 drives for 'distance2' while robot1 also drives */
robot1.driveDistanceNB(distance1, radius1);
robot2.delaySeconds(delaytime);
robot2.driveDistance(distance2, radius2);
robot1.moveWait();  // wait till robot1 moved 'distance1'

/* end recording time and distance */
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);

/* plot the data */
plot.title("Distances versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distances (inches)");
plot.data2DCurve(timedata1, distances1, numDataPoints1);
plot.legend("Distance for robot 1");
plot.data2DCurve(timedata2, distances2, numDataPoints2);
plot.legend("Distance for robot 2");
plot.plotting();
```

Program 13.10: Recording the distances of two Linkbot-Is and plotting the distances versus time. The distances for both robots are specified.

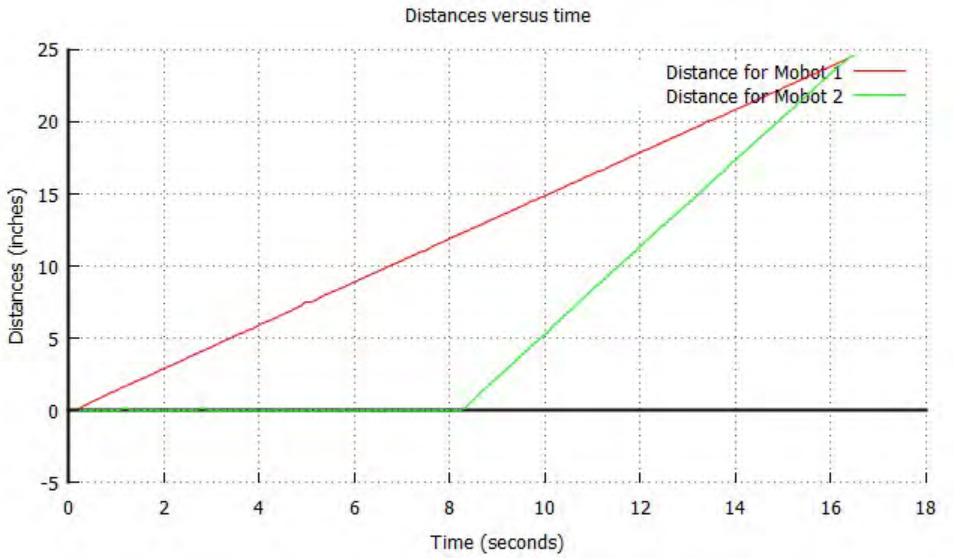


Figure 13.7: The plot for the distances of two Linkbot-Is versus time from Programs 13.10, 13.12, and 13.14.

The member function **recordDistanceBegin()** described in Section 8.3.1 can be used to record the time and distances for a robot. By default, only data points collected while the Linkbot-I is in motion will be stored in the second argument for the time and the third argument for the distance of the function **recordDistanceBegin()** for later use. Any data collected before the Linkbot-I starts moving or after it has stopped will be discarded. Effectively, the recorded data is shifted. Since we want to compare the distance versus time for both Linkbot-Is over the entire 16 second time period, we want to include the data points for `robot2` in the first 8 seconds even when it is standing still. Thus we do not want to shift the data, since we do not want the program to discard those data points. The default shifting of the recorded data can be disabled by the member function **recordNoDataShift()**. The general syntax for the function **recordNoDataShift()** is

```
robot.recordNoDataShift();
```

Similarly, the shifting can be enabled by **recordDataShift()**. The general syntax for the function **recordDataShift()** is

```
robot.recordDataShift();
```

The two lines

```
robot1.recordNoDataShift();
robot2.recordNoDataShift();
```

in Program 13.10 disable the shifting for recording data for both `robot1` and `robot2`. The two lines

```
robot1.recordDistanceBegin(timedata1, distances1, radius1, timeInterval);
robot2.recordDistanceBegin(timedata2, distances2, radius2, timeInterval);
```

initiate the recording of time and distance data from both `robot1` and `robot2`. Then `robot1` and `robot2` are commanded to drive the same way they were commanded in Program 13.9. After both `robot1` and `robot2` finished their movement, the lines

```
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);
```

stop the recording of time and distance data from both `robot1` and `robot2`. At this time, `timedata1`, `distances1` and `numDataPoints1` will hold the time data points, the distance data points, and the total number of data points collected for `robot1`. In addition, the variables `timedata2`, `distances2`, and `numDataPoints2` will hold the time data points, the distance data points, and the total number of data points collected for `robot2`.

The plotting of data for both Linkbot-Is will occur in a similar manner as in previous sections. Data for both Linkbot-Is will be plotted on the same graph. The title as well as the x-axis and y-axis labels will be the same as in Program 8.9. The **CPlot** member function **data2Curve()**, however, will need to be called twice. The line

```
plot.data2DCurve(timedata1, distances1, numDataPoints1);
```

plots the data for `robot1`. The next line

```
plot.legend("Distance for robot 1");
```

creates a legend entry for `robot1` using the **CPlot** member function **legend()**. Each entry of the legend will show a different color which corresponds to the data of a particular Linkbot. The next two lines

```
plot.data2DCurve(timedata2, distances2, numDataPoints2);
plot.legend("Distance for robot 2");
```

plots the data for `robot2` and creates a corresponding legend entry. The final graph is generated by calling the **CPlot** member function **plotting()**. The resulting graph of distance versus time for both Linkbot-Is can be seen in Figure 13.7.

The first robot drives at the speed of 1.5 inches per second. The relation between the distance (d) and time (t) in Figure 13.7 for the first robot can be formulated by the following linear equation.

$$d = 1.5t \tag{13.1}$$

The second robot drives at the speed of 3 inches per second and starts to drive 8 seconds later. The relation between the distance (d) and time (t) in Figure 13.7 for the second robot can be formulated by the following linear equation.

$$d = 3(t - 8) \tag{13.2}$$

The intersection point $(t, d) = (16, 24)$ of two straight lines in Figure 13.7 satisfies both equations (13.1) and (13.2). The coordinates of the intersection point are the time and location when two robots have traveled the same distance. In algebra, $(t, d) = (16, 24)$ is called the solution of the system of two linear equations (13.1) and (13.2).

Do Exercise 2 on page 253.

13.7.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.recordNoDataShift();
   ```

   to disable the shifting of the recorded data.

2. Call the **CLinkbotI** member function

   ```
   robot.recordDataShift();
   ```

   to enable the shifting of the recorded data.

13.7.2 Terminology

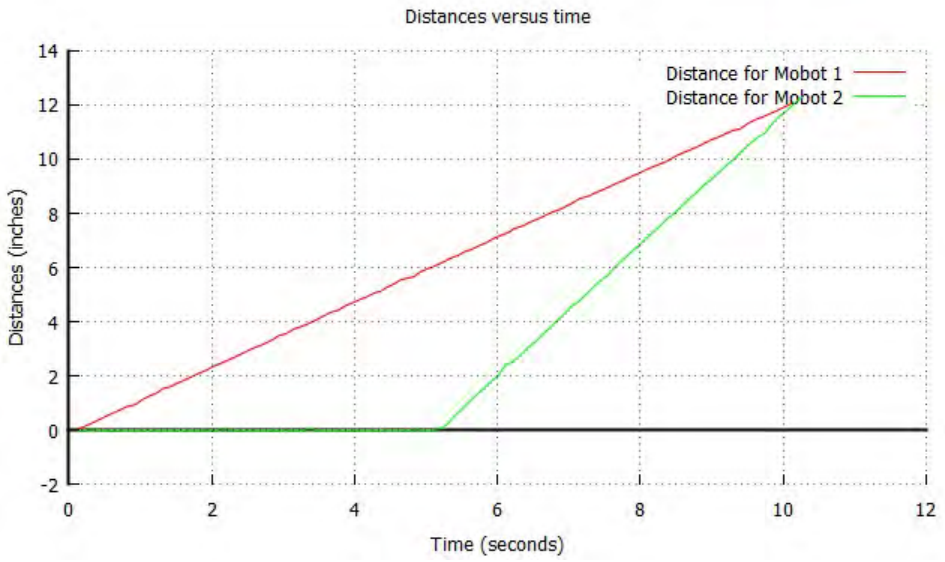
robot.recordNoDataShift(), robot.recordDataShift(), multiple data sets.

13.7.3 Exercises

1. Watch the following video tutorial for Ch Linkbot Controller in http://c-stem.ucdavis.edu/studio/tutorial/.

   (a) Control Two Robots

2. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. Write a program `recorddistancesdist2.ch` to control two Linkbot-Is with the following motions. The first robot drives at the speed of 1.2 inches per second for 12 inches. The 5 seconds after the first robot drives, the second robot drives at the speed of 2.4 inches per second for 12 inches. Record the distances and time during the motion of both robots. Plot the distances versus time as shown in the figure below. What are the equations for the linear relations shown in the figure? Verify that the point $(t, d) = (10, 12)$ is the solution for the two equations of the linear relations.



13.8 Move Multiple Linkbots with Specified Time

In Section 8.4, we learned how to drive a single Linkbot for a specified time using **driveTime()**. In some applications, we need to move multiple Linkbots synchronously for a specified time, using non-blocking functions. The following problem and corresponding Program 13.11 demonstrate how this is done.

Problem Statement:
Two Linkbot-Is are configured as two-wheel robots as shown in Figure 13.4 on page 245, The radii of the wheels are 1.75 inches. Write a program `drivetimenb.ch` for two Linkbots racing in the following manner. Both Linkbots will drive from a starting line at different times. The first Linkbot drives for 16 seconds at the speed of 1.5 inches per second. Then 8 seconds

after the first Linkbot has left the starting line, the second Linkbot races at the speed of 3 inches per second for 8 seconds.

```
/* File: drivetimenb.ch
   Drive two two-wheeled robots with different speeds continuously
   with a specified time. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double radius1=1.75; // the radius of the two wheels of robot1 in inches
double radius2=1.75; // the radius of the two wheels of robot2 in inches
double speed1=1.5, speed2=3; // speed of robots in inches per second
double time1=16;               // motion time in seconds for robot1
double delaytime=8;            // delay time in seconds for robot2
double time2=8;                // motion time in seconds for robot2

/* set the speed for robot1 */
robot1.setSpeed(speed1, radius1);
/* set the speed for robot2 */
robot2.setSpeed(speed2, radius2);

/* robot1 drives first for a total of 'time1' seconds. 'delaytime' seconds later,
   robot2 drives for 'time2' seconds while robot1 also drives */
robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveTime(time2);
robot1.moveWait();  // wait till robot1 moved 'time1' seconds
```

Program 13.11: Moving multiple Linkbots with specified times using **driveTimeNB**().

Similar to Program 13.9, Program 13.11 sets the movement state for joints 1 and 3 of multiple Linkbots simultaneously. The lines

```
    double radius1=1.75; // the radius of the two wheels of robot1 in inches
    double radius2=1.75; // the radius of the two wheels of robot2 in inches
```

declare two variables of **double** type and assigns them both the wheel radius value of 1.75 inches. The next line

```
    double speed1=1.5, speed2=3; // speed of robots in inches per second
```

declares two variables of type **double** and assigns them two different speeds in inches. Note that the speed of the second Linkbot-I is twice the speed of the first. Then the next three lines

```
    double time1=16; // motion time in seconds for robot1
    double delaytime=8; // delay time in seconds for robot2
    double time2=8; // motion time in seconds for robot2
```

declare two variables `time1` and`time2` of type **double** and assigns them two different motion times in seconds. Note that the motion time for second Linkbot-I is half the motion time for the first. Another variable `delaytime` of type **double** is also declared, and assigned a value of 8 seconds. This will be used to delay the start of `robot2` for 8 seconds after `robot1` starts moving.

The next six lines of code connect both `robot1` and `robot2` to Linkbots and resets both to the zero position. Then the next two lines

```
    robot1.setSpeed(speed1, radius1);
    robot2.setSpeed(speed2, radius2);
```

sequentially sets the speeds of `robot1` and `robot2`. The following line

```
robot1.driveTimeNB(time1);
```

sets the movement state of `robot1` for a specific time. This is done using the **CLinkbotI** member function **driveTimeNB()**. The syntax of this function is

```
robot.driveTimeNB(seconds);
```

The argument `seconds` specifies the length of time in seconds to drive the joints of the robot. The direction of the motion for each joint is determined by the speed of the joint.

Since the speeds for both robots are positive. the `robot1` starts moving forward for 16 seconds at 1.5 inches per second. The member function **driveTimeNB()** is the non-blocking counterpart of the function **driveTime()**. It moves joints 1 and 3 of a Linkbot, but it allows the next line of code to execute before it finishes doing so. Therefore the next line of code

```
robot2.delaySeconds(delaytime);
```

delays the start of `robot2` for 8 seconds after `robot1` starts moving. Since `robot1` has a total movement time of 16 seconds, however, it will only be half way through its movement when `robot2` starts to move. This makes the choice of **driveTimeNB()** appropriate for `robot1`. The next line

```
robot2.driveTime(time2);
```

drives `robot2` forward for 8 seconds at 3 inches per second. `robot2` is the last Linkbot in the program to begin moving, and its motion depends on a specific time. Thus the choice of a blocking function is appropriate for `robot2`. After `robot2` has completed its motion, the line

```
robot1.moveWait(); // wait unti  robot1 moved time1 seconds
```

is included to make sure that `robot1` has completed its motion before the program finishes execution.

Since the speed of the second Linkbot-I is twice as fast as the first one, both Linkbot-Is shall stop at the same finishing line at the same time. The distance traveled by two Linkbot-Is is the same. For the first robot, the distance can be calculated as follows.

$$distance = v * t = 1.5 * 16 = 24\ inches$$

The distance for the second robot can be calculated as follows.

$$distance = v * t = 3 * 8 = 24\ inches$$

There is also a non-blocking function to move a single joint for a specified time. The **CLinkbotI** member function **moveJointTimeNB()** has the following general syntax

```
robot.moveJointTimeNB(id, seconds);
```

The argument `id` indicates the joint to move. The argument `seconds` specifies the length of time in seconds to keep the joint in motion. If in Program 13.11 we changed the line

```
robot1.driveTimeNB(time1);
```

to

```
robot1.moveJointTimeNB(JOINT1, time1);
```

then only joint 1 of `robot1` would move for 16 seconds at 1.5 inches per second. Since **moevJointTimeMB()** is non-blocking, the next subsequent lines of code will execute simultaneously.

E Do Exercises 1 and 2 on page 258.

Section 8.4.3 deals with the plotting of distance versus time with a specified speed and time. We can also plot distances versus time with a specified speed and time for two Linkbots. Similar to Programs 13.9 and

13.10, Program 13.11 can be modified to add the data acquisition and plotting code to display the distances versus time. Program 13.12 is very similar to Program 13.10, except that the movement states for both Linkbot-Is are set for a specified time using the member functions **driveTimeNB**() and **driveTime**(). The lines of code moving robots are the same as those from Program 13.11. The plot generated by Program 13.12 looks identical to that generated by Program 13.10, as shown in Figure 13.7. The distances of two Linkbots and time are related by the equations (13.1) and (13.2) as described on page 252.

```
/* File: recorddistancestime.ch
   Record time and distances using driveTime(),
   Delay 8 seconds for robot2. The equations of motions are
          d = 1.5t
          d = 3(t-8)
   plot the acquired data for two robots */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot1, robot2;
double speed1=1.5, speed2=3; // speeds of robots in inches per second
double radius1=1.75, radius2=1.75; // the radii of two wheels of robot1 and 2
double time1=16, time2=8;    // motion time in seconds for robot1 and robot2
double delaytime=8;          // delay time for robot2
double timeInterval = 0.1;   // time interval in 0.1 second
int numDataPoints1, numDataPoints2; // number of data points recorded
robotRecordData_t timedata1, distances1; // recorded time and distances for robot1
robotRecordData_t timedata2, distances2; // recorded time and distances for robot2
CPlot plot;                  // plotting class

/* move to the zero position at the same time. */
robot1.resetToZeroNB();  robot2.resetToZeroNB();
robot1.moveWait();       robot2.moveWait();

/* set the speeds for robot1 and robot2 */
robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

/* disable record data shift and begin recording time and distance */
robot1.recordNoDataShift();
robot2.recordNoDataShift();
robot1.recordDistanceBegin(timedata1,distances1,radius1,timeInterval);
robot2.recordDistanceBegin(timedata2,distances2,radius2,timeInterval);

/* robot1 drives first for a total of 'time1' seconds. 'delaytime' seconds later,
   robot2 drives for 'time2' seconds while robot1 also drives */
robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveTime(time2);
robot1.moveWait();  // wait till robot1 moved 'time1' seconds

/* end recording time and distance */
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);

/* plot the data */
plot.title("Distances versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distances (inches)");
plot.data2DCurve(timedata1, distances1, numDataPoints1);
plot.legend("Distance for robot 1");
plot.data2DCurve(timedata2, distances2, numDataPoints2);
```

```
plot.legend("Distance for robot 2");
plot.plotting();
```

Program 13.12: Recording the distances of two Linkbot-Is and plotting the distances versus time. The movement states of the Linkbot-Is are set with a specified time.

E Do Exercises 3 and 4, on page 266.

If the speeds of two robots are the same, the two lines for distances versus time will be parallel. We can change

```
    double speed1=1.5, speed2=3;    // speeds of mobots in inches per second
```

in Program 13.12 to

```
    double speed1=1.5, speed2=1.5;  // speeds of mobots in inches per second
```

and save the new program as `recordparalleltime.ch`. This program is distributed along with other sample programs. In this program, the speed for the second robot is the same as the first one, 1.5 inches per second. When the program is executed, it will display the output as shown in Figure 13.8.

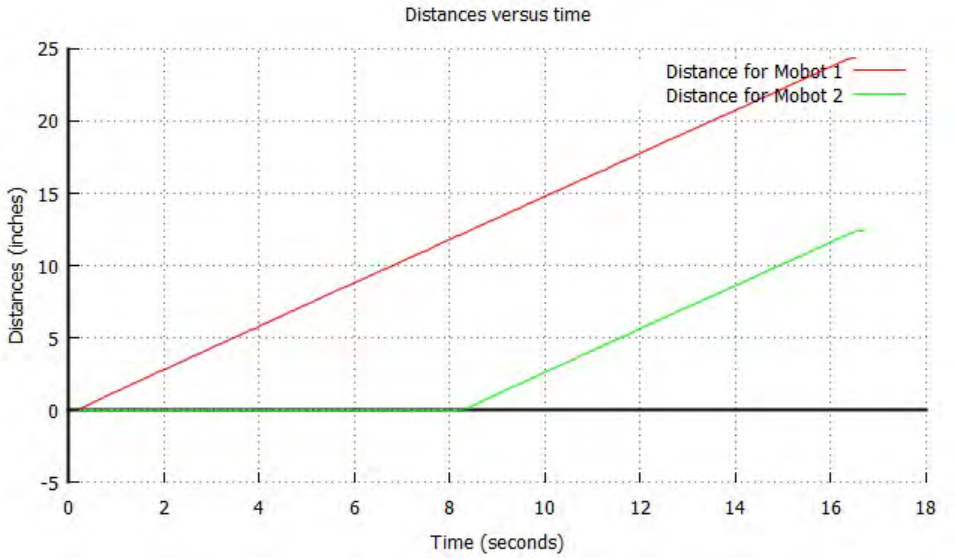


Figure 13.8: The plot for the distances of two Linkbot-Is moving at the same speed.

E Do Exercise 5 on page 259.

13.8.1 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.driveTimeNB(seconds);
   ```

   to move joints of a Linkbot for the specified time in seconds. The lines of code following the non-blocking member function will begin execution before the specified time in seconds has elapsed.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveJointTimeNB(id, seconds);
   ```

to move one joint of a Linkbot-I for a specified time. The lines of code following the non-blocking member function will begin execution before the specified time in seconds has elapsed.

13.8.2 Terminology

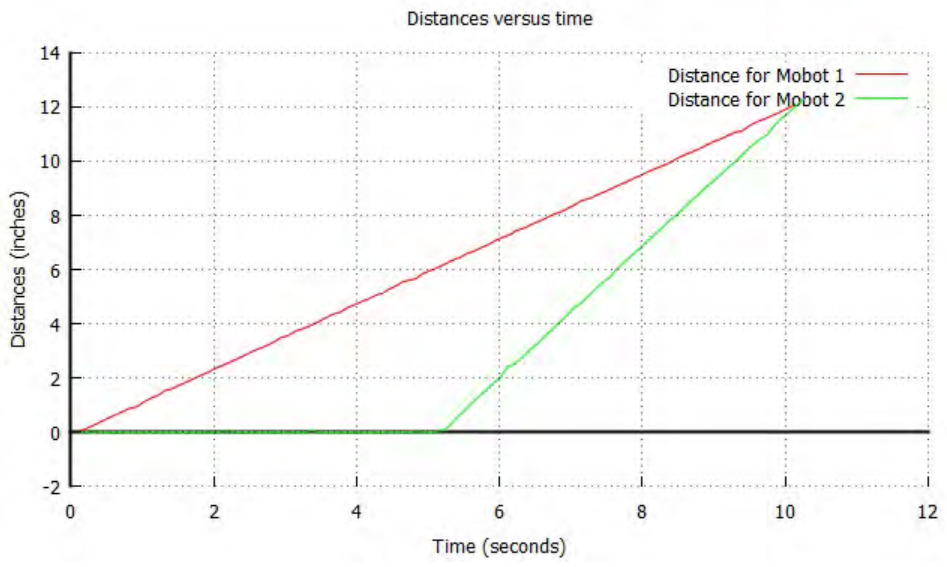
robot.driveTimeNB(), **robot.moveJointTimeNB()**.

13.8.3 Exercises

1. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. Write a program `drivetimenb2.ch` for two Linkbot-Is racing in the following manner. Two Linkbot-Is will drive from a starting line at different times. The first Linkbot-I drives for 10 seconds at the speed of 1.2 inches per second. Then 5 seconds after the first Linkbot-I has left the starting line, the second Linkbot-I races at the speed of 2.4 inches per second for 5 seconds.

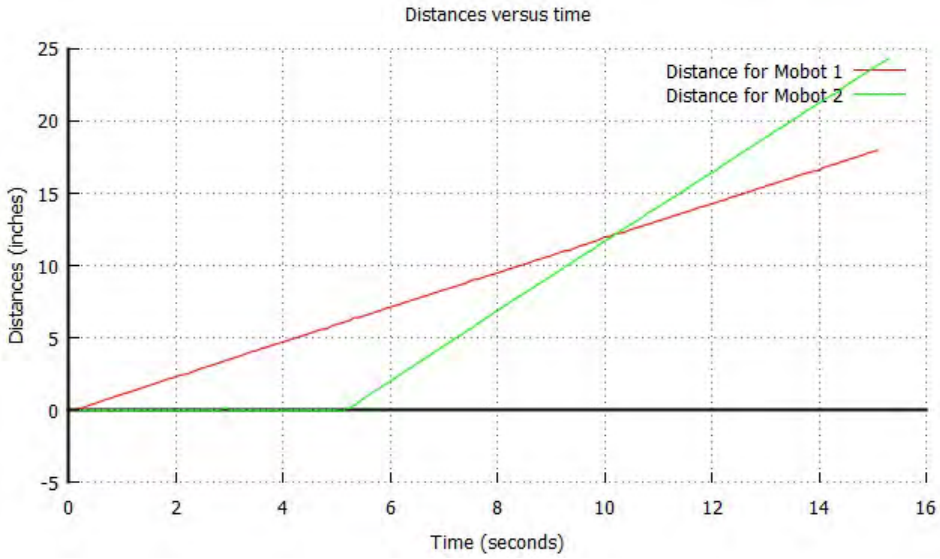
2. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels for the first Linkbot-I are 1.75 inches. The radii of the wheels for the second Linkbot-I are 2 inches. Write a program `drivetimenb3.ch` for two Linkbot-Is racing in the following manner. Two Linkbots will drive from a starting line at the same time at the same speed of 1.5 inches per second for 10 seconds.

3. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. Write a program `recorddistancestime2.ch` to control two Linkbot-Is with the following motions. The first robot drives at the speed of 1.2 inches per second. Then 5 seconds later, the second robot drives at the speed of 2.4 inches per second. Then 5 seconds later, both robots stop. Record the distances and time during the motion of both robots. Plot the distances versus time as shown in the figure below. What are the equations for the linear relations shown in the figure? Verify that the point $(t, d) = (10, 12)$ is the solution for the two equations of the linear relations.

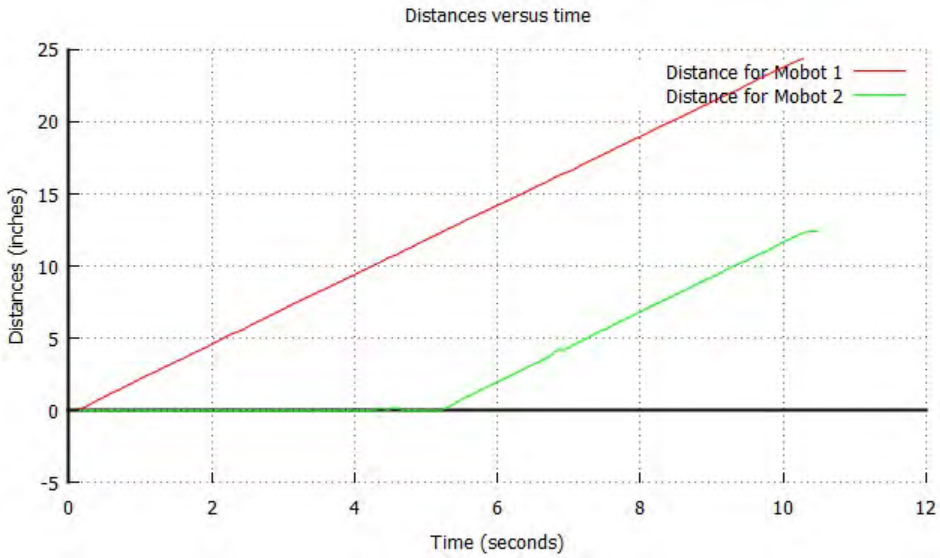


4. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. Write a program `recorddistancestime3.ch` to control two robots with the following motions. The first robot drives at the speed of 1.2 inches per second for 15 seconds. The 5 seconds after the first

robot drives, the second robot drives at the speed of 2.4 inches per second for 10 seconds. Both robots stop at the same time. Record the distances and time during the motion of both robots. Plot the distances versus time as shown in the figure below.



5. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. Write a program `recordparalleltime2.ch` to control two robots with the following motions. Both robots drive at the same speed of 2.4 inches per second. The first robot drives first. Then 5 seconds later, the second robot drives. Then 5 seconds later, both robots stop. Record the distances and time during the motion of both robots. Plot the distances versus time as shown in the figure below.



13.9 Move Multiple Linkbots Forever

In some situations, moving a robot for a specified time may not be desirable. In addition to moving joints with a specified time, a robot can be moved indefinitely with the member function **driveForeverNB()**. The motion problem for two two-wheel drive vehicles, as shown in Figure 13.4 on page 245 and described in section 13.8, can be solved using this function.

```
/* File: driveforevernb.ch
   Drive two two-wheeled robots with different speeds continuously
   with a specified time using driveForeverNB(). */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double radius1=1.75; // the radius of the two wheels of robot1 in inches
double radius2=1.75; // the radius of the two wheels of robot2 in inches
double speed1=1.5, speed2=3; // speed of robots in inches per second
double time1=16;             // motion time in seconds for robot1
double delaytime=8;          // delay time for robot2

/* set the speed for robot1 */
robot1.setSpeed(speed1, radius1);
/* set the speed for robot2 */
robot2.setSpeed(speed2, radius2);

/* robot1 drives first, 'delaytime' seconds later, robot2 drives.
   Both robots drive for 'time-delaytime' seconds at the same time.
   Then, both robots stop */
robot1.driveForeverNB();
robot2.delaySeconds(delaytime);
robot2.driveForeverNB();
robot2.delaySeconds(time1-delaytime);
robot1.holdJoints();
robot2.holdJoints();
```

Program 13.13: Moving multiple Linkbots with specified times using **driveForeverNB()**.

In Program 13.13, the variable `time` of type **double** is declared and assigned a value of 16 seconds. Since motion states for both Linkbots will not be set for a specific time, the variable `time` will be used to calculate delay. The **CLinkbotI** member function **driveForeverNB()** has the general syntax

```
robot.driveForeverNB();
```

Unlike the function **driveTimeNB()**, the function **driveForeverNB()** will move joints of the robot indefinitely. In other words, joints 1 and 3 of the Linkbot-I will move until another function call changes the joint states. **driveForeverNB()** is a non-blocking function, which means that the next line of code will start to execute immediately afterward. The line

```
robot1.driveForeverNB();
```

starts moving `robot1` forward at a speed of 1.5 inches per second. The resulting movement is the same as in Program 13.11, except that there is no time limit. Whereas in Program 13.11 `robot1` was commanded to move forward for only 16 seconds, in Program 13.13 `robot1` will continue moving forward until the movement state is changed. The next line of code calls the member function **delaySeconds()** to delay execution of the next line of code for 8 seconds, as in Program 13.11. The following line

```
robot2.driveForeverNB();
```

starts moving `robot2` forward at a speed of 3 inches per second. As it is for `robot1`, the motion time for `robot2` is also unlimited. At this point both `robot1` and `robot2` will be moving forward simultaneously. Since the movement state of `robot2` has been set using a non-blocking function, the next program statement

```
robot2.delaySeconds(time-delaytime);
```

is called to delay the execution of the next line of code for `time-delaytime` seconds, which equals 8 seconds.

You can hold all joint angles of a Linkbot by calling the member function **holdJoints()**. The general syntax of this function is

```
robot.holdJoints();
```

The lines

```
robot1.holdJoints();
robot2.holdJoints();
```

hold joints 1 and 3 of `robot1` and `robot2`. This will stop the motion of both `robot1` and `robot2`. Both Linkbots will hold their current positions as the program finishes execution, until they run out of power or are turned off manually.

As in Program 13.11, the speed of the second Linkbot-I is twice as fast as the first one. Both Linkbot-Is shall stop at the same finishing line at the same time, and the distance traveled by two Linkbot-Is is the same.

There is also a non-blocking function to move a single joint for an indefinite period of time. The **CLinkbotI** member function **moveJointForeverNB()** has the following general syntax

```
robot.moveJointForeverNB(id);
```

The argument `id` indicates the joint to move. Since this function is non-blocking, the next line of code will execute simultaneously.

E Do Exercises 1 and 2 on page 266.

Similar to Programs 13.9 and 13.10, and Programs 13.11 and 13.12, Program 13.13 can be modified to add the data acquisition and plotting code to display the distances versus time. Program 13.14 is very similar to Program 13.12, except that the movement states for both Linkbot-Is are set for a specified time using the member function **driveForeverNB**(). The lines of code moving robots are the same as those from Program 13.13. The plot generated by Program 13.14 looks identical to that generated by Programs 13.10 and 13.12, as shown in Figure 13.7. The distances of two Linkbots and time are also related by the equations (13.1) and (13.2) as described on page 252.

```
/* File: recorddistances.ch
   Record time and distances, plot the acquired data for two robots.
   The equations of motions are
          d = 1.5t
          d = 3(t-8)
*/
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot1, robot2;
double speed1=1.5, speed2=3; // speeds of robots in inches per second
double radius1=1.75, radius2=1.75; // the radii of two wheels of robot1 and 2
double time1=16;             // motion time in seconds for robot1
double delaytime=8;          // delay time for robot2
double timeInterval = 0.1;   // time interval in 0.1 second
int numDataPoints1, numDataPoints2; // number of data points recorded
robotRecordData_t timedata1, distances1; // recorded time and distances for robot1
robotRecordData_t timedata2, distances2; // recorded time and distances for robot2
CPlot plot;                  // plotting class

/* move to the zero position at the same time. */
robot1.resetToZeroNB();  robot2.resetToZeroNB();
robot1.moveWait();       robot2.moveWait();

/* set the speeds for robot1 and robot2 */
robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

/* disable record data shift and begin recording time and distance */
robot1.recordNoDataShift();
robot2.recordNoDataShift();
robot1.recordDistanceBegin(timedata1, distances1, radius1, timeInterval);
robot2.recordDistanceBegin(timedata2, distances2, radius2, timeInterval);

/* robot1 drives first, 'delaytime' seconds later, robot2 drives.
   Both robots drive for 'time-delaytime' seconds at the same time.
   Then, both robots stop */
robot1.driveForeverNB();
robot2.delaySeconds(delaytime);
robot2.driveForeverNB();
robot2.delaySeconds(time1-delaytime);
robot1.holdJoints();
robot2.holdJoints();

/* end recording time and distance */
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);

/* plot the data */
plot.title("Distances versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distances (inches)");
plot.data2DCurve(timedata1, distances1, numDataPoints1);
plot.legend("Distance for robot 1");
plot.data2DCurve(timedata2, distances2, numDataPoints2);
plot.legend("Distance for robot 2");
plot.plotting();
```

Program 13.14: Recording the distances of two Linkbot-Is and plotting the distances versus time.

 Do Exercises 3 and 4 on page 266.

Similar to the program `recordparalleltime.ch` which generates Figure 13.8, the program `recordparallel.ch` generates the same output using the member function **driveForeverNB**(). This program is distributed along with other sample programs.

 Do Exercise 5 on page 266.

13.9.1 A Baton Passing Robot Relay Race

An application of using multiple robots can be illustrated by a robot relay race. In this example, accelerometer data as described in section 12.9 is used to detect the motion of a robot.

An Application Example:
Write a program `batonpass.ch` to simulate the passing of a baton in a relay race using two two-wheel robots configured and oriented as shown in Figure 13.9. Both robots begin with red LEDs. When the first robot begins to move, its LED changes to green, while the second robot remains stationary with a red LED. The first robot should be able to travel any distance to reach the second. When the first robot runs into the second one, the accelerometer of the second robot responds to the bump. The first robot then stops and changes its LED back to red as the second robot begins traveling forward with a green LED for 6 inches.

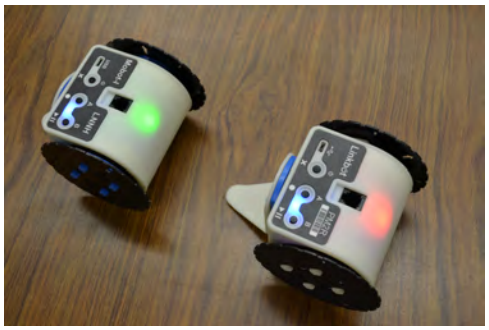


Figure 13.9: Passing a baton in a relay race with two robots.

```
/* File: batonpass.ch
   Pass a baton (green light) in a relay race with two Linkbot-Is
   using accelerometer data. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double distance=6;   // the distance of 6 inches to drive forward
double radius=1.75;  // the radius of 1.75 inches of the two wheels of the robot
double x, y1, y2, z; // accelerometer values in x, y, and z components

robot1.setLEDColor("red");
robot2.setLEDColor("red");
robot1.driveForeverNB();
robot1.setLEDColor("green");

/* while the y component of the accelerometer does not change, keep reading it. */
robot2.getAccelerometerData(x, y1, z);
y2=y1;
while(abs(y2-y1)<0.03) {
    robot2.getAccelerometerData(x, y2, z);
}
robot2.setLEDColor("green");
robot2.driveDistanceNB(distance, radius);
robot1.holdJoints();
robot1.setLEDColor("red");
robot2.moveWait();
robot2.setLEDColor("red");
```

Program 13.15: Passing a baton (green light) in a relay race with two robots using the accelerometer data.

When the first robot runs into the second one, the y component of the accelerometer of the second robot as shown in Figure 12.2 should change. We can use this information to control the motion and LED colors of both robots accordingly.

Program 13.15 use two variables `y1` and `y2` to keep the y components of the accelerometer for the second robot at different time for comparison. Program 13.15 first set the LED colors of two Linkbot-Is to red using the member function **setLEDColor**(). The statements

```
    robot1.driveForeverNB();
    robot1.setLEDColor("green");
```

then move the first robot forward and set its LED color to green. The lines

```
    robot2.getAccelerometerData(x, y1, z);
    y2=y1;
```

obtain the accelerometer data for the second robot and make sure the variables `y1` and `y2` contain the same value for the y component of the accelerometer before the comparison is made. The **while** loop

```
    while(abs(y2-y1)<0.03) {
        robot2.getAccelerometerData(x, y2, z);
    }
```

monitors the change of the y component of the accelerometer of the second robot. The mathematics function **abs**() returns the absolute value of its argument. For example, **abs**(1.5) returns 1.5 and **abs**(-1.5) also returns 1.5. Due to the noise, the difference expression $y2 - y1$ could be negative or positive at each reading of the accelerometer data. The condition expression $abs(y2 - y1) < 0.03$ uses the function **abs**() to obtain the absolute value of the difference between the values of the two variables `y1` and `y2`. When the second robot is not bumped, the absolute value for the difference for each reading of the accelerometer should be

less than 0.03. In this case, the **while** loop keeps reading the accelerometer and pass the y component to the variable `y2` for comparison. When the second robot is bumped by the first robot, the result `abs(y2-y1)` should be greater than 0.03. The control of the flow of the program will move to the following statements

```
robot2.setLEDColor("green");
robot2.driveDistanceNB(distance, radius);
robot1.holdJoints();
robot1.setLEDColor("red");
```

The above statements will set the LED color of the second robot to green, move the second robot for the specified `distance` and at the same time hold the first robot, and change the LED color of the first robot to red. The next statements

```
robot2.moveWait();
robot2.setLEDColor("red");
```

will wait for the second robot to finish its motion before its color is changed to red and the program exits.

E Do Exercises 6, and 7 on page 266.

13.9.2 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.driveForeverNB();
   ```

   to move joints of a Linkbot, for an unlimited amount of time. The following lines of code begin execution immediately before the statement containing **driveForeverNB()** has finished executing.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveJointForeverNB(id);
   ```

   move a joint of a Linkbot, for an unlimited amount of time. The following lines of code begin execution immediately before the statement containing **moveJointForeverNB()** has finished executing.

3. Call the **CLinkbotI** member function

   ```
   robot.holdJoints();
   ```

   to relax all joints of a Linkbot.

4. Call the function

   ```
   abs(x)
   ```

   to get the absolute value of its argument `x`.

13.9.3 Terminology

robot.driveForeverNB(), **robot.moveJointForeverNB()**, absolute value, **abs()**.

13.9.4 Exercises

1. Write a program `driveforevernb2.ch` to control two Linkbot-Is with the following motions. The first Linkbot drives joints 1 and 3 continuously until the second Linkbot-I finishes rotating forward by 360 degrees.

2. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. (a) Write a program `driveforevernb3.ch` to control two Linkbot-Is with the following motions. The first Linkbot drives at the speed of 1.2 inches per second. 5 seconds later, the second Linkbot drives at the speed of 2.4 inches per second. 5 seconds later, both Linkbots stop.

3. Modify the program `recorddistancestime2.ch` developed in Exercise 3 on page 258 as the program `recorddistances2.ch` using the member function **driveTimeNB()**.

4. Modify the program `recorddistancestime3.ch` developed in Exercise 4 on page 258 as the program `recorddistances3.ch` using the member functions **driveTimeNB()** and **driveForeverNB()**.

5. Modify the program `recordparalleltime2.ch` developed in Exercise 5 on page 259 as the program `recordparallel2.ch` using the member functions **driveTimeNB()** and **driveForeverNB()**.

6. Modify Program 13.15 as the program `batonpass2.ch` to simulate the passing of a baton in a relay race using two robots as shown in Figure 13.9. Change the **while** in Program 13.15 to the code below

```
printf("y1  = %lf\n", y1);
while(abs(y2-y1)<0.03) {
    robot2.getAccelerometerData(x, y2, z);
    printf("y2 inside the while loop  = %lf\n", y2);
}
printf("y2 after the while loop  = %lf\n", y2);
```

   to watch the y component of the accelerometer for the second robot before and when it is bumped by the first robot. Also modify the program so that when the second robot is bumped by the first robot, the buzzer will be set on with the frequency of 450 till the second robot drives for 8 inches to reach its finish line.

7. Write a program `batonpass3.ch` to simulate the passing of a baton in a relay race using three robots as shown in Figure 13.10. All robots begin with red LEDs. When the first robot begins to move, its LED changes to green, while the second and third robots remain stationary with red LEDs. The first and second robots should be able to travel any distance to reach the next. When the first robot runs into the second one, the the accelerometer of the second robot responds to the bump. The first robot then stops and change s its LED back to red as the second robot begins travelling forward with a green LED. The second robot travels to run into the third robot, passes the baton (the second robot LED turns red and the third robot LED turns green), and the third robot carries the baton for 6 inches to the finish and then changes its LED back to red. You may view the video `batonpass.mp4` distributed along with other C-STEM videos.

Figure 13.10: Passing a baton in a relay race with three robots.

13.10 Plot Recorded Distances and Time for Multiple Linkbots with an Offset for the Distance

In Section 8.3.3 we learned how to record and plot distance versus time for a single Linkbot-I with a specified speed and distance with an offset. In this section we will learn how to plot the distances versus time for two Linkbot-Is with an offset for the distance of a robot through an application example.

Problem Statement:
Two Linkbot-Is are configured as two-wheel robots, as shown in Figure 13.4 on page 245. The radii of the wheels are 1.75 inches. The first robot is placed in an X-Y coordinate system at the coordinate (0, 6). The second robot is placed at (6, 0). Write a program `recorddistancesoffset.ch` to move the first robot at the speed of 1.5 inches per second for 12 seconds. After 4 seconds delay, the second robot moves at the speed of 3 inches per second in the same direction. Both robots move together for 8 seconds. Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances for both robots versus time.

We can solve this problem using Ch Linkbot Controller (CLC) with the setup shown in Figure 13.11 to solve this problem conveniently.

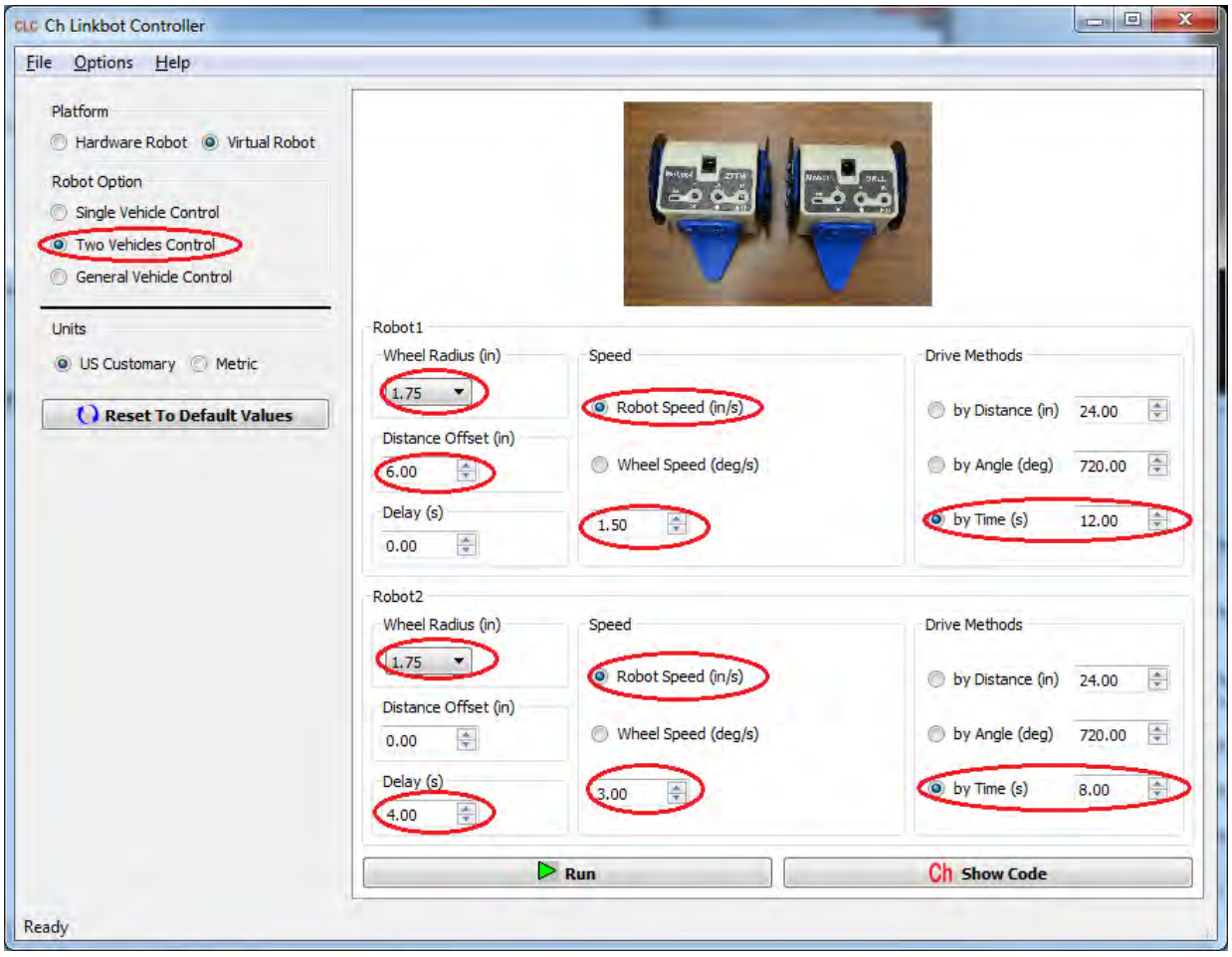


Figure 13.11: The setup of Ch Linkbot Controller for solving the problem.

```
/* File: recorddistancesoffset.ch
   Record time and distances using driveTime(), with an offset distance by
       robot1.recordDistanceOffset(offset);
   Delay 4 seconds for robot2. The equations of motions are
          d = 6 + 1.5t
          d = 3(t-4)
   Plot the acquired data for two robots */
#include <linkbot.h>
#include <chplot.h>
CLinkbotI robot1, robot2;
double speed1=1.5, speed2=3; // speeds of robots in inches per second
double radius1=1.75, radius2=1.75; // the radii of two wheels of robot1 and robot2
double offset=6;             // the offset for the distance of robot1
double time1=12, time2=8;    // motion time in seconds for robot1 and robot2
double delaytime=4;          // delay time for robot2
double timeInterval = 0.1;   // time interval in 0.1 second
int numDataPoints1, numDataPoints2; // number of data points recorded
robotRecordData_t timedata1, distances1; // recorded time and distances for robot1
robotRecordData_t timedata2, distances2; // recorded time and distances for robot2
CPlot plot;                  // plotting class

/* move to the zero position at the same time. */
robot1.resetToZeroNB();  robot2.resetToZeroNB();
robot1.moveWait();       robot2.moveWait();

/* set the speeds for robot1 and robot2 */
robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

/* set the offset of the distance for robot1 */
robot1.recordDistanceOffset(offset);

/* disable record data shift and begin recording time and distance */
robot1.recordNoDataShift();
robot2.recordNoDataShift();
robot1.recordDistanceBegin(timedata1, distances1, radius1, timeInterval);
robot2.recordDistanceBegin(timedata2, distances2, radius2, timeInterval);

/* robot1 drives first for a total of 'time1' seconds.
 'delaytime' seconds later, robot2 drives for 'time2' seconds while robot1 also drives */
robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveTime(time2);
robot1.moveWait();  // wait till robot1 moved 'time1' seconds
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);

/* plot the data */
plot.title("Distances versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distances (inches)");
plot.data2DCurve(timedata1, distances1, numDataPoints1);
plot.legend("Distance for robot 1");
plot.data2DCurve(timedata2, distances2, numDataPoints2);
plot.legend("Distance for robot 2");
plot.plotting();
```

Program 13.16: Recording the distances of two Linkbot-Is and plotting the distances versus time with an offset for the distance of robot 1.

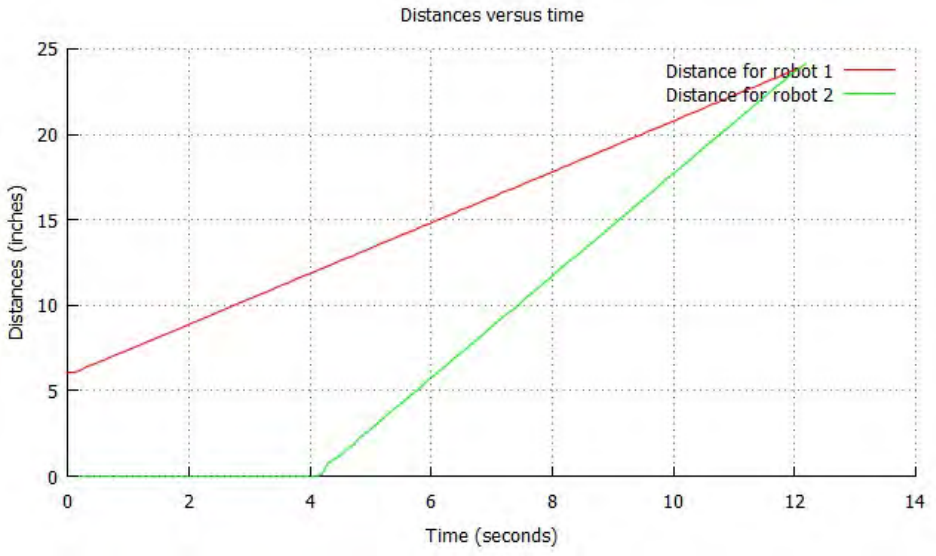


Figure 13.12: The plot for the distances of two Linkbot-Is versus time from Program 13.16.

Program 13.16 is written based on Program 13.12.We only changed the code

```
double time1=16, time2=8; // motion time in seconds for robot1 and robot2
double delaytime=8;       // delay time for robot2
```

in Program 13.12 to

```
double offset=6;          // the offset for the distance of robot1
double time1=12, time2=8; // motion time in seconds for robot1 and robot2
double delaytime=4;       // delay time for robot2
```

in Program 13.16. In addition, we added the statement

```
robot1.recordDistanceOffset(offset);
```

for the offset of the distance for robot 1. The resulting graph of distance versus time for both Linkbot-Is is shown in Figure 13.12.

As shown in Figure 13.12, the relation between the distance (d) and time (t) for the first robot can be formulated by the following linear equation.

$$d = 6 + 1.5t \tag{13.3}$$

The second robot moves at the speed of 3 inches per second and starts to move 4 seconds later. The linear relation between the distance (d) and time (t) can be formulated by the equation.

$$d = 3(t - 4) \tag{13.4}$$

The intersection point $(t, d) = (12, 24)$ of two straight lines in Figure 13.12 satisfies both equations (13.3) and (13.4). Physically, the first robot moves at the position 6 inches ahead of the second robot for 12 seconds and travels for 18 inches to reach the position 24 inches. The second robot moves at 6 inches behind the first robot after the first robot moved for 4 seconds. 8 seconds after the second robot moves, the second robot also reaches the position 24 inches. The total distance traveled by the second robot is 24 inches.

E Do Exercise 1 on page 275.

Problem Statement:
The first robot is placed in an X-Y coordinate system at the coordinate (0, 6). The second robot is placed at (6, 0), Write a program `recorddistancesoffset2.ch` to move the first robot at the speed of 1.5 inches per second for 12 seconds and the second robot at the speed of 3 inches per second for 8 seconds. Both robots start to move at the same time. Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances for both robots versus time.

The program `recordistancesoffset2.ch` distributed along with the other programs in this book is the same as Program 13.16, except that the statements

```
robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveForeverNB(time2);
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);
```

in Program 13.16 are changed to

```
robot1.driveTimeNB(time1);
robot2.driveTime(time2);
robot1.moveWait();
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);
```

in the program `recordistancesoffset2.ch` so that both robots 1 and 2 move at the same time. Since robot 1 moves for 12 seconds and robot 2 moves for 8 seconds, the statement

```
robot1.moveWait();
```

is used to wait until robot 1 has moved for 12 seconds before the statement

```
robot1.recordDistanceEnd(numDataPoints1);
```

is called to terminate the motion of robot 1. The plot from the program `recordistancesoffset2.ch` is shown in Figure 13.13.

As shown in Figure 13.13, the linear relation between the distance (d) and time (t) for the first robot is as follows.

$$d = 6 + 1.5t \tag{13.5}$$

The linear relation between the distance (d) and time (t) for the second robot can be formulated as follows.

$$d = 3t \tag{13.6}$$

The intersection point $(t, d) = (4, 12)$ of two straight lines in Figure 13.13 satisfies both equations (13.5) and (13.6). Physically, it means that 4 seconds after the motion starts for both robots 1 and 2, robot 1 has moved 8 inches and robot 2 have moved 12 inches. Because robot 1 is placed 4 inches ahead of robot2, they both reach the position 12 inches.

E Do Exercise 2 on page 276.

The program `recorddistancesoffset22.ch` removed the statement

```
robot1.moveWait();
```

It generates the plot shown in Figure 13.14. In this case, the data recording for robot 1 only lasts for 8 seconds although robot 1 moves for 12 seconds.

The program `recorddistancesoffset23.ch` uses the following statements for motion synchronization.

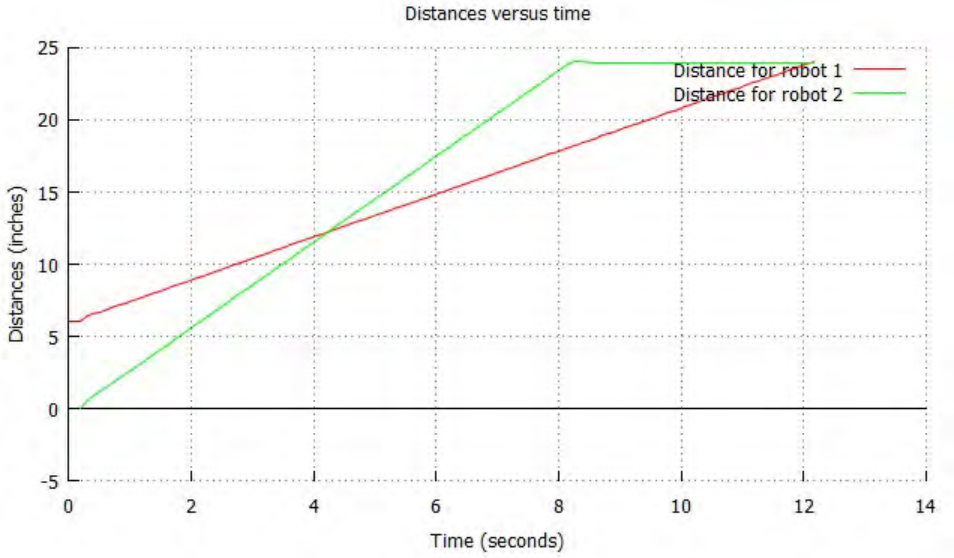


Figure 13.13: The plot for the distances of two Linkbot-Is versus time from the program `recorddistancesoffset2.ch` without the time delay.

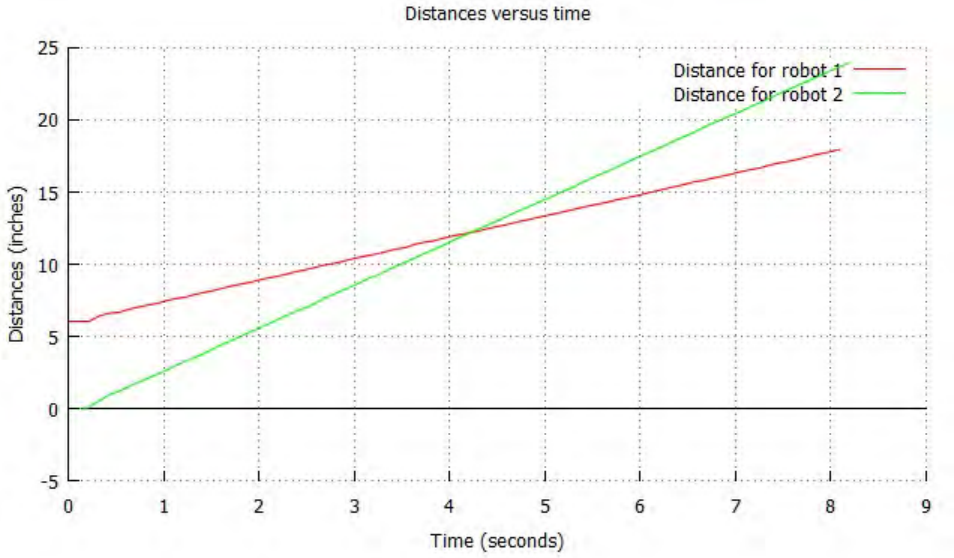


Figure 13.14: The plot for the distances of two Linkbot-Is versus time from the program `recorddistancesoffset22.ch` without the **moveWait**() statement.

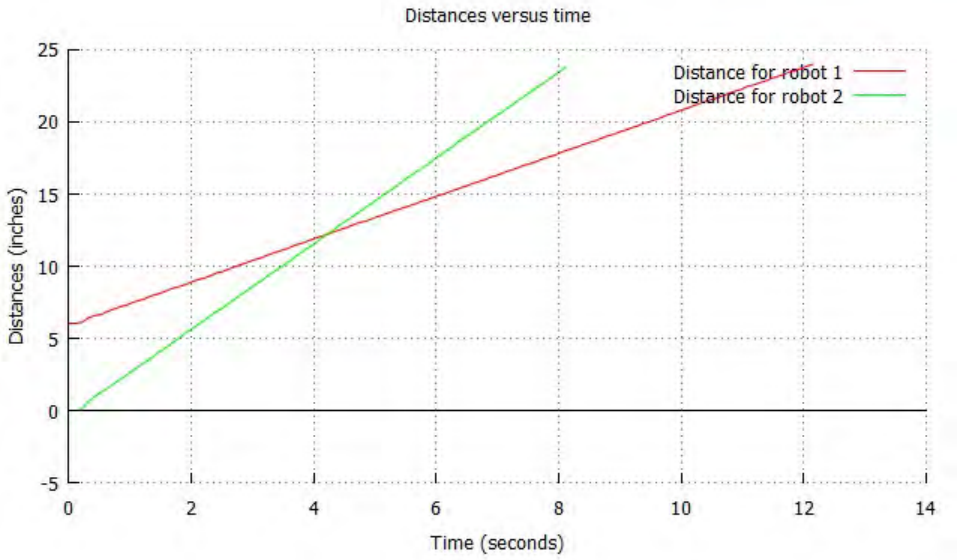


Figure 13.15: The plot for the distances of two Linkbot-Is versus time from the program `recorddistancesoffset23.ch`.

```
    robot1.driveTimeNB(time1);
    robot2.driveTime(time2);
    robot2.recordDistanceEnd(numDataPoints2);
    robot1.moveWait();
    robot1.recordDistanceEnd(numDataPoints1);
```

Once the motion for robot 2 finishes, the data recording is stopped while robot 1 is still moving. The program `recorddistancesoffset23.ch` generates the plot shown in Figure 13.15.

E Do Exercise 3 on page 276.

Problem Statement:
Robot 1 is placed 17 inches ahead of robot 2. Write a program `recordinterceptoffset.ch` to move the first Linkbot-I at the speed of 1.5 inches per second backward. Then 2 seconds later, the second Linkbot-I moves at the speed of 2 inches per second. Then 4 seconds later, both Linkbot-Is stop. Record the distances as two robots travel with a time interval of 0.1 second. Plot the distances for both Linkbot-Is versus time.

We can solve this problem conveniently using Ch Linkbot Controller (CLC) with the setup shown in Figure 13.16.

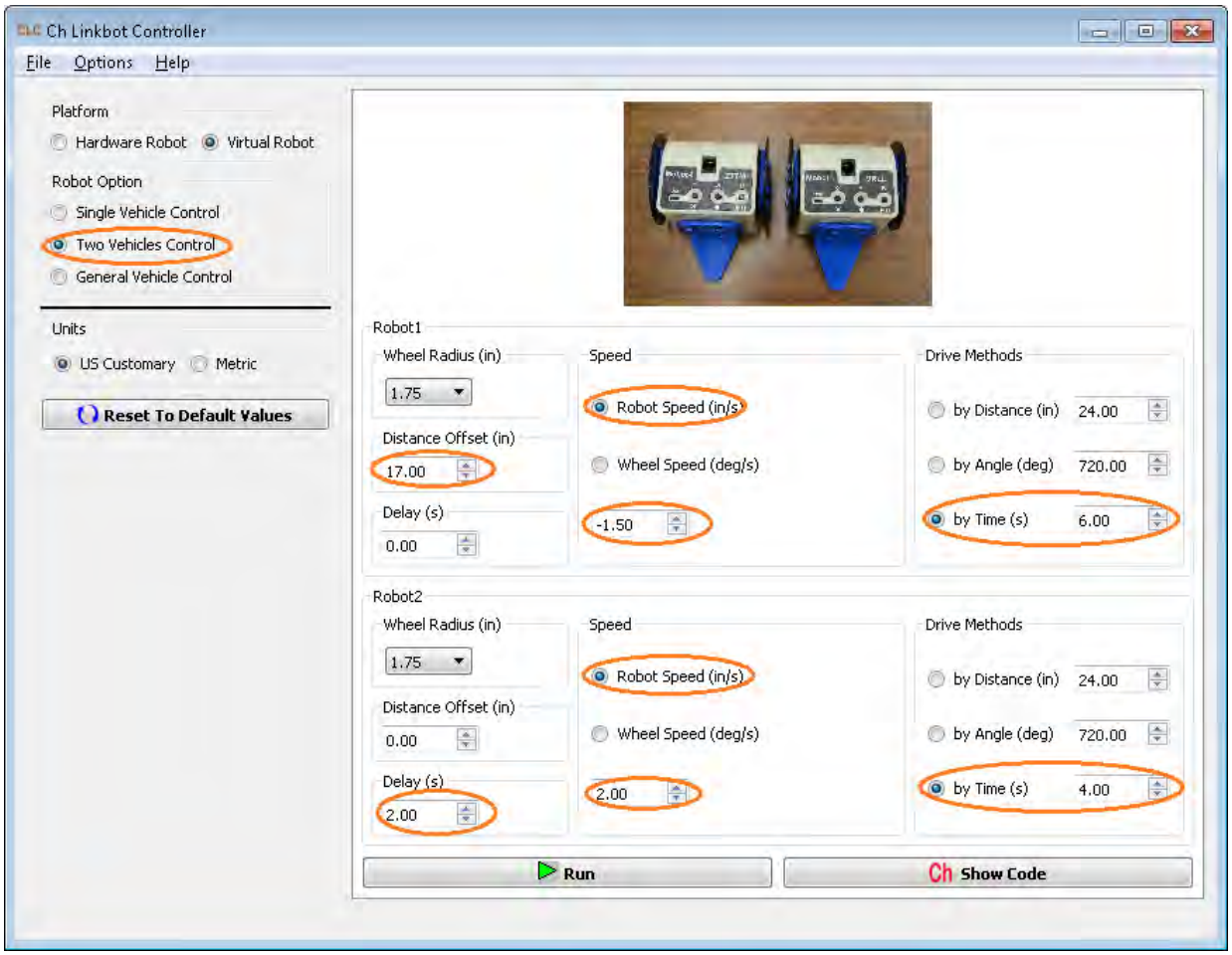


Figure 13.16: The setup of Ch Linkbot Controller for solving the problem.

You can call the member functions **driveTime()** and **driveTimeNB()** to move a Linkbot-I backward if the speed of a robot is negative. For example, the program `recordinterceptoffset.ch`, uses the statements below to move a Linkbot-I backward, with the generated plot shown in Figure 13.17.

```
robot1.setSpeed(-speed1, radius1); // negative speed for robot1
...
robot.driveTimeNB(time);
```

The program is distributed along with other sample programs.

E Do Exercise 4 on page 277.

Instead of using the member functions **driveTime()** and **driveTimeNB()** with the specified time, the distances and time for multiple Linkbots with an offset for the distance can be solved using the member functions **driveDistance()** and **driveDistanceNB()** with the specified distance. Programs `recorddistancesdistoffset.ch` and `recordinterceptdistoffset.ch` demonstrate how to use these member functions with an offset for the distance. These programs are distributed along with other sample programs.

E Do Exercises 5 and 6 on page 277

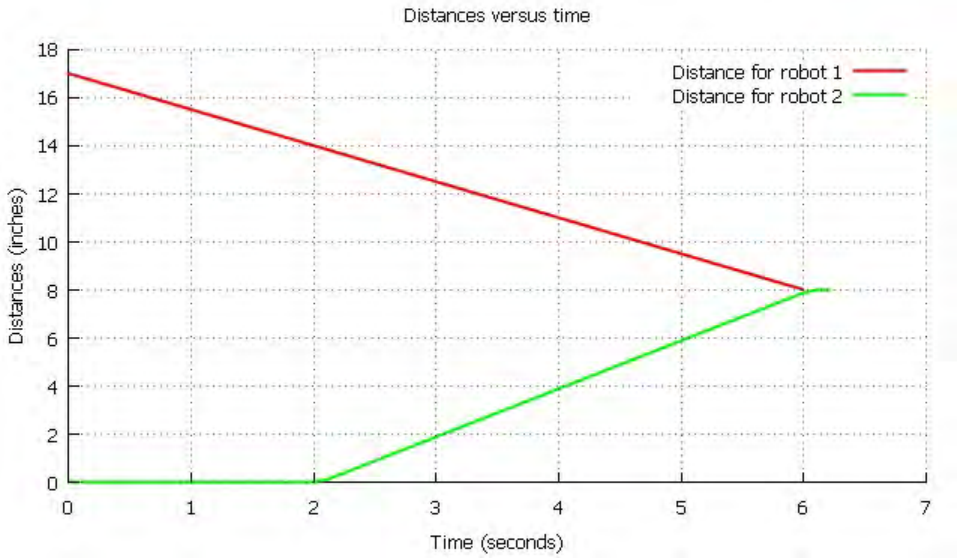


Figure 13.17: The plot for the distances of two Linkbot-Is versus time from the program `recordinterceptoffset.ch`.

13.10.1 Summary

1. Plot the distances versus time for multiple robots with an offset for the distance.

13.10.2 Terminology

13.10.3 Exercises

1. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. The first robot is placed in an X-Y coordinate system at the coordinate (0, 6). The second robot is placed at (6, 0). Write a program `recorddistancesoffset3.ch` to drive the first robot at the speed of 1.2 inches per second for 15 seconds. After 5 seconds delay, the second robot drives at the speed of 2.4 inches per second. Both robots drive together for 10 seconds. Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances versus time as shown in the figure below. What are the equations for the linear relations shown in the figure? Verify that the point $(t, d) = (15, 24)$ is the solution for the two linear equations.

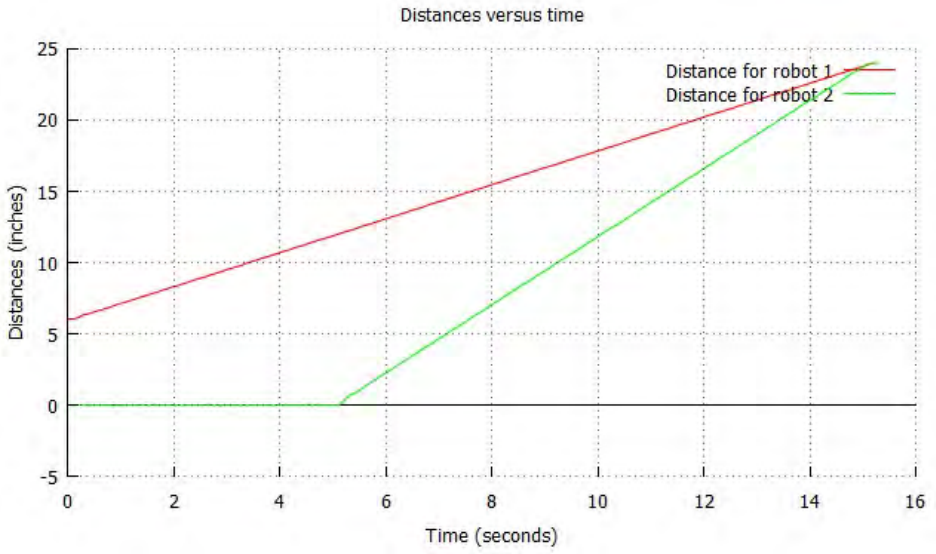
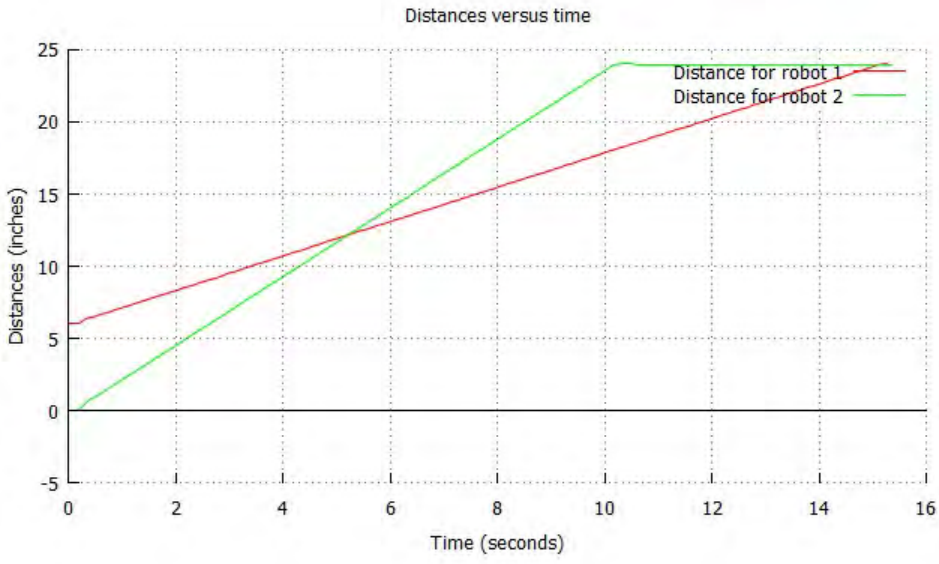
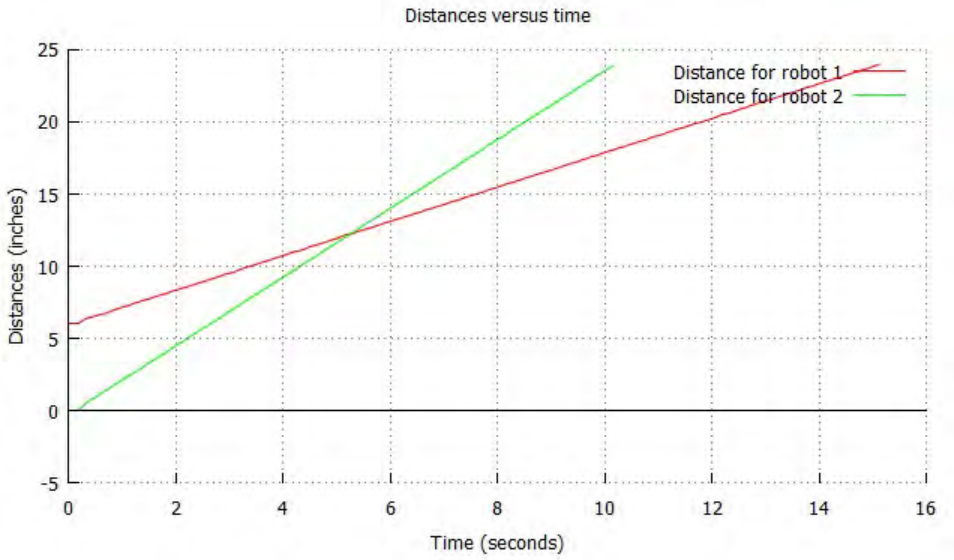


Figure 13.18: Two robots moving in the same direction.

2. Two Linkbot-Is are configured as two-wheel robots. The radii of the wheels are 1.75 inches. The first robot is placed in an X-Y coordinate system at the coordinate (0, 6). The second robot is placed at (6, 0). Write a program `recorddistancesoffset4.ch` to drive the first robot at the speed of 1.2 inches per second for 15 seconds. and the second robot at the speed of 2.4 inches per second for 10 seconds. Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances versus time as shown in the figure below. What are the equations for the linear relations shown in the figure? Verify that the point $(t, d) = (5, 12)$ is the solution for the two linear equations.



3. Modify the program `recorddistancesoffset4.ch` developed in Exercise 2 as `recorddistancesoffset43.ch` to produce a plot shown below.



4. Two Linkbots are configured as two-wheel robots. The radii of the wheels are 1.75 inches. The first robot is placed in an X-Y coordinate system at the coordinate (0, 21). The second robot is placed at (6, 0). Write a program `recordinterceptoffset2.ch` to drive the first Linkbot backward using the member function **driveTimeNB**() at the speed of 1.5 inches per second for 6 seconds. Then 2 seconds after the first Linkbots drives, the second Linkbot drives forward at the speed of 3 inches per second for 4 seconds using the member function **driveTime**(). Record the distances as the robots travel with a time interval of 0.1 second. Plot the distances for both Linkbots versus time similar to the one shown in the figure below. What are the equations for the linear relations shown in the figure? Verify that the point $(t, d) = (6, 12)$ is the solution for two linear equations.

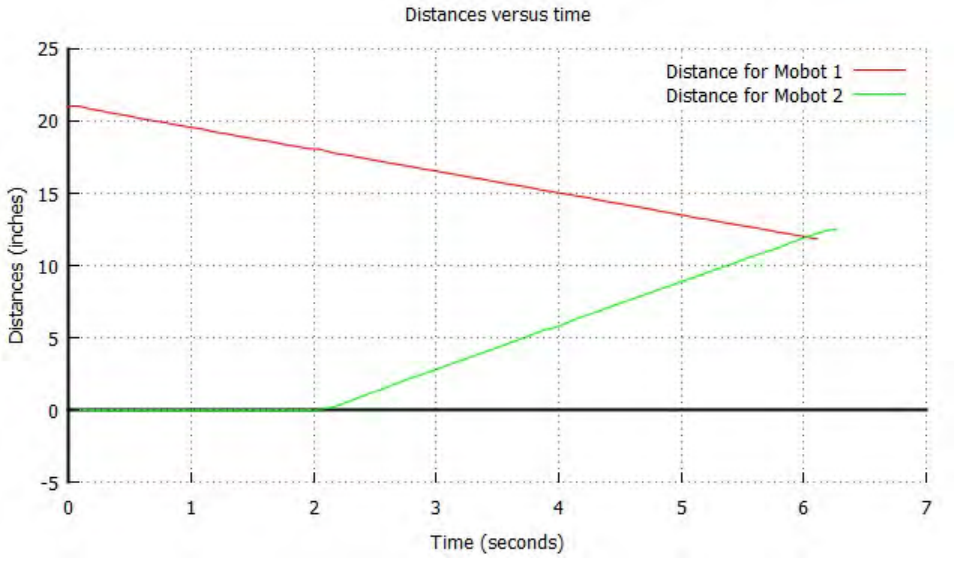


Figure 13.19: Two robots moving in opposite directions.

5. Write a program `recorddistancesdistoffset2.ch` using the member functions **driveDis-**

tanceNB() and **driveDistance**() to reproduce Figure 13.18 shown in Exercise 1.

6. Write a program `recordinterceptdistoffset2.ch` using the member functions **driveDistanceNB**() and **driveDistance**() to reproduce Figure 13.19 shown in Exercise 4.

13.11 ‡ Copy Motions of a Controller Linkbot to a Controlled Linkbot

13.11.1 Repeat Actions Using a while Loop

Another way to control multiple Linkbots with a single program is by making one Linkbot copy the actions of another Linkbot which is being controlled manually. In this section we will learn how to accomplish this with a **while** loop to help control the flow of the program.

```
/* File: copycat.ch
   Copy the motion of the 1st robot to the 2nd robot by moving joints of
   the 1st robot manually */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double angle1, angle3;

/* move to the zero position at the same time. */
robot1.resetToZeroNB(); robot2.resetToZeroNB();
robot1.moveWait(); robot2.moveWait();

/* Set colors for robots. Use the green robot to control the red one */
robot1.setLEDColor("green");
robot2.setLEDColor("red");

/* relax all joints of robot1 */
robot1.relaxJoints();

printf("You can now move joints of robot1 manually to control robot2\n");
while(1) {
    /* get the 1st robot joint angles */
    robot1.getJointAngles(angle1, NaN, angle3);
    /* drive the 2nd robot to the same position of the 1st robot */
    robot2.moveToByTrackPosNB(angle1, NaN, angle3);
}
```

Program 13.17: Copying motions of a controller Linkbot to a controlled Linkbot using the function **moveToByTrackPosNB**().

Program 13.17 copies the motions of one Linkbot-I to another Linkbot-I. In order to distinguish between Linkbots, the statements

```
    robot1.setLEDColor("green");
    robot2.setLEDColor("red");
```

sets the LED color of `robot1` to green and the LED color of `robot2` to red. The Linkbot-I with the green LED will control the Linkbot-I with the red LED. The next line

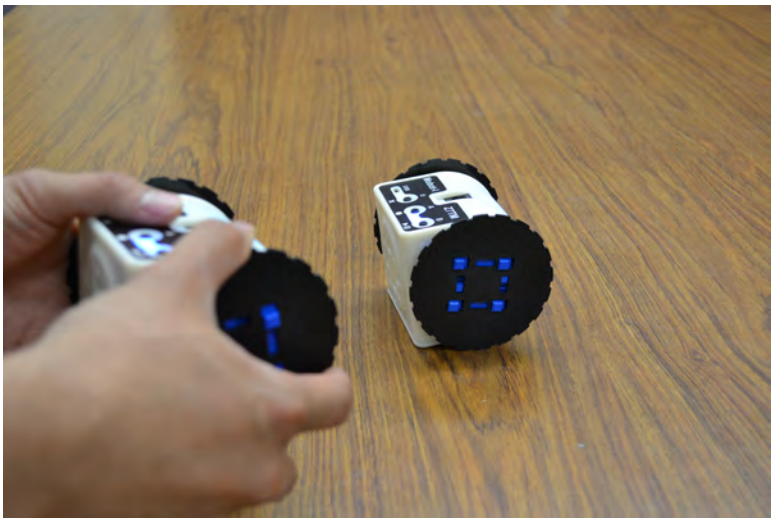


Figure 13.20: Copying the motion from one Linkbot to the other Linkbot.

```
robot1.relaxJoints();
```

relaxes joints 1 and 3 of `robot1`. This allows the user to turn each joint freely. Finally, the lines

```
while(1) {
    /* get the 1st robot joint angles */
    robot1.getJointAngles(angle1, NaN, angle3);
    /* drive the 2nd robot to the same position of the 1st robot */
    robot2.moveToByTrackPosNB(angle1, NaN, angle3);
}
```

copy the joint angles of the manually controlled Linkbot-I to the other Linkbot-I. The commands to copy motions are repeated with the help of a **while** loop. Recall from Section 12.3.2 that a **while** loop repeats a sequence of statements multiple times until the loop condition expression is no longer true. In Program 13.17 the loop condition is 1, so it is always true. The loop repeats until either the user presses the `Stop` button on the debug bar in ChIDE or until the user turns the robot off. Inside the **while** loop, the first line of code

```
robot1.getJointAngles(angle1, NaN, angle3);
```

gets the angle values for joints 1 and 3 from the manually controlled `robot1`. The second line

```
robot2.moveToByTrackPosNB(angle1, NaN, angle3);
```

moves joints 1 and 3 of `robot2` to the absolute joint angle positions copied from `robot1`. The **CLinkbotI** member function **moveToByTrackPosNB()** has the general syntax

```
robot.moveToByTrackPosNB(angle1, NaN, angle3);
```

where `angle1` and `angle3` are the joint angle values used to drive joints 1 and 3 to their absolute joint angle positions. The member function **moveToByTrackPosNB()** is used instead of **moveToNB()** in this case because the joint positions of `robot2` are being tracked instead of joint speeds.

The function **moveToByTrackPosNB()** is non-blocking, so it allows the loop to repeat immediately instead of waiting for `robot2` to complete its action. The blocking version of this function is **moveToByTrackPos()**. For the position control, instead of velocity control, use the member function **moveToByTrackPos()**.

When you run Program 13.17, you may open the Robot Control Panel of Linkbot Labs to watch the joint angles of the controlled robot.

E Do Exercise 1, 2, 3, and 4, on page 282.

We can also use sensor data from a controller Linkbot-I to determine the motion of a second Linkbot-I. Program 13.18 shows an example using accelerometer data from one Linkbot to control the motions of another Linkbot. You do not need to attach wheels to the controller Linkbot in order to run this program. The controlled Linkbot, however, will need wheels attached.

```
/* File: accelcontrol.ch
   The accelerometer value for the 1st robot in green color will determine
   the motion for the 2nd robot in red color. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double x, y, z;
int motorPower, motorOffset;

/* Set colors for robots. Use the green robot to control the red one */
robot1.setLEDColor("green");
robot2.setLEDColor("red");

/* relax all joints of robot2 */
robot2.relaxJoints();

while(1) {
    /* Get the accelerometer values */
    robot1.getAccelerometerData(x, y, z);
    /* Calculate forward-backward */
    motorPower = 200 * y;
    motorOffset = 50 * x;
    robot2.moveJointByPowerNB(JOINT1, -motorPower - motorOffset);
    robot2.moveJointByPowerNB(JOINT3, motorPower - motorOffset);
}
```

Program 13.18: Using the accelerometer of one Linkbot to determine the motion of a second Linkbot.

Program 13.18 uses the X and Y component values of the controller Linkbot's accelerometer to calculate the joint motor power of the controlled Linkbot. This program works the same way that TiltDrive mode works between two Linkbots connected using BumpConnect. The line

```
    int motorPower, motorOffset;
```

declares the variables of type **int**, `motorPower` and `motorOffset`. The value for the variable `motorPower` will be calculated using the Y component of acceleration and `motorOffset` will be calculated using the X component of acceleration from `robot1`. The difference between `motorPower` and `motorOffset` will determine the power of each joint motor of `robot2`. As in Program 13.17, the LED of `robot1` is set to green and the LED of `robot2` is set to red. Then the line

```
    robot2.relaxJoints();
```

relaxes joints 1 and 3 of `robot2`. Since the accelerometer data of `robot1` is being used as the controlling input instead of manual wheel motions, **relaxJoints()** does not need to be called on `robot1`. Then, as in Program 13.17, a **while** loop is used to set the motor power of `robot2` using the accelerometer data of `robot1`. Since the loop condition equals 1, this loop will keep repeating until the user turns off the Linkbots or presses the `Stop` button on the ChIDE debug panel. Inside the **while** loop, the line

```
    robot1.getAccelerometerData(x, y, z);
```

retrieves the X, Y, and Z components of the acceleration of `robot1`. The next lines

```
motorPower = 200 * y;
motorOffset = 50 * x;
```

use `y` from the accelerometer data to calculate `motorPower` and `x` to calculate `motorOffset`. The final two lines

```
robot2.moveJointByPowerNB(JOINT1, -motorPower - motorOffset);
robot2.moveJointByPowerNB(JOINT3, motorPower - motorOffset);
```

move joints by setting the motor power of joints 1 and 3 of `robot2` using the calculated difference between `motorPower` and `motorOffset`. This is done for each joint using the **CLinkbotI** member function **moveJointByPowerNB()**. The general syntax of this function is as follows

```
robot.moveJointByPowerNB(id, power);
```

The argument `id` specifies the joint to set the power. For a Linkbot-I this argument can have the values **JOINT1** or **JOINT3**. The argument `power` indicates the power at which to set the specified joint. This argument can have any integer value from -100 to 100.

To give an example of how the motion of `robot1` affects the motion of `robot2`, tilt `robot1` so that joint 1 is facing the floor. In this position, the X component of acceleration will be close to 1, and `motorOffset` will equal 50. The Y component of acceleration will be close to zero, so `motorPower` will equal zero. According to Program 13.18 the power of both joints 1 and 3 of `robot2` will be set to -50. As a result `robot2` will left turn. As `robot1` is gradually tilted back into a position where both joints 1 and 3 are facing outward, `robot2` will turn left more slowly until it stops. At this point the X component of acceleration will be close to zero. In a similar manner, if `robot1` is tilted so that joint 3 is facing the floor, `robot2` will turn right. Tilting `robot1` so that the curved part of the Linkbot is facing up makes the Y component of acceleration almost equal to 1. This will make `motorPower` equal 200. The X component will be close to zero in this position, so `motorOffset` will equal zero. Since the argument `power` can be no more than 100 and no less than -100, then the power of joint 1 of `robot2` will be rounded up to -100 and the power of joint 3 will rounded down to 100. As a result `robot2` will move forward quickly. Tilting `robot1` forward gradually until the curved part of this Linkbot is facing forward will make `robot2` move forward more slowly until it stops. Similarly, tilting `robot1` so that the curved part is facing the floor will make `robot2` move backward quickly.

When you run Program 13.18, you may open the Robot Control Panel of Linkbot Labs to watch the joint angles of the controlled robot or the Sensors Panel to view the x, y, and z components of the accelerometer of the controlling robot.


Do Exercises 5, 6, and 7 on page 5.

13.11.2 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.relaxJoints();
   ```

   to relax all joints of a Linkbot.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.moveToByTrackPosNB(angle1, NaN, angle3);
   ```

   to rotate the joint positions of a Linkbot-I to their absolute joint angle positions. The blocking version of this function is **moveToByTrackPos()**.

3. Call the **CLinkbotI** member function

```
robot.moveJointByPowerNB(id, power);
```

to move a joint by setting the power for the joint motor of a Linkbot-I.

13.11.3 Terminology

robot.setLEDColor(), **robot.relaxJoints()**, **robot.moveToByTrackPos()**, **robot.moveToByTrackPosNB()**, **robot.moveJointByPowerNB()**.

13.11.4 Exercises

1. What is the difference between **moveToByTrackPos**() and **moveTo**()?
2. Run the program `copycat.ch` in Program 13.17 to move the controlled Linkbot. Open the Robot Control Panel of Linkbot Labs to watch the joint angles of the controlled robot.
3. Two groups of students, each run the program `copycat.ch` in Program 13.17 in their own computers collaboratively using two controlled Linkbots controlled by two controller Linkbots to accomplish a common task, for example, race to see which controlled Linkbot crosses the finish line first or crosses an obstacle course first.
4. Write a program `copycat2.ch`, based on the program `copycat.ch` in Program 13.17, to copy the motion of the controller Linkbot-I to three controlled Linkbot-Is.
5. Run the program `accelcontrol.ch` in Program 13.18 to move the controlled Linkbot. Open the Robot Control Panel of Linkbot Labs to watch the joint angles of the controlled robot, then open the Sensors Panel to view the x, y, and z components of the accelerometer of the controlling robot.
6. Write a program `accelcolor.ch` to use the accelerometer data of one Linkbot-I to set the LED color of another Linkbot-I. First, sound the buzzer of the first Linkbot at 450 Hz and set its LED to green. Then use a **while** loop to do the following multiple times in succession: Get the X, Y, and Z components of the accelerometer data. Then calculate the R, G, and B values using these components. And then finally set the LED using these RGB values.
7. Write a program `accelcontrol2.ch` based on the program `accelcontrol.ch` in Program 13.18. This program will drive the Linkbot forward and backward at half the speed of the original program but turn the Linkbot left or right at least twice as fast as the original program. To do this, change the power and offset as follows:

```
motorPower = 100 * y;
motorOffset = 100 * x;
```

CHAPTER 14

Moving Multiple Robots in a Coordinate System

In Chapter 9, we learned how to drive a single virtual Linkbot-I in an x and y coordinate system. In this chapter, we will learn how to drive multiple virtual Linkbot-Is in an x and y coordinate system. For the convenience for defining the motion of robots in a coordinate system, the **CLinkbotI** member functions **drivexyNB**() **drivexyToNB**(), **drivexyToExprNB**(), and **drivexyWait**() are implemented to control multiple robots in a coordinate system.

14.1 Move Multiple Linkbot-Is in a Coordinate System

In Section 9.1, we used the **CLinkbotI** member function **drivexyTo**() to drive a single Linkbot in Robosim. To drive multiple virtual Linkbot-Is in an x and y coordinate system, the **CLinkbotI** member function **drivexyToNB**() will be used. The general syntax of **drivexyToNB**() is

```
robot.drivexyToNB(x, y, radius, trackwidth);
```

Like the member function **drivexyTo**(), the arguments `x` and `y` specify the coordinates to drive the Linkbot-I to. The arguments `radius` and `trackwidth` specify the radius of and distance between the Linkbot-I's wheels, respectively. The member function **drivexyToNB**() is the non-blocking version of the member function **drivexyTo**(). It can be used to drive multiple Linkbot-Is to different locations at the same time. This is demonstrated in Program 14.1. Before running Program 14.1, set the initial position of robot 1 to $(-6, 0)$ and the initial position of robot 2 to $(6, 0)$ in the RoboSim GUI.

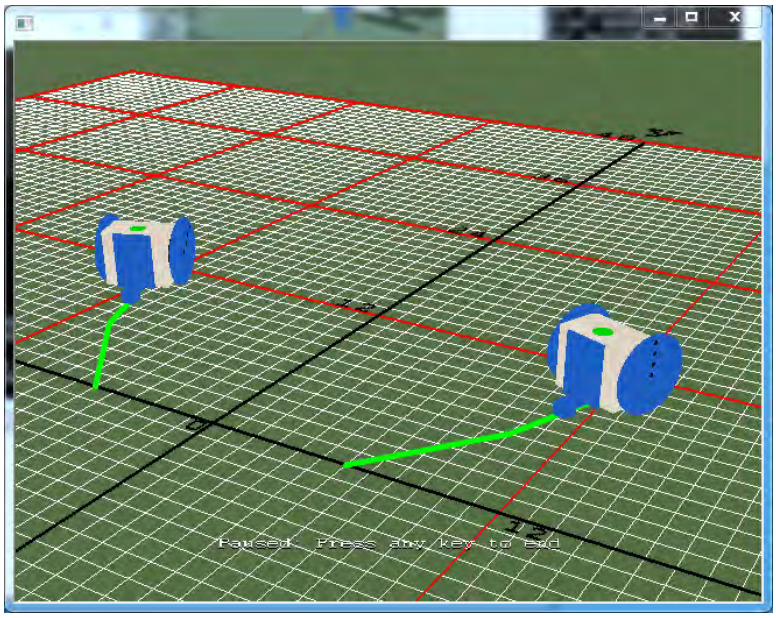


Figure 14.1: The RoboSim scene with two robot trajectories from Program 14.1.

```
/* File: drivexytonb.ch
   Note: This program uses drivexyToNB() available in RoboSim only
         to move one Linkbot-I from (-6, 0) to (-10, 5)
         and another Linkbot-I from (6, 0) to (10, 5) simultaneously.
         Then, to (-12, 10) and (12, 10) at the same time.
   Set the initial position (x, y) in RoboSim GUI to (-6, 0) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 0) for robot2. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels

// uncomment lines below for controlling hardware robots
//robot1.initPosition(-6, 0, 90);
//robot2.initPosition(6, 0, 90);

robot1.drivexyToNB(-10, 5, radius, trackwidth);
robot2.drivexyTo(10, 5, radius, trackwidth);
robot1.drivexyWait();

robot1.drivexyToNB(-12, 10, radius, trackwidth);
robot2.drivexyTo(12, 10, radius, trackwidth);
robot1.drivexyWait();
```

Program 14.1: Moving two Linkbot-Is at the same time using **drivexyTo**() and **drivexyToNB**().

Program 14.1 is similar to Program 9.1 from Section 9.1. The difference is that in Program 14.1, two Linkbot-Is are driven along two different straight lines.

The initial positions for robots are different from their default initial positions. As described in section 9.1.2, in order to move hardware robots not starting in their default positions, Program 14.1 will need to contain the following statements

```
robot1.initPosition(-6, 5.5, 90);
robot2.initPosition(6, 5.5, 90);
```

In Program 14.1, these two lines are commented out as they are not needed to control robots in RoboSim.

The statement

```
robot1.drivexyToNB(-10, 5, radius, trackwidth);
```

drives `robot1` from $(-6, 0)$ to $(-10, 5)$. Before this statement finishes execution, the next statement

```
robot2.drivexyTo(10, 5, radius, trackwidth);
```

drives `robot2` from (6, 0) to (10, 5). Since a non-blocking function is used to drive `robot1`, both Linkbots execute their motions simultaneously. The final statement

```
robot1.drivexyWait();
```

ensures that `robot1` will reach $(-10, 5)$ before the program finishes executing.

The syntax of the member function **drivexyWait**() is as follows.

```
robot.drivexyWait();
```

The function **drivexyWait()** has no argument. It pauses the program until the motion by the member function **drivexyToNB**() is completed. Since the motion for **drivexyToNB()** involves both moving distance and turning, the member function **drivexyWait()**, instead of **moveWait()**, is used for synchronization.

Similarly, the statements below

```
robot1.drivexyToNB(-12, 10, radius, trackwidth);
robot2.drivexyTo(12, 10, radius, trackwidth);
robot1.drivexyWait();
```

drive `robot1` from (-10, 5) to (-12, 10) and `robot2` from (10, 5) to (12, 10).

Figure 14.1 shows the trajectories and positions of two virtual Linkbot-Is when these two robots finish the above motions.

E Do Exercises 1 and 2(a) on page 286.

The member function **drivexyNB**() is the non-blocking version of the member function **drivexy**() to drive a robot relative its current position. The general syntax of **drivexyNB**() is

```
robot.drivexyNB(x, y, radius, trackwidth);
```

The program `drivexynb.ch`, distributed along with other sample programs, uses the member function **drivexyNB**() to drive two robots in the same manner as they are controlled by Program 14.1 using the member function **drivexyToNB**().

E Do Exercise 2(b) on page 286.

14.1.1 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.drivexyToNB(x, y, radius, trackwidth);
   ```

   on multiple Linkbot-Is to drive them simultaneously from one point to another in an x and y coordinate system. This function works in RoboSim only.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.drivexyNB(x, y, radius, trackwidth);
   ```

   on multiple Linkbot-Is to drive them simultaneously from one point to another relative to its current position in an x and y coordinate system. This function works in RoboSim only.

3. Call the **CLinkbotI** member function

```
robot.drivexyWait();
```

to pause the program until the motion by a previous member function **robot.drivexyToNB()** or **robot.drivexyNB()** is completed. This function works in RoboSim only.

14.1.2 Terminology

robot.drivexyToNB(), robot.drivexyNB(), robot.drivexyWait().

14.1.3 Exercises

1. Write a program `drivexytonb2.ch`, based on Program 14.1, to drive two virtual Linkbot-Is. Move the first Linkbot-I from (0, -6) to (5, -10) and the second Linkbot-I from (0, 6) to (5, 10).

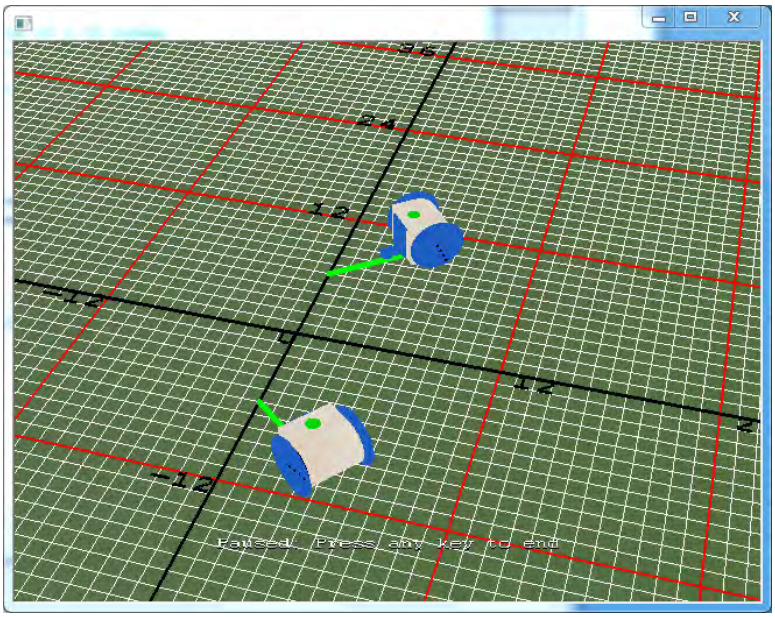


Figure 14.2: The RoboSim scene with the robot trajectories for program `drivexytonb2.ch`

2. Write a program `drivexytonb3.ch`, based on Program 14.1, to move two virtual Linkbot-Is. Move the first Linkbot-I from (0, -6) to (10, -12) and the second Linkbot-I from (0, 6) to (10, 12) simultaneously. Then drive the first robot to (24, -5) and the second one to (24, 5) at the same time.

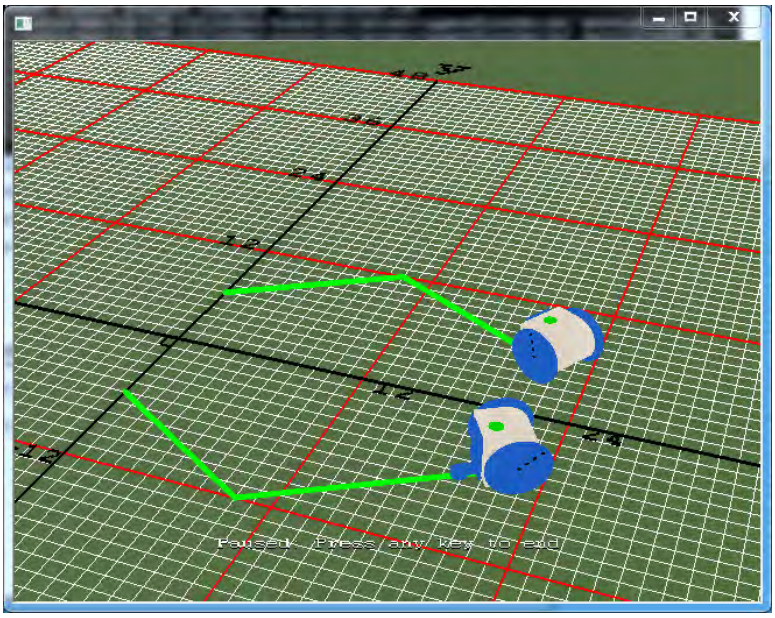


Figure 14.3: The RoboSim scene with the robot trajectories for programs `drivexytonb3.ch` and `drivexynb3.ch`

(a) Write a program `drivexytonb3.ch` using the member functions **drivexyTo**() and **drivexyToNB**().

(b) Write a program `drivexynb3.ch` using the member functions **drivexy**() and **drivexyNB**().

14.2 Trace and Record the Positions of Multiple Linkbot-Is in RoboSim

The trajectories for multiple robots can be traced simultaneously. Program 14.2 demonstrates how to trace multiple trajectories for two Linkbot-Is, as shown in Figure 14.4. Assume the `robot1` is initially placed at (0, 0) and `robot2` is at (6, 0) set in a RoboSim GUI. The following code segment in Program 14.2

```
robot1.traceOn();
robot2.traceOn();
robot1.drivexyToNB(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)
robot2.drivexyTo(12, 5, radius, trackwidth);   // line from (6, 0) to (12, 5)
robot1.drivexyWait();
```

traces the line from (0, 0) to (6, 5) for `robot1` and the line from (6, 0) to (12, 5) for `robot2` simultaneously. For the next segment of the code,

```
robot1.traceOff();
robot2.traceOff();
robot1.drivexyToNB(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)
robot2.drivexyTo(6, 10, radius, trackwidth);   // line from (12, 5) to (6, 10)
robot1.drivexyWait();
```

before `robot1` drives from (6, 5) to (0, 10) and `robot2` drives from (12, 5) to (6, 10), the trace is turned off for both robots. Finally, the segment of the code,

```
   robot1.traceOn();
   robot2.traceOn();
   robot1.drivexyToNB(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)
   robot2.drivexyTo(12, 15, radius, trackwidth);  // line from (6, 10) to (12, 15)
   robot1.drivexyWait();
```

traces the lines from (0, 10) to (6, 15) and from (6, 10) to (12, 15).

```
/* File: traceon2.ch
   Turn trace on and off with two Linkbot-Is
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 0) for robot2. */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels

robot2.setLEDColor("red"); // set the robot2 LED color to red

robot1.traceOn();
robot2.traceOn();
robot1.drivexyToNB(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)
robot2.drivexyTo(12, 5, radius, trackwidth);   // line from (6, 0) to (12, 5)
robot1.drivexyWait();

robot1.traceOff();
robot2.traceOff();
robot1.drivexyToNB(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)
robot2.drivexyTo(6, 10, radius, trackwidth);   // line from (12, 5) to (6, 10)
robot1.drivexyWait();

robot1.traceOn();
robot2.traceOn();
robot1.drivexyToNB(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)
robot2.drivexyTo(12, 15, radius, trackwidth);  // line from (6, 10) to (12, 15)
robot1.drivexyWait();
```

Program 14.2: Tracing trajectories for two Linkbot-Is.

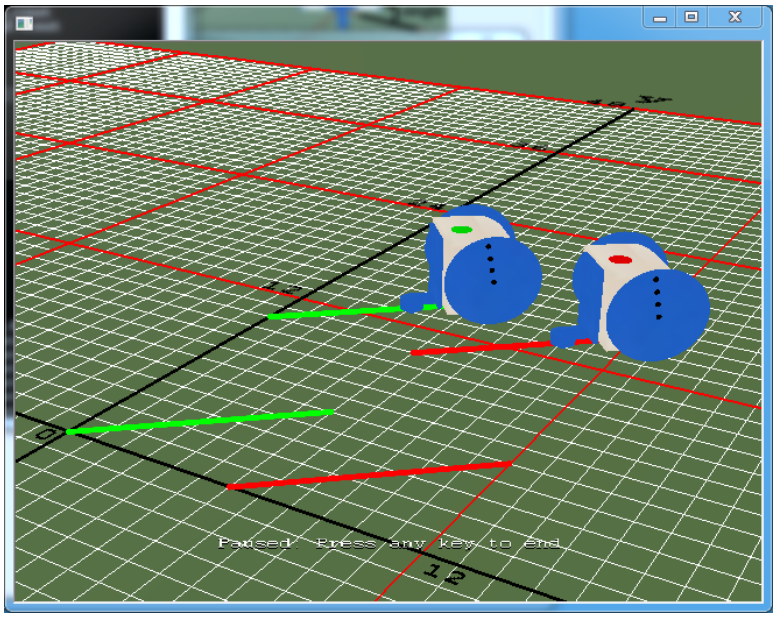


Figure 14.4: The RoboSim scene with the robot trajectories from Programs 14.2 and 14.3.

E Do Exercise 1, on 291.

The positions for the traced trajectories for multiple Linkbot-Is can be recorded simultaneously and then plotted. Program 14.3 demonstrates how to record and plot positions of the traced trajectories for two Linkbot-Is. Program 14.3 will trace the trajectories shown in Figure 14.4 and plot the acquired positions for the traced trajectories shown in Figure 14.5. The tracing and motion statements for two Linkbots-Is in Program 14.3 are the same as those in Program 14.2.

The statements

```
    /* begin recording x and y positions */
    robot1.recordxyBegin(xdata1, ydata1, timeInterval);
    robot2.recordxyBegin(xdata2, ydata2, timeInterval);
```

start recording the positions of traced trajectories for two Linkbot-Is.

After two Linkbot-Is stop moving, the statements

```
    /* end recording x and y positions */
    robot1.recordxyEnd(numDataPoints1);
    robot2.recordxyEnd(numDataPoints2);
```

stop recording the positions for the traced trajectories.

The statements below

```
    plot.scattern(xdata1, ydata1, numDataPoints1, "green");
    plot.scattern(xdata2, ydata2, numDataPoints2, "red");
```

plot two sets of the data points in scattern plot. The first data set is plotted in the green color. and the second set in the red color.

```
/* File: recordxy2.ch
   Trace and record the x and y positions for two Linkbot-Is,
   plot the acquired data.
   Set the initial position (x, y) in RoboSim GUI to (0, 0) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 0) for robot2. */
#include <linkbot.h>
#include <chplot.h>
```

```
CLinkbotI robot1, robot2;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double timeInterval = 0.1; // time interval in 0.1 second
int numDataPoints1;        // number of data points recorded for robot1
int numDataPoints2;        // number of data points recorded for robot2
robotRecordData_t xdata1, ydata1; // recorded x and y positions for robot1
robotRecordData_t xdata2, ydata2; // recorded x and y positions for robot2
CPlot plot;                // plotting class

robot2.setLEDColor("red"); // set the robot2 LED color to red

/* begin recording x and y positions */
robot1.recordxyBegin(xdata1, ydata1, timeInterval);
robot2.recordxyBegin(xdata2, ydata2, timeInterval);

robot1.traceOn();
robot2.traceOn();
robot1.drivexyToNB(6, 5, radius, trackwidth);  // line from (0, 0) to (6, 5)
robot2.drivexyTo(12, 5, radius, trackwidth);   // line from (6, 0) to (12, 5)
robot1.drivexyWait();

robot1.traceOff();
robot2.traceOff();
robot1.drivexyToNB(0, 10, radius, trackwidth); // line from (6, 5) to (0, 10)
robot2.drivexyTo(6, 10, radius, trackwidth);   // line from (12, 5) to (6, 10)
robot1.drivexyWait();

robot1.traceOn();
robot2.traceOn();
robot1.drivexyToNB(6, 15, radius, trackwidth); // line from (0, 10) to (6, 15)
robot2.drivexyTo(12, 15, radius, trackwidth);  // line from (6, 10) to (12, 15)
robot1.drivexyWait();

/* end recording x and y positions */
robot1.recordxyEnd(numDataPoints1);
robot2.recordxyEnd(numDataPoints2);

/* plot the data */
plot.title("Position");
plot.label(PLOT_AXIS_X, "X (inches)");
plot.label(PLOT_AXIS_Y, "Y (inches)");
plot.axisRange(PLOT_AXIS_X, -5, 15);
plot.axisRange(PLOT_AXIS_Y, 0, 20);
plot.scattern(xdata1, ydata1, numDataPoints1, "green");
plot.scattern(xdata2, ydata2, numDataPoints2, "red");
plot.sizeRatio(1);
plot.plotting();
```

Program 14.3: Recording and plotting the positions of the traced trajectories for two Linkbot-Is.

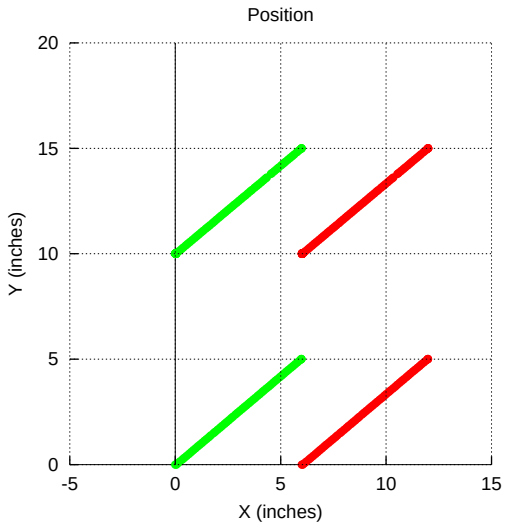


Figure 14.5: The plot for the traced trajectories for two robots from Program 14.3.

Do Exercise 2, on 291.

14.2.1 Exercises

1. Write a program `traceon4.ch` to trace two letters 'A' in RoboSim as shown in Figure 14.6. The horizontal line for the first letter 'A' can be drawn from the point $(-2.5, 5)$ to the point (2.5 5). The horizontal line for the second letter 'A' is from the point (7.5, 5) to the point (12.5, 5).

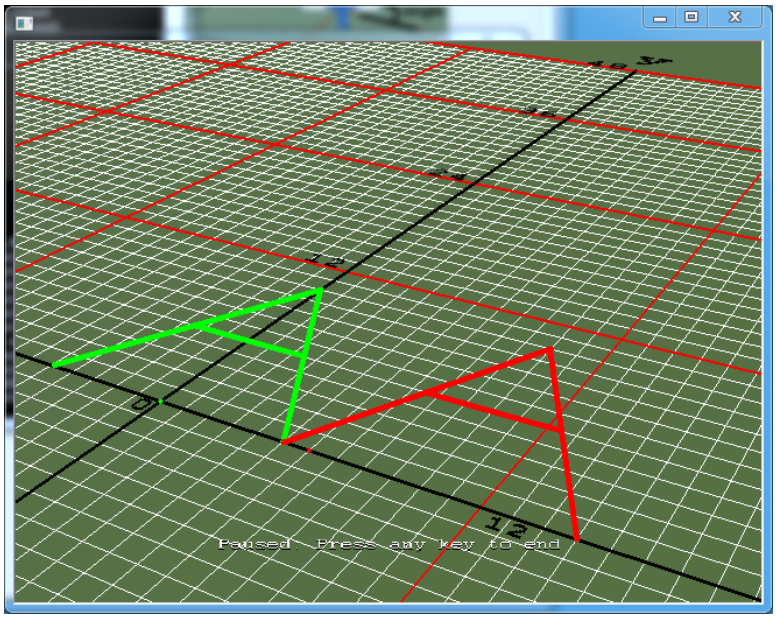


Figure 14.6: Moving two virtual Linkbot-Is to trace two letters of 'A'.

2. Based on the program `traceon4.ch` developed in Exercise 2, write a program `recordxy4.ch` to record and plot the traced trajectories for two Linkbot-Is with two letters of 'A' as shown in Figure 14.7.

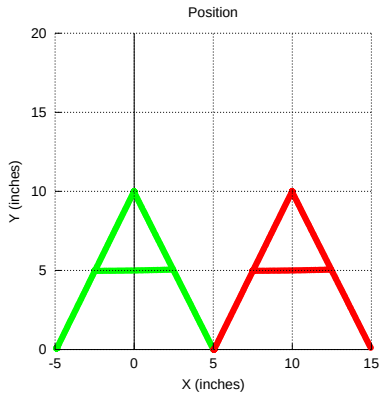


Figure 14.7: A plot generated by the program `recordxy4.ch`.

14.3 Move Multiple Linkbot-Is Along Different Trajectories

In section 9.3, we learned how to drive a single Linkbot-I along a trajectory specified by an expression using the member function **drivexyToExpr**() or by a function using the member function **drivexyToFunc**(). In this section, we will learn how to drive two Linkbot-Is along two separate trajectories simultaneously. The two sample trajectories are polynomial curves specified by the formulas

$$y_1 = 0.5(x+5)(x-5) \tag{14.1}$$
$$y_2 = 0.5(x-7)(x-17) \tag{14.2}$$

The second polynomial is a translation of the first polynomial in the x direction for 12 inches. We will use three different approaches of using expressions, functions, and a loop to drive two robots simultaneously along the above two polynomials. The first robot starts at the position $(-6, 5.5)$ and the second one begins at $(6, 5.5)$.

14.3.1 Move Multiple Linkbot-Is Along Different Trajectories Using Expressions

Similar to the method described in section 9.3.1, in this section, we will use member function **drivexyToExpr**() and non-blocking member function **drivexyToExprNB**(). Program 14.4 moves two robots along two polynomial trajectories specified in Equations (14.1) and (14.2).

As the robots do not start at their default initial positions, we need to set the initial positions of robot 1 and robot 2 to (−6, 5.5) and (6, 5.5), respectively, in the RoboSim GUI or call the member function **initPosition**() as follows

```
robot1.initPosition(-6, 5.5, 90);
robot2.initPosition(6, 5.5, 90);
```

as shown in Program 14.4.

```
/* File: polynomials.ch
   Note: This program uses drivexyToExpr() and drivexyToExprNB().
   Set the initial position (x, y) in RoboSim GUI to (-6, 5.5) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 5.5) for robot2.

   robot1 moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6.
   robot2 moves along a polynomial curve y = 0.5(x-7)(x-17) for x from 6 to 18.
   Move both robots with 30 points.
   Plot the 1st polynomial y for x from -6 to 6 with 500 points.
   Plot the 2nd polynomial y for x from 6 to 18 with 500 points.
   The range of x-axis is from -12 to 24.
   The range of y-axis is from -15 to 12.
   The tics range for x and y axes is 1. */
#include <chplot.h>
#include <linkbot.h>
CPlot plot;
CLinkbotI robot1;
CLinkbotI robot2;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels

// uncomment lines below for controlling hardware robots
//robot1.initPosition(-6, 5.5, 90);
//robot2.initPosition(6, 5.5, 90);

/* change the default green color to red for robot 1 */
robot1.setLEDColor("red");

/* move the robots along the polynomial curves */
robot1.drivexyToExprNB(-6, 6, 30, "0.5*(x+5)*(x-5)", radius, trackwidth);
robot2.drivexyToExpr(6, 18, 30, "0.5*(x-7)*(x-17)", radius, trackwidth);
robot1.drivexyWait();

/* plot the polynomial curve */
plot.title("");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 24);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_XY, 1);
plot.expr(-6, 6, 500, "0.5*(x+5)*(x-5)");
plot.legend("y = 0.5(x+5)(x-5)");
plot.expr(6, 18, 500, "0.5*(x-7)*(x-17)");
plot.legend("y = 0.5(x-7)(x-15)");
plot.plotting();
```

Program 14.4: Moving two robots to follow two polynomial curves using **drivexyToExpr**() and **drivexyToExprNB**().

In Program 14.4, the red color for the trajectory of the first robot is set by the member function call

```
robot1.setLEDColor("red");
```

by default, the color for the trajectory of the second robot is the green.

The member function **drivexyToExpr**() can be used to move a robot along a trajectory based a Ch expression. The non-blocking member function **drivexyToExprNB**() can be used to conveniently to move multiple robots along given trajectories simultaneously. Similar to the member function **drivexyToExpr**(),

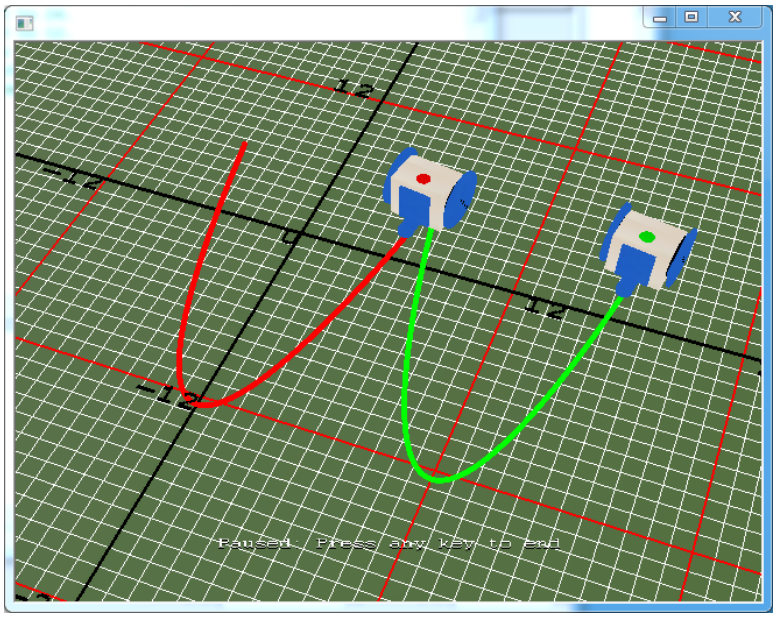


Figure 14.8: The RoboSim scene with the robot trajectories of two polynomials from Programs 14.4 and 14.7.

the syntax of the member function **drivexyToExprNB**() is as follows.

```
robot.drivexyToExprNB(x0, xf, num, expr, radius, trackwidth);
```

This non-blocking member function drives a Linkbot-I following the trajectory specified by the expression `exprr` in the range `[x0, xf]` using `num` points. The last two arguments specify the radius of the two wheels and the track width of the robot.

Program 14.4 uses the statements

```
robot1.drivexyToExprNB(-6, 6, 30, "0.5*(x+5)*(x-5)", radius, trackwidth);
robot2.drivexyToExpr(6, 18, 30, "0.5*(x-7)*(x-17)", radius, trackwidth);
robot1.drivexyWait();
```

to drive two robots simultaneously along the two polynomials $0.5*(x+5)*(x-5)$ and $0.5*(x-7)*(x-17)$. Figure 14.8 shows the parabolic trajectories of `robot1` and `robot2`.

The trajectories of the two Linkbot-Is are plotted, in a manner similar to Program 9.5. In Section 13.7, it has been demonstrated that the motion of two Linkbot-Is can be plotted on the same graph. This is also done in Program 14.4. To allow room for both parabolas, the line

```
plot.axisRange(PLOT_AXIS_X, -12, 24);
```

sets a larger range on the x-axis, from -12 to 24. The additional statements

```
plot.expr(-6, 6, 500, "0.5*(x+5)*(x-5)");
plot.legend("0.5*(x+5)*(x-5)");
plot.expr(6, 18, 500, "0.5*(x-7)*(x-17)");
plot.legend("0.5*(x-7)*(x-17)");
```

plot the expression `0.5*(x+5)*(x-5)` for `robot1` in the range `[−6, 6]` and for `robot2` in the range `[6, 18]`. Both trajectories use 500 points. Figure 14.9 shows the generated plot.

E Do Exercise 1 on page 302.

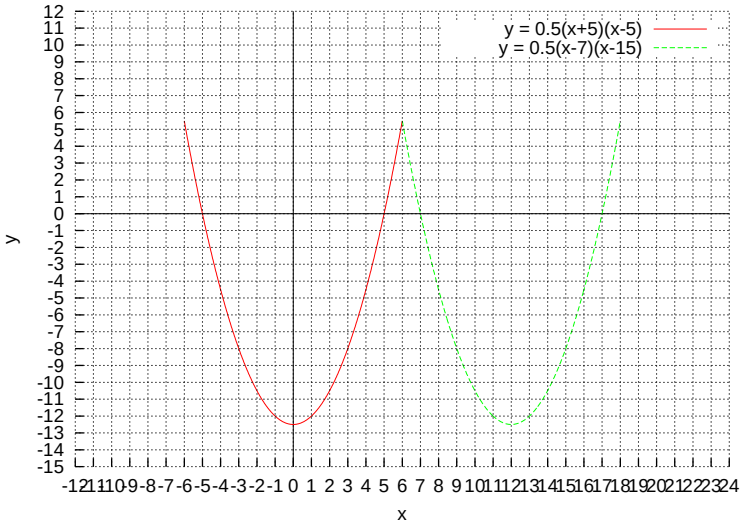


Figure 14.9: The plot for two polynomials traveled by two robot from Programs 14.4, 14.5, and 14.7.

14.3.2 Move Multiple Linkbot-Is Along Different Trajectories Using Functions

Similar to the method described in section 9.3.2, in this section, we will use member function **drivexyToFunc**() and non-blocking member function **drivexyToFuncNB**(). Program 14.5 moves two robots along two polynomial trajectories specified in Equations (14.1) and (14.2).

The syntax of the member function **drivexyToFuncNB**() is as follows.

```
robot.drivexyToFuncNB(x0, xf, num, func, radius, trackwidth);
```

This non-blocking member function drives a Linkbot-I following the trajectory specified by the function defined as `func()` in the range `[x0, xf]` using `num` points. The last two arguments specify the radius of the two wheels and the track width of the robot.

Program 14.5 is the same as Program 14.4, except that it uses the statement

```
robot1.drivexyToFuncNB(-6, 6, 30, func, radius, trackwidth);
robot2.drivexyToFunc(6, 18, 30, func2, radius, trackwidth);
robot1.drivexyWait();
```

to drive the robot 1 and robot 2 along two parabolas through the functions `func()` and `func2()`, instead of using the member functions **drivexyToExpr**() and **drivexyToExprNB**().

In Program 14.5, the functions `func()` and `func2()` are defined as follows.

```
/* define the function func() */
double func(double x) {
    return 0.5*(x+5)*(x-5);
}

/* define the function func2() */
double func2(double x) {
    return 0.5*(x-7)*(x-17);
}
```

The relations represented in these functions in Program 14.5 are plotted by using the **CPlot** member function **func2D()** with statement below.

```
plot.func2D(-6, 6, 500, func);
plot.legend("0.5*(x+5)*(x-5)");
```

```
    plot.func2D(6, 18, 500, func2);
    plot.legend("0.5*(x-7)*(x-17)");
```

The output from Program 14.5 is the same as that from Program 14.4, as shown in Figures 14.8 and 14.9.

```
/* File: polynomialsfunc.ch
   Note: This program uses drivexyToFunc() and drivexyToFuncNB().
   Set the initial position (x, y) in RoboSim GUI to (-6, 5.5) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 5.5) for robot2.

   robot1 moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6.
   robot2 moves along a polynomial curve y = 0.5(x-7)(x-17) for x from 6 to 18.
   Move both robots with 30 points. */
#include <chplot.h>
#include <linkbot.h>
CPlot plot;
CLinkbotI robot1;
CLinkbotI robot2;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels

double func(double x) {
    return 0.5*(x+5)*(x-5);
}

double func2(double x) {
    return 0.5*(x-7)*(x-17);
}

// uncomment lines below for controlling hardware robots
//robot1.initPosition(-6, 5.5, 90);
//robot2.initPosition(6, 5.5, 90);

/* change the default green color to red for robot 1 */
robot1.setLEDColor("red");

/* move the robots along the polynomial curves */
robot1.drivexyToFuncNB(-6, 6, 30, func, radius, trackwidth);
robot2.drivexyToFunc(6, 18, 30, func2, radius, trackwidth);
robot1.drivexyWait();

/* plot the polynomial curve */
plot.title("");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 24);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_XY, 1);
plot.func2D(-6, 6, 500, func);
plot.legend("y = 0.5(x+5)(x-5)");
plot.func2D(6, 18, 500, func2);
plot.legend("y = 0.5(x-7)(x-17)");
plot.plotting();
```

Program 14.5: Moving two Linkbot-Is to follow polynomial curves using **drivexyToFunc**() and **drivexyToFuncNB**().

E Do Exercise 2 on page 302.

14.3.3 Move Multiple Linkbot-Is Along Different Trajectories Using a Loop

When a robot travels along a trajectory, it must travel to many more points. Internally, the motion for the member functions **drivexyToExpr**(), **drivexyToExprNB**(), **drivexyToFunc**(), and **drivexyToFuncNB**() are actually implemented using the member function **drivexyTo**() and **drivexyToNB**() using a loop.

The use of a **for** loop together with a user-defined function is helpful for calculating the next point and then calling **drivexyTo**() multiple times in succession. Details for the **for** are described in section A.3 in Appendix A. In this section, we will first use the member function **drivexyTo**() with a **for** loop to move a single Linkbot-I along a trajectory. Then, we will use both the member function **drivexyTo**() and its non-block version **drivexyToNB**() to understand how to move two robots to trace two different trajectories at the same time.

```
/* File: polynomial.ch
   Note: This program uses drivexyTo() available in RoboSim only.
   Set the initial position (x, y) in RoboSim GUI to (-6, 5.5) for the robot.

   A robot moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6.
   Plot the polynomial y for x from -6 to 6 with 500 points.
   The range of x-axis is from -12 to 12.
   The range of y-axis is from -15 to 12.
   The tics range for x and y axes is 1. */
#include <chplot.h>
#include <linkbot.h>
CPlot plot;
CLinkbotI robot;
double radius = 1.75;       // radius of 1.75 inches
double trackwidth = 3.69;   // the track width, the distance between two wheels
double x, y;                // x and y coordinates

double func(double x) {
    return 0.5*(x+5)*(x-5);
}

robot.setSpeed(3, 1.75);

/* move the robot along the polynomial curve */
for(x=-6; x<=6.1; x=x+0.4) {
    y = func(x);
    printf("drivexyTo(x, y) = (%lf, %lf)\n", x, y);
    robot.drivexyTo(x, y, radius, trackwidth);
}

/* plot the polynomial curve */
plot.title("y = 0.5(x+5)(x-5)");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 12);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.func2D(-6, 6, 500, func);
plot.plotting();
```

Program 14.6: Moving a Linkbot-I to follow a polynomial curve using **drivexyTo**() and a **for** loop.

Program 14.6 shows how this can be done to drive a Linkbot-I along the trajectory of a parabola. Before running Program 14.6, set the initial position of the Linkbot-I in the RoboSim GUI to $(-6, 5.5)$. In Program 14.6, a function `func()` is used to define this quadratic equation, which calculates the y-coordinate for each point the Linkbot-I travels on the parabola. The function `func()` is defined as

```
    /* define the function func() */
    double func(double x) {
        return 0.5*(x+5)*(x-5);
    }
```

The lines

```
    for(x=-6; x<=6.1; x=x+0.4) {
        y = func(x);
        printf("drivexyTo(x, y) = (%lf, %lf)\n", x, y);
```

```
        robot.drivexyTo(x, y, radius, trackwidth);
    }
```

use a **for** loop to repeat the program statements that drive the Linkbot-I to each point along its trajectory. Each time the loop repeats, the value of the x-coordinate is incremented by 0.4, and the corresponding y-coordinate is calculated using `func()`. Then the x and y coordinates are printed to the input/output pane, and **drivexyTo**() is used to drive the Linkbot-I to these coordinates. Since the value of the x-coordinate is incremented by 0.4 each iteration, the **for** loop repeats a total of 30 times, and the Linkbot-I is driven a total of 30 points along the trajectory of the parabola. Although **drivexyTo**() drives the Linkbot-I in a straight line between the points, the points are close enough together that these lines appear as a smooth curve in the RoboSim GUI.

The output from Program 14.6 is the same as that from Programs 9.5 and 9.6, as shown in Figures 9.13 and 9.14.

E Do Exercise 3 on page 302.

Just as it is possible to drive two Linkbot-Is along two separate straight-lines simultaneously as shown in Program 14.1, it is also possible to simultaneously drive two Linkbot-Is along two separate polynomial curves. Program 14.7 demonstrates how to do this using **drivexyToNB**(). Before running Program 14.7, set the initial positions of robot 1 and robot 2 to $(-6, 5.5)$ and $(6, 5.5)$ in the RoboSim GUI.

```
/* File: polynomial2.ch
   Note: This program uses drivexyToNB() and drivexyWait() available in RoboSim only.
   Set the initial position (x, y) in RoboSim GUI to (-6, 5.5) for robot1.
   Set the initial position (x, y) in RoboSim GUI to (6, 5.5) for robot2.

   robot1 moves along a polynomial curve y = 0.5(x+5)(x-5) for x from -6 to 6.
   robot2 moves along a polynomial curve y = 0.5(x-7)(x-17) for x from 6 to 18. */
#include <chplot.h>
#include <linkbot.h>
CPlot plot;
CLinkbotI robot1;
CLinkbotI robot2;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69;  // the track width, the distance between two wheels
double x, y1, y2;          // x and y coordinates

double func(double x) {
    return 0.5*(x+5)*(x-5);
}

double func2(double x) {
    return 0.5*(x-7)*(x-17);
}

/* change the default green color to red for robot 1 */
robot1.setLEDColor("red");

/* move the robot along the polynomial curve */
for(x=-6; x<=6; x=x+0.4) {
    y1 = func(x);
    y2 = func2(x+12);
    robot1.drivexyToNB(x, y1, radius, trackwidth);
    robot2.drivexyTo(x+12, y2, radius, trackwidth);
    robot1.drivexyWait();
}

/* plot the polynomial curve */
plot.title("");
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y");
plot.axisRange(PLOT_AXIS_X, -12, 24);
plot.axisRange(PLOT_AXIS_Y, -15, 12);
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 1);
plot.func2D(-6, 6, 500, func);
plot.legend("y = 0.5(x+5)(x-5)");
plot.func2D(6, 18, 500, func2);
plot.legend("y = 0.5(x-7)(x-17)");
plot.plotting();
```

Program 14.7: Moving two Linkbot-Is to follow polynomial curves using **drivexyTo**() and **drivexyToNB**().

Program 14.6 demonstrated how to drive a single Linkbot-I along the trajectory of a parabola. Program 14.7 is very similar to Program 14.6, except that two Linkbot-Is are driven along different parabolic trajectories at the same time.

As in Program 14.6, the user-defined function `func()` is used to calculate the y-coordinates of the points on the first parabola. For the second parabola, the new user-defined function `func2()`, is defined

by the following program statements.

```
double func2(double x) {
    return 0.5*(x-7)*(x-17);
}
```

The two Linkbot-Is are driven along their parabolic trajectories using the following **for** loop

```
for(x=-6; x<=6; x=x+0.4) {
    y1 = func(x);
    y2 = func2(x+12);
    robot1.drivexyToNB(x, y1, radius, trackwidth);
    robot2.drivexyTo(x+12, y2, radius, trackwidth);
    robot1.drivexyWait();
}
```

The **for** loop in Program 14.7 is similar to the **for** loop in Program 14.6. The difference is that in Program 14.7 the statements repeated in the **for** loop calculate the y-coordinates of the next points for `robot1` and `robot2`, using `func()` and `func2()`, respectively. Then both `robot1` and `robot2` are driven to their new points simultaneously. Since the non-blocking member function **drivexyToNB**() is used to drive `robot1`, **drivexyWait()** is called to ensure that `robot1` will finish its motion before the next iteration of the **for** loop. As in Program 14.6, the **for** loop will iterate a total of 30 times, so each Linkbot-I will drive through a total of 30 points. Both parabolas will look similar, except that the parabolic trajectory of `robot2` will be from (6, 5.5) to (18, 5.5). Figure 14.8 shows the parabolic trajectories of `robot1` and `robot2`.

At the end of Program 14.7, the trajectories of two Linkbot-Is are plotted using the same code in Program 14.5.

E Do Exercise 4 on page 302.

14.3.4 Summary

1. Call the member functions **drivexyTo**(), **drivexyToNB**(), and **drivexyWait**() in a loop to drive multiple Linkbot-Is simultaneously along different trajectories.

2. Call the non-blocking **CLinkbotI** member function

   ```
   robot.drivexyToExprNB(x0, xf, num, expr, radius, trackwidth);
   ```

   drives a Linkbot-I based on an expression `expr` in terms of the variable x, for x from `x0` to `xf` with `num` number of points.

3. Call the non-blocking **CLinkbotI** member function

   ```
   robot.drivexyToFuncNB(x0, xf, num, func, radius, trackwidth);
   ```

   drives a Linkbot-I based on a function `func` in terms of the variable x, for x from `x0` to `xf` with `num` number of points.

14.3.5 Terminology

robot.drivexyToExprNB(), **robot.drivexyToFuncNB**().

14.3.6 Exercises

1. Write a program `polynomials2.ch`, based on Program 14.4, that drives two Linkbot-Is along the trajectory of two parabolas, from $(-6, -5.5)$ to $(6, -5.5)$ for `robot1` and from $(6, -5.5)$ to $(18, -5.5)$ for `robot2`. The two parabolas are defined by the formulas $y_1 = -0.5(x + 5)(x - 5)$ and $y_2 = -0.5(x - 7)(x - 17)$. The program shall use member functions **drivexyToExpr**() and **drivexyToExprNB**(). These parabolas will open downward, with the vertex of the first at (0, 12.5) and the vertex of the second at (12, 12.5). Use the CPlot member functions **axisRange()**, **ticsRange()**, and **expr()** to generate a plot of the trajectories of the two Linkbot-Is.

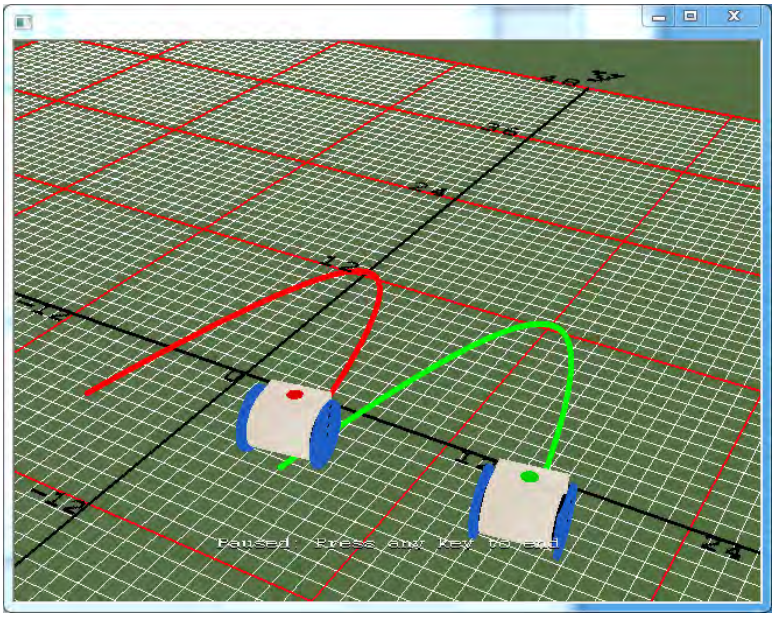


Figure 14.10: The RoboSim scene with the robot trajectories for the program `polynomials2.ch`

2. Write a program `polynomialsfunc2.ch`, based on Program 14.5, that drives two Linkbot-Is along the trajectory of two parabolas described in Exercise 1 as shown in Figure 14.10. Use the CPlot member function **func2D()** to generate a plot of the trajectories of the two Linkbot-Is.

3. Write a program `polynomial3.ch`, using the member function **drivexyTo**() and a **for** loop, based on Program 14.6, that drives a Linkbot-I along the trajectory of a parabola from (-6, -5.5) to (6, -5.5), as shown in Figure 9.15 on page 155.

4. Write a program `polynomial4.ch`, based on Program 14.7, that drives two Linkbot-Is along the trajectory of two parabolas described in Exercise 1 as shown in Figure 14.10. Use the CPlot member function **func2D()** to generate a plot of the trajectories of the two Linkbot-Is.

CHAPTER 15

Writing Programs to Control One or Multiple Groups of Linkbots

In Chapter 7, we learned how to control a group of Linkbots with identical movements. In Chapter 13, we learned how to control multiple individual Linkbots in synchronous motions. In this chapter, we will learn how to control multiple groups of Linkbots in synchronous motions. Many of the non-blocking **CLinkbot** member functions from Chapter 13 have **CLinkbotIGroup** counterparts. With these functions, choreographed synchronized motions can be achieved simply using a few lines of code.

15.1 Copy Motions from One Linkbot to Multiple Linkbots with Identical Movements

In Section 13.11 we learned how to copy the motions of one Linkbot to another. In this section we will extend Program 13.17 to copy the motions of one Linkbot to a group of Linkbots. The following program allows the user to control two Linkbot-Is simultaneously using a single Linkbot-I as a controller.

```
/* File: copycatgrouptwo.ch
   One robot controls two robots, either individually or connected.
   robot1 is the controller
*/
#include <linkbot.h>
CLinkbotI robot1, robot2, robot3;
CLinkbotIGroup group;
/* angles from robot1 */
double angle1, angle3;

group.addRobot(robot2);
group.addRobot(robot3);
```

```
/* move to the zero position at the same time. */
robot1.resetToZeroNB();
group.resetToZeroNB();
robot1.moveWait();
group.moveWait();

/* relax all joints of robot1 */
robot1.relaxJoints();

printf("You can now move joints of robot1 to control robot2 and 3\n");
while(1){
    /* get joint angles of robot1 */
    robot1.getJointAngles(angle1, NaN, angle3);

    /* move  robots in the group by the angles from robot1 */
    group.moveToByTrackPosNB(angle1, NaN, angle3);
}
```

Program 15.1: Copying motions of a controller Linkbot to a group of two controlled Linkbots.

You can run Program 15.1 to copy the motion of a Linkbot to two Linkbots in a group simultaneously. The **CLinkbotIGroup** class includes non-blocking member functions that are similar to those in class **CLinkbotI**. Examples seen in Program 15.1 include **resetToZero()**, **moveWait()**, and **moveToByTrackPosNB()**. The **CLinkbotIGroup** versions of these member functions move all the Linkbot-Is in a group identically, instead of only a single Linkbot-I.

In Program 15.1 `robot2` and `robot3` are added to `group`. This group is used to copy the motions of `robot1` to `robot2` and `robot3` simultaneously. The line

```
    group.resetToZeroNB();
```

is similar to the preceding statement **robot1.resetToZeroNB()**. The only difference is that multiple Linkbot-Is are reset to the zero position using only one line of code. This non-blocking function allows the next line of code to execute simultaneously. The next two lines

```
    robot1.moveWait();
    group.moveWait();
```

pause the program until the joints of all Linkbots are finished resetting to zero. The statement **group.moveWait()** is similar to the previous line of code. The only difference is that **group.moveWait()** pauses the program for multiple Linkbot-Is using only one line of code. Once the joints of `robot1` are set to neutral, the loop

```
    while(1){
        /* get joint angles of robot1 */
        robot1.getJointAngles(angle1, NaN, angle3);

        /* move robots in the group by the angles from robot1 */
        group.moveToByTrackPosNB(angle1, NaN, angle3);
    }
```

copies the joint motions from the controller Linkbot-I to the Linkbot-Is in `group`. This **while** loop is similar to the **while** loop in Program 13.17. The key difference is the line

```
    group.moveToByTrackPosNB(angle1, NaN, angle3);
```

which drives the joints of all the Linkbot-Is in `group` to the absolute joint angle position copied from `robot1`. Thus `robot2` and `robot3` will copy `robot1` in identical motions. Since **moveToByTrackPosNB()** is non-blocking, the loop will start to repeat before `robot2` and `robot3` have completed their motions.

E Do Exercises 1 and 2 on page 305.

15.1.1 Summary

1. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.resetToZeroNB();
   ```

   to reset joints 1 and 3 of all Linkbot-Is in a group to the zero position. The next line of code will begin executing before the function **resetToZeroNB()** has finished.

2. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.moveWait();
   ```

   to pause the program until all Linkbot-Is in the group have finished their motions.

3. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.moveToByTrackPosNB(angle1, NaN, angle3);
   ```

   to rotate the joint positions of all Linkbot-Is in a group to their absolute joint angle positions by tracking their positions. The next line of code will begin executing before the function **moveToByTrackPosNB()** has finished.

15.1.2 Terminology

group.resetToZeroNB(), **group.moveWait()**, **group.moveToByTrackPosNB()**.

15.1.3 Exercises

1. Run the program `copycatgrouptwo.ch` in Program 15.1 to copy the motion of a controller Linkbot to two controlled Linkbots.

2. Write a program `copycatgroupthree.ch`, based on the program `copycatgrouptwo.ch` in Program 15.1, to copy the motion of the controller Linkbot to three controlled Linkbots.

15.2 Control Multiple Groups of Linkbots

It is also possible for a single Linkbot-I to belong to multiple groups. Any command given to a group where a Linkbot is a part of that group will result in that Linkbot moving. Using multiple groups can help synchronize Linkbots in more complex patterns of movements.

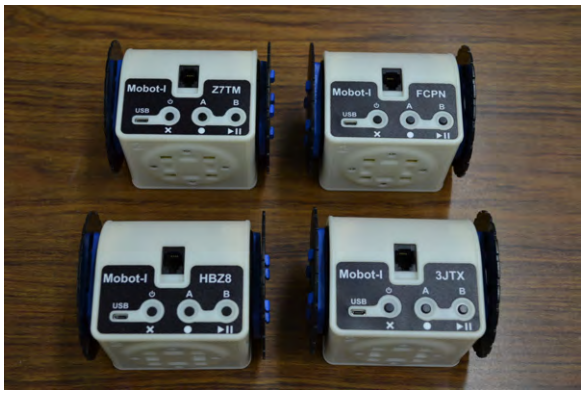


Figure 15.1: Controlling multiple Linkbot groups.

```
/* File: groups.ch
   Control multiple robot groups simultaneously using the CLinkbotGroup class */
#include <linkbot.h>
CLinkbotI robot1, robot2, robot3, robot4;
CLinkbotIGroup groupA, groupB, groupC, groupD;
double radius = 1.75;      // radius of 1.75 inches
double trackwidth = 3.69; // the track width, the distance between two wheels

/* add the robots to groups. The groups are organized as such:
   Group A: 1, 2, 3, 4
   Group B: 1, 2
   Group C: 3, 4
   Group D: 1, 2, 3 */
/* Group A */
groupA.addRobot(robot1);
groupA.addRobot(robot2);
groupA.addRobot(robot3);
groupA.addRobot(robot4);
/* Group B */
groupB.addRobot(robot1);
groupB.addRobot(robot2);
/* Group C */
groupC.addRobot(robot3);
groupC.addRobot(robot4);
/* Group D */
groupD.addRobot(robot1);
groupD.addRobot(robot2);
groupD.addRobot(robot3);

/* make group B roll forward and group C roll backward at the same time */
groupB.driveAngleNB(360);
groupC.driveAngleNB(-360);
groupB.moveWait();
groupC.moveWait();

/* make all robots roll forward */
groupA.driveAngle(360);

/* make robots 1 and 2 (Group B) turn left, and robots 3 and 4 (Group C) turn right. */
groupB.turnLeftNB(90, radius, trackwidth);
groupC.turnRightNB(90, radius, trackwidth);
groupB.moveWait();
groupC.moveWait();

/* make robot 4 roll forward
   while three robots in group D drive backward. */
groupD.driveAngleNB(-360);
robot4.driveAngleNB(360);
groupD.moveWait();
robot4.moveWait();
```

Program 15.2: A program with multiple groups.

Program 15.2 coordinates four Linkbot-Is using four different groups to perform independent, synchronous motions. All four Linkbot-Is are added to `groupA` using **addRobot()**. Then `robot1` and `robot2` are added to `groupB`. Similarly, `robot3` and `robot4` are added to `groupC`. Finally `robot1`, `robot2`, and `robot3` are added to `groupD`. Then the line

```
groupA.resetToZero();
```

resets all Linkbot-Is to the zero position. Since all four Linkbot-Is are in `groupA`, there is no need to call **resetToZero()** on any of the other groups. The next line

```
groupB.driveAngleNB(360);
```

drives `robot1` and `robot2` forward 360 by degrees at the same time using the **CLinkbotIGroup** member function **driveAngleNB()**. The general syntax of this function is

```
group.driveAngleNB(angle);
```

where the argument `angle` specifies the amount to roll the wheels forward relative to their current positions. Since this function is non-blocking, the next line of code starts executing immediately. The next line

```
groupC.driveAngleNB(-360);
```

drives `robot3` and `robot4` backward 360 by degrees at the same time that `robot1` and `robot2` roll forward. Since **driveAngleNB()** is non-blocking, the next two lines

```
groupB.moveWait();
groupC.moveWait();
```

are required to pause the program until `groupB` and `groupC` are done with their movements. Then the following line

```
groupA.driveAngle(360);
```

drives all four Linkbot-Is forward by 360 degrees. The blocking version of the function is used in this case, since all four Linkbots have the same motion. The next line

```
groupB.turnLeftNB(90, radius, trackwidth);
```

causes `robot1` and `robot2` to turn left by 90 degrees at the same time. The **CLinkbotIGroup** member function **turnLeftNB()** has the following general syntax

```
group.turnLeftNB(angle, radius, trackwidth);
```

The argument `angle` specifies the amount to turn left. The argument `radius` specifies the radius of the two wheels. The argument `trackwidth` specifies the distance between the two wheels. The units must be the same for `radius` and `trackwidth`. Since **turnLeftNB()** is non-blocking, the next line of code

```
groupC.turnRightNB(90, radius, trackwidth);
```

will execute synchronously. Thus `robot3` and `robot4` will turn right by 90 degrees at the same time that `robot1` and `robot2` are turning left. The general syntax of the member function **turnRightNB()** is

```
group.turnRightNB(angle, radius, trackwidth);
```

The argument `angle` specifies the amount to turn right. The argument `radius` specifies the radius of the two wheels. The argument `trackwidth` specifies the distance between the two wheels. Since **turnRightNB()** is a non-blocking function, then **moveWait()** needs to be called on `groupB` and `groupC` to pause the program. Once both groups have finished their motions, the line

```
groupD.driveAngleNB(-360);
```

drives `robot1`, `robot2`, and `robot3` backward by 360 degrees at the same time. The following line

```
robot4.driveAngleNB(360);
```

drives `robot4` forward by 360 degrees at the same time the Linkbot-Is in `groupD` are moving forward. Note that the member function **driveAngleNB()** is being used on a single Linkbot-I instead of a group. The member functions **turnLeftNB()** and **turnRightNB()** also have counterparts in the **CLinkbotI** class. Finally, the lines

```
groupD.moveWait();
robot4.moveWait();
```

pause the program until `groupD` and `robot4` finish their motions.

E Do Exercises 1 2 on page 310.

15.2.1 Summary

1. Call the non-blocking **CLinkbotI** member function

   ```
   robot.driveAngleNB(angle);
   ```

   to drive a single Linkbot-I forward or backward by the specified angle relative to its current position. The next line of code begins execution before this motion has finished.

2. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.driveAngleNB(angle);
   ```

   to drive all Linkbot-Is in a group forward or backward by the specified angle relative to its current position. The next line of code begins execution before this motion has finished.

3. Call the non-blocking **CLinkbotI** member function

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

   to turn a single Linkbot-I left by the specified angle. The next line of code begins execution before this motion has finished.

4. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.turnLeftNB(angle, radius, trackwidth);
   ```

   to turn all Linkbot-Is in a group left by the specified angle. The next line of code begins execution before this motion has finished.

5. Call the non-blocking **CLinkbotI** member function

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

   to turn a single Linkbot-I right by the specified angle. The next line of code begins execution before this motion has finished.

6. Call the non-blocking **CLinkbotIGroup** member function

   ```
   group.turnRightNB(angle, radius, trackwidth);
   ```

   to turn all Linkbot-Is in a group right by the specified angle. The next line of code begins execution before this motion has finished.

15.2.2 Terminology

robot.driveAngleNB(), **robot.turnLeftNB()**, **robot.turnRightNB()**, **group.driveAngleNB()**, **group.turnLeftNB()**, **group.turnRightNB()**, multiple groups.

15.2.3 Exercises

1. Run the program `groups.ch` in Program 15.2 to control multiple Linkbot groups.

2. Based on the program `groups.ch` in Program 15.2, write a program `groups2.ch` to control four Linkbots. The program `groups2.ch` has four Linkbot groups of A, B, C, and D, similar to groups in Program 15.2. These groups will complete the following motions in order: First, Group A rolls forward for two full rotations of 720 degrees, then Group B turns left for 90 degrees, afterwards Group C turns right for 180 degrees, next Group D rolls forward for 360 degrees, and finally Linkbot 4 rolls backward for 360 degrees.

CHAPTER 16

Controlling Multiple Connected Linkbots

In Chapters 13 and 15, we learned how to control multiple individual Linkbots and multiple groups of Linkbots in synchronous motions, respectively. In this chapter, we will learn how to control connected Linkbots as well as groups of connected Linkbots in synchronous motions.

16.1 Control Multiple Connected Linkbots

In this section, we will learn how to write various programs for controlling multiple connected Linkbot-Is. Program 16.1 controls two Linkbot-Is connected in the four-wheel drive configuration shown in Figure 16.1. To assemble this configuration, connect two simple connectors to opposite sides of a cubic connector. Then attach joint 2, which is the non-moving joint, of the two Linkbot-Is to the simple connectors on the cubic connector. Be sure to attach wheels to joints 1 and 3 of both Linkbot-Is.

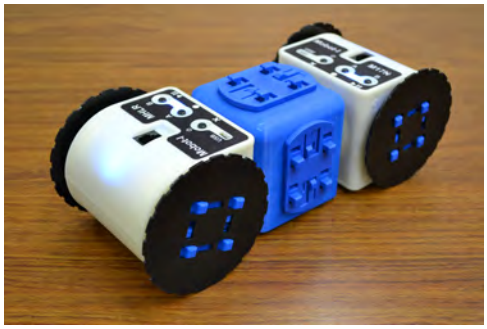


Figure 16.1: A four-wheel drive configured with two Linkbot-Is.

```
/* File name: fourwheeldrive.ch
 * Four wheel drive configuration with two Linkbot-Is
 *          Top view
 *
 *           |----|
 *           |____| <- robot1
 *            |--|
 *            |__| <- cubic connector
 *           |----|
 *           |____| <- robot2
 *
 * Note: attach wheels for two Linkbot-I's */
#include <linkbot.h>
CLinkbotI robot1, robot2;

/* drive forward */
robot1.driveAngleNB(360);
robot2.driveAngleNB(-360);
robot1.moveWait();
robot2.moveWait();

/* drive backward */
robot1.driveAngleNB(-360);
robot2.driveAngleNB(360);
robot1.moveWait();
robot2.moveWait();
```

Program 16.1: Controlling two Linkbot-Is as a four-wheel drive vehicle.

Program 16.1 moves the four-wheel drive forward by 360 degrees, then backward by 360 degrees. The lines

```
    robot1.driveAngleNB(360);
    robot2.driveAngleNB(-360);
```

use non-blocking functions to drive `robot1` forward by 360 degrees and `robot2` backward by 360 degrees simultaneously. The simultaneous motions of `robot1` and `robot2` result in the four-wheel drive moving forward. The program pauses until both `robot1` and `robot2` have finished their motions. Then the lines

```
    robot1.driveAngleNB(-360);
    robot2.driveAngleNB(360);
```

use non-blocking functions to drive `robot1` backward by 360 degrees and `robot2` forward by 360 degrees simultaneously. The simultaneous motions of both Linkbot-Is result in the four-wheel drive moving backward.

E Do Exercise 1, on page 318.

Other considerations may arise when dealing with multiple-configuration designs. To accomplish certain motions, it may be necessary to perform motions in steps, with delays and pauses for motions to finish, before doing the next motion. The next program demonstrates how to address this consideration, controlling four Linkbot-Is in lifting and lowering motions as shown in Figure 16.2.

To assemble the configuration in Figure 16.2, connect `robot1` and `robot2` with a bridge connector at joint 1 of both Linkbot-Is. Then connect `robot1` and `robot2` with another bridge connector at joint 3 of both Linkbot-Is. `robot3` and `robot4` will be connected in the same way as `robot1` and `robot2`. Then connect `robot2` and `robot3` with a simple connector at joint 2 of both Linkbot-Is.

(a) original position (b) First lift (c) Second lift (d) Third lift (e) original position

Figure 16.2: Lifting and unlifting motions with four connected modules.

```
/* File: lift.ch
 * Lift four connected Linkbot-Is
 *             Top View
 *
 *         -----       -----
 *       +--+ +--+   +--+ +--+
 *       | 1| | 2| | | 3| | 4|
 *       +--+ +--+   +--+ +--+
 *         -----       -----
 *           ^     ^
 *           |     |
 *         Bridge  |__  Simple
 *        Connecter    Connecter
 * The joint 1's of both robot 1 and robot 2  are connected with the bridge connecter
 * The joint 3's of both robot 1 and robot 2  are connected with the bridge connecter
 * The same configuration for connecting robot 3 and robot 4.
 * Also, robot 2 and robot 3 are connected by a simple connecter at joint 2's
 * of both robots.
 */
#include <linkbot.h>
CLinkbotI robot1, robot2, robot3, robot4;

/* move to zero position */
robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.resetToZeroNB();
robot4.resetToZero();

/* first lift */
robot1.moveToNB(90, NaN, -90);
robot4.moveTo(-90, NaN, 90);
robot1.delaySeconds(1);

/* second lift */
robot1.moveToNB(45, NaN, -45);
robot4.moveToNB(-45, NaN, 45);
robot2.moveToNB(-45, NaN, 45);
robot3.moveTo(45, NaN, -45);
robot1.delaySeconds(1);

/* third lift */
robot1.moveToNB(0, NaN, 0);
robot4.moveToNB(0, NaN, 0);
robot2.moveToNB(-90, NaN, 90);
robot3.moveTo(90, NaN, -90);
robot1.delaySeconds(1);
```

```
/* unlift */
robot2.moveToNB(0, NaN, 0);
robot3.moveTo(0, NaN, 0);
```

Program 16.2: Lifting for four connected Linkbot-Is.

Program 16.2 controls four connected Linkbot-Is in lifting and unlifting motions. The lines

```
    robot1.resetToZeroNB();
    robot2.resetToZeroNB();
    robot3.resetToZeroNB();
    robot4.resetToZero();
```

resets joints 1 and 3 of all four Linkbot-Is simultaneously. Notice that the blocking version of the member function **resetToZero()** is used to reset `robot4`. This eliminates the need to call **moveWait()** on the four Linkbot-Is, since the next line of code will not begin executing until `robot4` has finished resetting to the zero position. Using a blocking version of a motion function for the last robot in a configuration can sometimes help reduce the size of the program. The next lines

```
    robot1.moveToNB(90, NaN, -90);
    robot4.moveTo(-90, NaN, 90);
```

perform the first stage of the lift. The feet of this configuration, `robot1` and `robot4`, rotate simultaneously to absolute joint angle positions of 90 degrees and -90 degrees respectively. This puts the feet of the configuration in the correct position for the second stage of the lift. Since the blocking version of the member function **moveTo()** is used on `robot4`, there is no need to use **moveWait()** to pause the program. However, the next line

```
    robot1.delaySeconds(1);
```

inserts a one-second pause after the first stage of the lift to help differentiate between the lift stages. Pausing between stages gives more stability to the motions of this particular configuration. The following lines

```
    robot1.moveToNB(45, NaN, -45);
    robot4.moveToNB(-45, NaN, 45);
    robot2.moveToNB(-45, NaN, 45);
    robot3.moveTo(45, NaN, -45);
    robot1.delaySeconds(1);
```

perform the second stage of the lift. The feet of the configuration, `robot1` and `robot4`, rotate simultaneously to absolute joint angle positions of 45 degrees and -45 degrees respectively. At the same time, `robot2` and `robot3` also rotate to absolute joint angle positions of -45 degrees and 45 degrees respectively. The simultaneous motions of the four Linkbot-Is help lift `robot2` and `robot3` up off the ground. The program is then delayed for another second, to help stabilize the connected Linkbot-Is in this position. The next lines

```
    robot1.moveToNB(0, NaN, 0);
    robot4.moveToNB(0, NaN, 0);
    robot2.moveToNB(-90, NaN, 90);
    robot3.moveTo(90, NaN, -90);
    robot1.delaySconds(1);
```

perform the third stage of the lift. The feet of the configuration, `robot1` and `robot4`, rotate simultaneously to absolute joint angle positions of 0 degrees. Meanwhile `robot2` and `robot3` also rotate to absolute joint angle positions of -90 degrees and 90 degrees respectively. After the simultaneous motions of the four Linkbot-Is have completed, the configuration will be standing with its legs completely upright, feet touching each other. Another one second delay helps stabilize the configuration in this position. Then the final lines

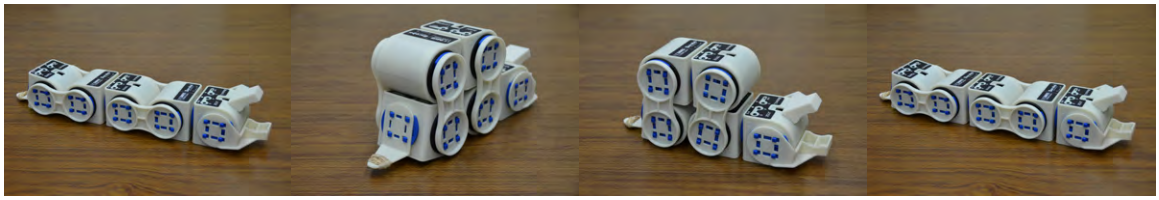
```
    robot2.moveToNB(0, NaN, 0);
    robot3.moveTo(0, NaN, 0);
```

lower all four Linkbot-Is back down to their original positions. Since `robot1` and `robot4` already have their joints in the zero position after stage three of the lift, they do not need to be moved during the lowering stage. So only the joints of `robot2` and `robot3` need to be moved back to the zero position in order to lower the configuration back down to the floor.

E Do Exercise 2 on page 318.

Program 16.3 moves five connected Linkbot-Is in a snake configuration. To assemble the snake configuration, attach joints 1 and 3 of `robot5` and `robot4` with bridge connectors. Be sure that joint 3 of both Linkbot-Is are connected with the same bridge connector. Connect `robot3` and `robot2` in the same way as `robot5` and `robot4`. Attach the upper part of the claw to joint 1 of `robot1` and the lower part to joint 3. This will be the head of the snake. Connect `robot3` and `robot4` with a simple connector at joint 2 of both Linkbot-Is to form the body of the snake. Make sure that all four Linkbot-Is in the body of the snake are connected so that joint 3 of each Linkbot is on the same side. Connect the head of the snake to the body with a simple connector at joint 2 of `robot1` and `robot2`. The head of the snake should have joint 3 on the same side as all the Linkbots in the body of the snake. Finally, connect a caster to joint 2 of `robot5` to give the snake a tail. Wrap rubber bands around the tail to give the snake the right amount of friction.

The video for the snake motion can be viewed in a video file SnakeMotin.mp4 available at c-stem.ucdavis.edu.



(a) Snake position 1 (b) Snake position 2 (c) Snake position 3 (d) Snake position 1

Figure 16.3: The snake motion with five Linkbot-Is.

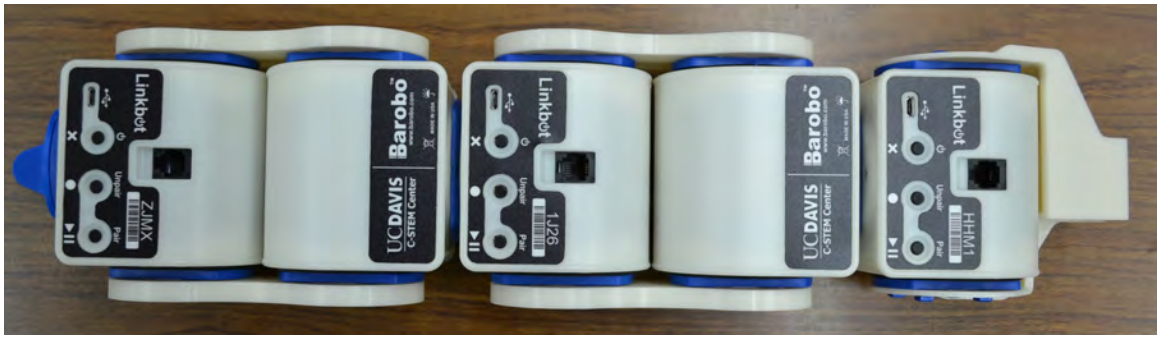


Figure 16.4: The configuration for a snake with five Linkbot-Is. Joint 3 for all Linkbot-Is are on the same side.

```
/* File: snake.ch
 * This program uses 5 Linkbot-Is for snake motion.
 * The linkbots are connected by bridges and simple connectors.
 * The robot1 has a claw attached as a mouth. The robot5 has a caster on
 * the back as a tail. The rubber-bands on the tail provide the needed friction.
```

```
 *
 * Initial configuration: Joint 3 for all Linkbot-Is are on the same side
 * as shown in snake.jpg.
 *
 *             Top View
 *
 *        -----       -----
 *      +--+ +--+   +--+ +--+   +--+
 *   _| | 5| | 4| | | 3| | 2| | | 1|  <- Claw
 *      +--+ +--+   +--+ +--+   +--+
 *    ^   -----       -----
 *    |     ^                 ^
 * Caster   |                 |
 *        Bridge            Simple
 *       Connecter         Connecter
 */
#include <linkbot.h>
CLinkbotI robot1, robot2, robot3, robot4, robot5;

robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.resetToZeroNB();
robot4.resetToZeroNB();
robot5.resetToZeroNB();
robot1.moveWait();
robot2.moveWait();
robot3.moveWait();
robot4.moveWait();
robot5.moveWait();

/* Lift the tail up */
robot5.moveTo(-30, NaN, 30);

/* Move the body of the snake together */
robot1.moveToNB(-10, NaN, -30);
robot2.moveToNB(95, NaN, -95);
robot3.moveToNB(95, NaN, -95);
robot4.moveToNB(-95, NaN, 95);
robot5.moveToNB(-110, NaN, 110);
robot1.moveWait();
robot2.moveWait();
robot3.moveWait();
robot4.moveWait();
robot5.moveWait();

/* Set the tail down */
robot3.moveToNB(85, NaN, -85);
robot4.moveToNB(-100, NaN, 100);
robot5.moveToNB(-85, NaN, 85);
robot3.moveWait();
robot4.moveWait();
robot5.moveWait();

/* Set the body flat to move it forward */
robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.resetToZeroNB();
robot4.resetToZeroNB();
robot5.resetToZeroNB();
```

```
robot1.moveWait();
robot2.moveWait();
robot3.moveWait();
robot4.moveWait();
robot5.moveWait();
```

Program 16.3: The Snake motion with five connected Linkbot-Is.

Program 16.3 moves five connected Linkbot-Is in the motion of a snake. The line

```
    robot5.moveTo(-30, NaN, 30);
```

lift up the tail of the snake by rotating `robot5` to an absolute joint angle position of 30 degrees. The next lines

```
    /* Move the body of the snake together */
    robot1.moveToNB(-10, NaN, -30);
    robot2.moveToNB(95, NaN, -95);
    robot3.moveToNB(95, NaN, -95);
    robot4.moveToNB(-95, NaN, 95);
    robot5.moveToNB(-110, NaN, 110);
```

move all five Linkbot-Is synchronously to move the body to position 2 shown in Figure 16.4. The head opens its claw by rotating joints 1 and 3 of `robot1` by -10 degrees and -30 degrees respectively. The front half of the body moves by rotating `robot2` and `robot3` to an absolute joint angle position of 95 degrees. The back half of the body moves by rotating `robot4` and `robot5` to absolute joint angle positions of -95 and -110 respectively. Then **moveWait()** is called on all Linkbots to pause the program until all parts of the snake have finished their motions. The next lines

```
    robot3.moveToNB(85, NaN, -85);
    robot4.moveToNB(-100, NaN, 100);
    robot5.moveToNB(-85, NaN, 85);
```

set the tail back down by synchronously rotating the joints of `robot3`, `robot4`, and `robot5` to absolute joint angle positions of 85 degrees, -100 degrees, and -85 degrees respectively. The snake is now in position 3, shown in Figure 16.4. The joint rotations are small, but they help position the snake for the final movement back to position 1. The function **moveWait()** is called on all Linkbots to pause the program until all parts of the snake have finished their motions for this position. Finally, all five Linkbots are reset to zero simultaneously. This results in the snake moving its body forward into a flat position. The function **moveWait()** is called on all Linkbot-Is one final time, to pause the program until the final motion is finished.

E Do Exercises 3 and 4 on page 318.

16.1.1 Summary

1. Perform motions in steps, using either the member function **delaySeconds**() or the member function **moveWait()** to pause the program between each step.
2. Use the blocking version of a **CLinkbotI** member function for the last Linkbot-I in a configuration, for each step in a series of motions. This can reduce the program size when controlling multiple connected Linkbots.

16.1.2 Terminology

connected Linkbots-Is, movement in steps, delay between steps.

16.1.3 Exercises

1. Run the program `fourwheeldrive.ch` in Program 16.1 with two Linkbot-Is connected as shown in Figure 16.1. Modify the program `fourwheeldrive.ch` as the program `fourwheeldrive2.ch` by adding the following statements for turning Linkbots left and right.

```
/* turn left at the same time */
robot1.turnLeftNB(90, radius, trackwidth);
robot2.turnLeftNB(90, radius, trackwidth);
robot1.moveWait();
robot2.moveWait();
/* turn right for robot1, then robot2. */
robot1.turnRight(90, radius, trackwidth);
robot2.turnRight(90, radius, trackwidth);
```

2. Modify the program `lift.ch` in Program 16.2 as a program `lift2.ch` by changing 45 to 60 and -45 to -60. Run this new lifting program.

3. Modify the program `snake.ch` in Program 16.3 as the program `snake2.ch` to move the snake forward continuously using a **while** loop.

4. Write a program `snake3.ch` with five Linkbot-Is for the snake motion shown in Figure 16.3. The program shall begin with the following comments on how the snake robot is configured.

```
/* File: snake3.ch
 * This program uses 5 Linkbot-Is for snake motion.
 * The linkbots are connected by bridges and simple connectors.
 * The robot5 has a claw attached as a mouth. The robot1 has a caster on
 * the back as a tail. The rubber-bands on the tail provides the needed friction.
 *             Top View
 *
 *        -----       -----
 *      +--+ +--+   +--+ +--+   +--+
 *   _| | 1| | 2| | | 3| | 4| | | 5|  <- Claw
 *      +--+ +--+   +--+ +--+   +--+
 *    ^   -----       -----
 *    |     ^               ^
 * Caster   |               |
 *        Bridge          Simple
 *       Connector       Connector
 */
```

16.2 Control Multiple Connected Linkbot-Ls

In Section 16.1 we learned how to control multiple connected Linkbot-Is. In this section we will learn how to write programs to control multiple connected Linkbot-Ls. The following program demonstrates how to control two connected Linkbot-Ls to stand up and lay back down again. To assemble this configuration, attach a square faceplate with a SnapConnection to joint 2 of `robot1` and `robot2` or fasten a square faceplate using a Phillips-head screwdriver. When the connected Linkbot-Ls are eventually in the standing position, this faceplate will be the base on which they stand. Then connect joint 1 on each Linkbot-L using a bridge connector. Then place the connected Linkbot-Ls in position 1, shown in Figure 16.5.

(a) position 1 (b) position 2 (c) position 3 (d) position 4 (e) position 5

Figure 16.5: Standing and unstanding motions with two connected Linkbot-Ls and a faceplate.

```
/* File: stand.ch
   Stand and unstand two linked Linkbot-Ls.
   Joint 2 of robot1 is connected to a square faceplate, placed on the ground.
   Joint 1 of two robots are connected by a bridge connector.

  Top View in its initial configuration

   robot1 robot2
   |----| |----|
  2|    | |    | 2
   |----| |----|
      1-----1
         ^
         |
   Bridge Connector


  Front View in its standing position

            +--+
   Joint1 ->||  |<- robot2
  Bridge    |+--+
connector ->|+--+
   Joint1 ->||  |<- robot1
            +--+
           ------
             ^
             |
        Joint 2 with
    Square faceplate attached

*/
#include <linkbot.h>
CLinkbotL robot1, robot2;

/* move to the zero position at the same time. */
robot1.resetToZeroNB();
robot2.resetToZero();

/* stand */
robot1.moveTo(90, 0, NaN);
robot2.moveTo(-60, 0, NaN);
robot1.moveTo(90, 45, NaN);
robot1.moveTo(-5, 45, NaN);

/* rotate two face plates at the same time */
robot1.moveJointNB(JOINT2, 360);
```

```
robot2.moveJoint(JOINT2, -360);

/* unstand */
robot1.resetToZeroNB();
robot2.resetToZero();

robot1.moveTo(90, 0, NaN);
robot1.moveTo(0, 0, NaN);
```

Program 16.4: Controlling two Linkbot-Ls for standing and unstanding motions.

Most of the **CLinkbotI** member functions are also available for the **CLinkbotL** class. The main difference is that the Linkbot-L uses joints 1 and 2 instead of joints 1 and 3. Recall, however, that only the Linkbot-I can be treated as a two-wheel vehicle. Therefore member functions such as **driveAngle()**, **turnLeft()**, **turnRight()**, as well as their non-blocking counterparts can only be used for the **CLinkbotI** class. In Program 16.4, the line

```
    robot1.moveTo(90, 0, NaN);
```

moves joint 1 to the absolute joint angle position of 90 degrees using the **CLinkbotL** version of the member function **moveTo()**. Since we do not want joint 2 to move just yet, the second argument has a value of zero. Since joint 3 of a Linkbot-L does not move, the argument has the value NaN. At this point the Linkbot-Ls will be in position 2, as shown in Figure 16.5. The following line

```
    robot2.moveTo(-60, 0, NaN);
```

moves joint 1 of `robot2` by -60 degrees. We do not yet want to move joint 2 of `robot2` either, so the second argument is given a value of zero. The Linkbot-Ls will be in position 3, as shown in Figure 16.5. The next line

```
    robot1.moveTo(90, 45, NaN);
```

will rotate the faceplate on joint 2 of `robot1` by 45 degrees while keeping joint 1 in its absolute joint angle position of 90 degrees. The connected Linkbot-Ls will now be in position 4, as shown in Figure 16.5. Then the line

```
    robot1.moveTo(-5, 45, NaN);
```

keeps the faceplate on joint 2 of `robot1` in its absolute joint angle position of 45 degrees. At the same time joint 1 of `robot1` rotates to the absolute joint angle position of -5 degrees. This will move `robot1` into a vertical position, holding `robot2` up in the air. The Linkbot-Ls are now in position 5, as shown in Figure 16.5. The next two lines

```
    robot1.moveJointNB(JOINT2, 360);
    robot2.moveJoint(JOINT2, -360);
```

rotate faceplates on joint 2 of both `robot1` and `robot2` simultaneously to the absolute joint angle positions of 360 degrees and -360 degrees respectively. Since the **CLinkbotL** version of the member functions **moveJointNB()** and **moveJoint()** are used, the enumerated value **JOINT2** is the appropriate value for the first argument in each statement. To make the connected Linkbot-Ls lay back down, joints 1 and 2 of both Linkbots are reset to the zero position. The **CLinkbotL** member function **resetToZeroNB()** is used for `robot1` in this case to allow both Linkbot-Ls to reset simultaneously. Finally, the line

```
    robot1.moveTo(90, 0, NaN);
```

will rotate joint 1 of `robot1` by 90 degrees, folding the connected Linkbot-Ls forward toward the ground. And the last line

```
    robot1.moveTo(0, 0, NaN);
```

will lay the connected Linkbots back down into position 1.

Do Exercises 1 and 2 on page 332.

Just as we can make two connected Linkbot-Ls stand up from a horizontal position, we can also make two connected Linkbot-Ls bow from a standing position. The configuration needed for Program 16.5 is the same one used for Program 16.4. Before running the program, however, the connected Linkbots should be placed in a standing position, with the square faceplate attached to joint 2 of `robot1` placed on the ground. The various positions for a bowing motion can be seen in Figure 16.6.

(a) position 1 (b) position 2 (c) position 3

Figure 16.6: Bowing motion with two connected Linkbot-Ls and a faceplate.

```
/* File: bow.ch
   Bow two linked Linkbot-Ls.
   Joint 2 of robot1 is connected to a square faceplate, placed on the ground.
   Joint 1 of two robots are connected by a bridge connector.

  Top View in its initial configuration
             +--+
   Joint1 ->||  |<- robot2
  Bridge    |+--+
connector ->|+--+
   Joint1 ->||  |<- robot1
             +--+
            ------
              ^
              |
        Joint 2 with
    Square faceplate attached
*/

#include <linkbot.h>
CLinkbotL robot1, robot2;

/* move to the zero position at the same time. */
robot1.resetToZeroNB();
robot2.resetToZero();

/* bow */
robot1.moveTo(0, 45, NaN);
robot2.moveToNB(90, 0, NaN);
robot1.moveTo(-45, 45, NaN);

/* back to the original position */
robot1.resetToZeroNB();
robot2.resetToZero();
```

Program 16.5: Controlling two Linkbot-Ls for bowing motion.

The line

```
robot1.moveTo(0, 45, NaN);
```

will rotate the faceplate attached to joint 2 of `robot1` by 45 degrees. This will partially turn the standing Linkbots right, as shown in position 2 of Figure 16.6. The next line

```
robot2.moveToNB(90, 0, NaN);
```

rotates joint 1 of `robot2` by 90 degrees. This will cause `robot2` to fold forward. Then the statement

```
robot1.moveTo(-45, 45, NaN);
```

holds the position of joint 2 of `robot1` in its absolute joint angle position of 45 degrees, while rotating joint 1 forward by 45 degrees. The connected Linkbot-Ls should now be in position 3 as shown in Figure 16.6. The final two lines of the program simultaneously reset joints 1 and 2 of both Linkbot-Ls to the zero positions. This will cause the bowing linkbots to turn and stand back up in the original position.

Do Exercise 3 on page 332.

We can also make two connected Linkbot-Ls crawl like an inchworm. The configuration needed for Program 16.6 is the same as the one used for Program 16.5. Be sure, however, to lay the connected Linkbot-Ls flat as shown in Figure 16.7 before running Program 16.6.

(a) original position (b) Move robot 1 (c) Move robot 2 (d) Move robot 1 (e) original position

Figure 16.7: The positions for an inchworm moving left.

```
/* File: inchworm.ch
 *
 *     Top View
 *
 *  robot1 robot2
 *  |----| |----|
 * 2|    | |    | 2
 *  |----| |----|
 *     1-----1
 *        ^
 *        |
 * Bridge Connector
 *
 * Joint 2 of each robot is connected with a square faceplate.
 * Joint 1 of two robots are connected by a bridge connector.
 */

#include <linkbot.h>

CLinkbotL robot1, robot2;

robot1.resetToZeroNB();
robot2.resetToZero();

/* inchworm left */
robot1.moveJointTo(JOINT1, -45);
robot2.moveJointTo(JOINT1, 45);
robot1.moveJointTo(JOINT1, 0);
robot2.moveJointTo(JOINT1, 0);

/* inchworm right */
robot2.moveJointTo(JOINT1, 45);
robot1.moveJointTo(JOINT1, -45);
robot2.moveJointTo(JOINT1, 0);
robot1.moveJointTo(JOINT1, 0);
```

Program 16.6: Controlling two connected Linkbot-Ls in an inchworm motion.

Program 16.6 first moves the inchworm left, as in Figure 16.7. Then it moves the inchworm right, in a manner similar to the movement left. The line

```
    robot1.moveJointTo(JOINT1, -45);
```

rotates joint 1 of `robot1` by -45 degrees. This begins the compression cycle of the worm movement. When this motion completes, the inchworm will be in the position shown in Figure 16.7 part (b). The next line

```
    robot2.moveJointTo(JOINT1, 45);
```

moves joint 1 of `robot2` by 45 degrees. After this movement completes, this inchworm is now completely compressed as in Figure 16.7 part (c). The following lines

```
    robot1.moveJointTo(JOINT1, 0);
    robot2.moveJointTo(JOINT1, 0);
```

stretch the inchworm to the left back into a flat position. First, joint 1 of `robot1` will move back to the zero position. The inchworm is now in the position shown in Figure 16.7 part (d). Then joint 1 of `robot2` will move back to the zero position. The inchworm is now flat, as shown in Figure 16.7 part (e). Next, the

inchworm will inch to the right. This movement will be a mirror image of movement left shown in Figure 16.7. The lines

```
robot2.moveJointTo(JOINT1, 45);
robot1.moveJointTo(JOINT1, -45);
```

compress the inchworm. This time joint 1 of `robot2` moves first, by 45 degrees, starting the compression cycle. Then `robot1` follows, moving its joint 1 by -45 degrees to complete the compression. Finally, the lines

```
robot2.moveJointTo(JOINT1, 0);
robot1.moveJointTo(JOINT1, 0);
```

stretch the inchworm to the right into a flat position. First joint 1 of `robot2` moves back to the zero position, then joint 1 of `robot1` moves back into the zero position.

Do Exercise 4 on page 333.

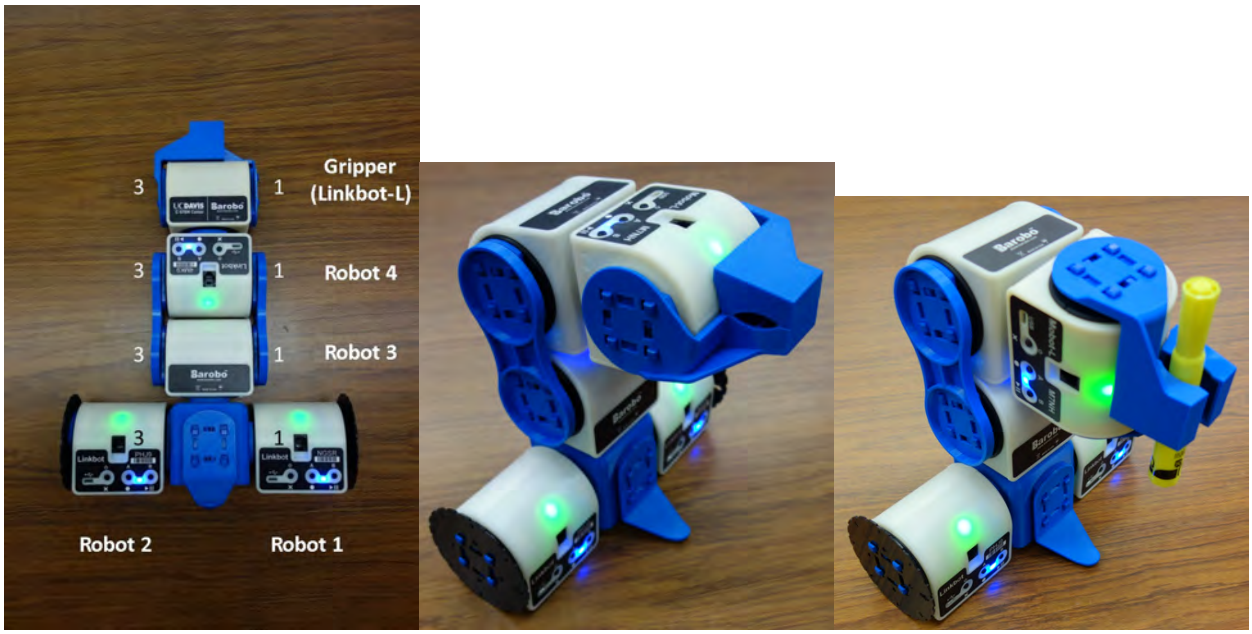
Up until this point we have learned how to control multiple connected Linkbots of only one kind. But it is also possible to control configurations that include both Linkbot-Is and Linkbot-Ls. The program, `explorer.ch`, controls a configuration consisting of four Linkbot-Is and one Linkbot-L. Before assembling an explorer, make sure that zero positions for all joint angles of each Linkbot are calibrated based on the instructions in section 2.1.4.

To assemble the explorer configuration, connect joints 1 and 3 of `robot3` and `robot4` with bridge connectors. This will be the arm of the explorer. Then connect joint 2 of `robot3` to the top of a cubic connector. The open end of the cubic connector should be on the bottom. Joint 1 of `robot1` should be connected to the right side of the cubic connector, and *a 4-inch diameter wheel* should be connected to joint 3 of this Linkbot-I. Connect joint 3 of `robot2` to the left side of the cubic connector, and attach *a 4-inch diameter wheel* to joint 1 of this Linkbot-I. `robot1` and `robot2` will be the wheels of the explorer. Then connect a claw to the Linkbot-L, with the top half of the claw attached to joint 1 and the bottom half attached to joint 3 (the non-moving joint). This will be the `gripper` of the explorer. Connect the `gripper` to the arm by connecting joint 2 of the `gripper` with joint 2 of `robot4` using a simple connector. Remember that joint 2 on the gripper is a moving joint, while joint 2 on `robot4` is a non-moving joint. Finally, place a caster on the front of the cubic connector in order to stabilize the explorer. Place a highlighter pen standing upright on the ground in front of the explorer. The explorer will pick up this pen when the program is run.

Before running Program 16.7, the explorer should be set in the initial position shown in Figure 16.8. In this starting position:

1. `robot3` has joint 1 positioned at -12 degrees and joint 3 positioned at 12 degrees.
2. `robot4` has joint 1 positioned at -90 degrees and joint 3 positioned at 90 degrees.
3. All other Linkbots have their joints initially set in the zero position.

Wheels with *4-inch diameter* are recommended to ensure that the explorer moves smoothly.



(a) Joint locations (b) Initial position (c) Final position

Figure 16.8: An explorer with four Linkbot-Is and one Linkbot-L picking up a highlighter pen.

```
/* File: explorer.ch
 * An explorer using four Linkbot-Is and one Linkbot-L for a gripper to pick up
 * a highlighter pen.
 * 1) Diameters of two wheels are of 4 inches.
 * 2) Numbers 1, 2 and 3 indicate joints.
 * 3) 1 caster is attached at the front of the cubic connector.
 * 4) Before running the program explorer.ch, the explorer should be set
 *    in the initial position shown in explorer.jpg,
 *    In this starting position:
 *      a) robot3 has joint 1 positioned at -12 degrees and
 *         joint 3 positioned at 12 degrees.
 *      b) robot4 has joint 1 positioned at -90 degrees and joint 3 positioned
 *         at 90 degrees.
 *      c) All other robots have their joints initially set in the zero position.
 * 5) When mounting the gripper, make sure joint 1 is in zero position
 *    if the gripper is closed
 * 6) Front View in its initial configuration
 *                            |--| gripper
 * gripper attached here -> |__|
 *                            --  <- simple connector
 *                          3|--|1 robot4
 *                          ||__||
 *     bridge connector -> |     | <- bridge connector
 *                          ||--||
 *                          3|__|1 robot3
 *                      1|--|----|--|3
 *           robot2  -> |__||   ||__| <- robot1
 *                              ^
 *                              |
 *                     cubic connector
 *
 */
```

```
#include <linkbot.h>
CLinkbotI robot1, robot2, robot3, robot4;
CLinkbotL gripper;

robot1.holdJoint(JOINT1);
robot2.holdJoint(JOINT3);

/* Initial position */
robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.moveToNB(-12, NaN, 12);
robot4.moveToNB(-90, NaN, 90);
gripper.resetToZeroNB();
robot1.moveWait();
robot2.moveWait();
robot3.moveWait();
robot4.moveWait();
gripper.moveWait();

/* move forward */
robot1.moveNB(0, NaN, 360);
robot2.moveNB(-360, NaN, 0);
robot1.moveWait();
robot2.moveWait();

/* turn left */
robot1.moveNB(0, NaN, -218);
robot2.moveNB(-218, NaN, 0);
robot1.moveWait();
robot2.moveWait();

/* move forward */
robot1.moveNB(0, NaN, 360);
robot2.moveNB(-360, NaN, 0);
robot1.moveWait();
robot2.moveWait();

/* Open the gripper by 60 and joint2 to 90 */
gripper.openGripper(60);
gripper.moveJointTo(JOINT2, 90);

/* arm ready */
robot3.moveToNB(70, NaN, -70);
robot4.moveToNB(-10, NaN, 10);
robot3.moveWait();
robot4.moveWait();

/* Close the gripper to grab an object */
gripper.closeGripper();

/* lift the arm */
robot3.setJointSafetyAngle(50);
robot3.moveToNB(-12, NaN, 12);
robot4.moveToNB(-90, NaN, 90);
robot3.moveWait();
robot4.moveWait();

/* move backward */
```

```
robot1.moveNB(0, NaN, -360);
robot2.moveNB(360, NaN, 0);
robot1.moveWait();
robot2.moveWait();

/* turn right */
robot1.moveNB(0, NaN, 195);
robot2.moveNB(195, NaN, 0);
robot1.moveWait();
robot2.moveWait();

/* move backward */
robot1.moveNB(0, NaN, -360);
robot2.moveNB(360, NaN, 0);
robot1.moveWait();
robot2.moveWait();

robot3.holdJointsAtExit();   // robot3 holds joints
gripper.holdJointsAtExit();  // gripper holds the object
```

Program 16.7: Controlling an explorer with four Linkbot-Is and one Linkbot-L.

You can hold a joint angle of a Linkbot by calling the function **holdJoint()**. The general syntax of this function is

```
robot.holdJoint(id);
```

The argument `id` specifies the joint that you wish to hold.

After all Linkbots have connected, the lines

```
robot1.holdJoint(JOINT1);
robot2.holdJoint(JOINT3);
```

hold the movement of joint 1 of `robot1` and joint 3 of `robot2`. These are the joints attaching the wheels to the cubic connector. Since we do not want these particular joints to move, they are held. Then the lines

```
robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.moveToNB(-12, NaN, 12);
robot4.moveToNB(-90, NaN, 90);
gripper.resetToZeroNB();
```

make sure that all joints of all Linkbots are set in the desired positions for the initial configuration. `robot1`, `robot2`, and `gripper` are all reset to the zero position simultaneously. Since we want `robot3` and `robot4` to start in the initial position shown in Figure 16.8, the function **moveToNB()** is used instead of **resetToZeroNB()** for these two Linkbot-Is. The function **moveWait()** is called on all five Linkbots to pause the program until all joints are finished setting to the initial configuration. Then the lines

```
/* move forward */
robot1.moveNB(0, NaN, 360);
robot2.moveNB(-360, NaN, 0);
robot1.moveWait();
robot2.moveWait();
```

simultaneously move the wheels of the explorer forward by 360 degrees. Since the inner joints don't move, the arguments for these joints are assigned an angle value of zero. The function **moveWait()** is used on `robot1` and `robot2` to pause the program until the explorer has finished moving forward. Then the next statements

```
/* turn left */
```

```
robot1.moveNB(0, NaN, -218);
robot2.moveNB(-218, NaN, 0);
robot1.moveWait();
robot2.moveWait();
```

cause the explorer to make a 90-degree left turn. Then, the explorer moves forward again.

The statements

```
/* Open the gripper by 60 and joint2 to 90 */
gripper.openGripper(60);
gripper.moveJointTo(JOINT2, 90);
```

open joint 1 mounted with a gripper by 60 degrees and joint 2 to 90 degrees. The syntax of the member function **openGripper**() is as follows.

```
robot.openGripper(angle);
```

The function opens joint 1 mounted with a gripper by an angle in degrees specified in its argument. When mounting the gripper using a Linkbot-L, make sure joint 1 is in zero position if the gripper is closed. When using a Linkbot-I as the gripper, joints 1 and 3 should be in zero positions if the gripper is closed.

The statements

```
/* arm ready */
robot3.moveToNB(70, NaN, -70);
robot4.moveToNB(-10, NaN, 10);
robot3.moveWait();
robot4.moveWait();
```

prepare the arm of the explorer to grab the highlighter pen. The joints of `robot4` rotate 10 degrees backward, while the joints of `robot3` rotate 70 degrees forward. The overall effect will be the arm bending forward toward the highlighter pen, while the `gripper` turns sideways and opens its claw around the pen. The member function **moveWait()** is called on `gripper`, `robot3`, and `robot4` to pause the program until all Linkbots have finished the current motion.

The statement

```
/* Close the gripper to grab an object */
gripper.closeGripper();
```

closes the claw of the `gripper` around the highlighter pen. The syntax of the member function **closeGripper**() is as follows.

```
robot.closeGripper();
```

The function closes the gripper.

The lines

```
/* lift the arm */
robot3.setJointSafetyAngle(50);
robot3.moveToNB(-12, NaN, 12);
robot4.moveToNB(-90, NaN, 90);
robot3.moveWait();
robot4.moveWait();
```

rotate joints 1 and 3 of `robot3` and `robot4` of the explorer arm back into the initial position. The program pauses until this motion completes.

The Linkbot is equipped with a safety feature to protect itself and its surrounding environment. When a motor deviates by a certain amount from its expected value, the Linkbot will shut off all power to the motor, in case it has hit an obstacle, or for any other reason. The amount of deviation required to trigger the safety protocol is the joint safety angle which can be set using the member function **setJointSafetyAngle**(). The general syntax of this function is

```
robot.setJointSafetyAngle(angle);
```

The argument `angle` is the safety angle. The default safety angle is 10 degrees. Higher values indicate "less safe" behavior of the Linkbot because the Linkbot will not engage safety protocols until the joint has deviated by a greater amount. Since `robot3` will carry a heavy weight at this cantilever position, the safety angle is set to 50 degrees.

The lines

```
/* move backward */
robot1.moveNB(0, NaN, -360);
robot2.moveNB(360, NaN, 0);
robot1.moveWait();
robot2.moveWait();
```

move the wheels of the explorer backward by 360 degrees.

The lines

```
/* turn right */
robot1.moveNB(0, NaN, 195);
robot2.moveNB(195, NaN, 0);
robot1.moveWait();
robot2.moveWait();
```

turn the wheels of the explorer right by 90 degrees. The program pauses again until `robot1` and `robot2` complete this turn.

The explorer moves back again. The explorer should be in its original position, as in Figure 16.8, but now holding the highlighter pen in its claw. The program pauses one last time to allow this backward motion to finish.

Finally, the statements

```
robot3.holdJointsAtExit();   // robot3 holds joints
gripper.holdJointsAtExit();  // gripper holds the object
```

make sure the `robot3` holds all joints and the `gripper` holds the object even after the program finishes its execution.

Program 16.7 can also be executed in RoboSim as one of Preconfigured Linkbot Configurations in explorer.ch.

Do Exercises 5, 6, 7, 8, 9, and 10 on page 334.

Another way to control connected Linkbot-Ls is in the four-wheel omnidrive configuration. This configuration is shown in Figure 16.9. It uses four Linkbot-Ls, one for each wheel. To assemble this configuration, connect joint 2 of each Linkbot-L to the four corners of an H-connector. Then connect a wheel to joint 1 of each Linkbot-L. If you look at the omnidrive from the top, `robot4` will be on the front left corner, `robot2` will be on the front right corner, `robot3` will be on the back left corner, and `robot1` will be on the back right corner.

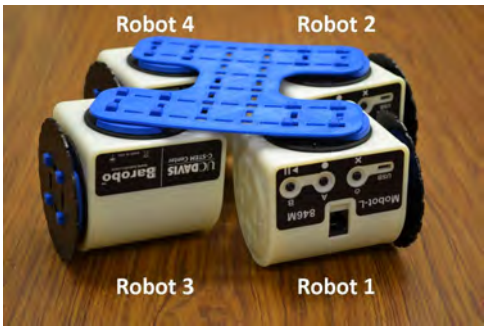


Figure 16.9: An omnidrive with four Linkbot-Ls.

```
/* File: omnidrive.ch
 * This program uses 4 Linkbot-Ls to make a four-wheel omnidrive robot
 * where each wheel can turn.
 * H connector connects joint 2 of each Linkbot-L.
 * Wheels are attached to joint 1 of each Linkbot-L.
 *
 *       Top  View
 *
 *     robot4    robot2
 *     |---|     |---|
 *    1| 2-+-----+-2 |1
 *     |---|  |  |---|
 *            |
 *            | <---------- connector
 *     |---|  |  |---|
 *    1| 2-+-----+-2 |1
 *     |---|     |---|
 *    robot3     robot1
 *
*/
#include <linkbot.h>
CLinkbotL robot1, robot2, robot3, robot4;
double angle = 250;

robot1.resetToZeroNB();
robot2.resetToZeroNB();
robot3.resetToZeroNB();
robot4.resetToZero();

/* Move Forward */
robot1.moveNB(angle,0,NaN);
robot2.moveNB(angle,0,NaN);
robot3.moveNB(-angle,0,NaN);
robot4.move(-angle,0,NaN);

/* Move Backward */
robot1.moveNB(-angle,0,NaN);
robot2.moveNB(-angle,0,NaN);
robot3.moveNB(angle,0,NaN);
robot4.move(angle,0,NaN);

/* Sharp Turn 90 */
robot1.moveJointToNB(JOINT2, 90);
robot2.moveJointToNB(JOINT2, -90);
```

```
robot3.moveJointToNB(JOINT2, -90);
robot4.moveJointTo(JOINT2, 90);

/* Move Forward */
robot1.moveNB(angle,0,NaN);
robot2.moveNB(-angle,0,NaN);
robot3.moveNB(angle,0,NaN);
robot4.move(-angle,0,NaN);
```

Program 16.8: Controlling an omnidrive with four Linkbot-Ls.

Running Program 16.8 will roll the four wheels simultaneously to move the omnidrive forward and backward. The line

```
    double angle = 250;
```

declares a variable `angle` of **double** type and assigns it the value 250. This variable is used to control each of the four wheels in forward and backward motions of 250 degrees. Then all Linkbot-Ls connect and reset their joints to the zero position. Then the lines

```
    robot1.moveNB(angle,0,NaN);
    robot2.moveNB(angle,0,NaN);
    robot3.moveNB(-angle,0,NaN);
    robot4.move(-angle,0,NaN);
```

simultaneously rotate `robot1` and `robot2` by 250 degrees and rotate `robot3` and `robot4` by -250 degrees. Since joint 1 of both `robot1` and `robot2` are on the left side of the omnidrive, they rotate by a positive angle value according to the right-hand rule. Since joint 1 of `robot3` and `robot4` are on the right side of the omnidrive, they rotate by a negative angle value according to the right-hand rule. The combined motion of all four Linkbot-Ls moves the omnidrive forward by 250 degrees. A blocking motion function is used for `robot4` so that the program will wait until all four wheels have finished moving forward. The next statements

```
    robot1.moveNB(-angle,0,NaN);
    robot2.moveNB(-angle,0,NaN);
    robot3.moveNB(angle,0,NaN);
    robot4.move(angle,0,NaN);
```

simultaneously move all four wheels of the omnidrive backward by 250 degrees. This time the angle values for joint 1 of `robot1` and `robot2` are negative, and the angle values for joint 1 of `robot3` and `robot4` are positive. The next lines

```
    robot1.moveJointToNB(JOINT2, 90);
    robot2.moveJointToNB(JOINT2, -90);
    robot3.moveJointToNB(JOINT2, -90);
    robot4.moveJointTo(JOINT2, 90);
```

simultaneously move joint 2 of `robot1` and `robot4` to an absolute joint angle position of 90 degrees. At the same time, joint 2 of `robot2` and `robot3` move to an absolute joint angle position of -90 degrees. The cumulative effect of these motions is to turn all four wheels of the omnidrive in a sharp left turn. The last four lines of the program

```
    robot1.moveNB(angle,0,NaN);
    robot2.moveNB(-angle,0,NaN);
    robot3.moveNB(angle,0,NaN);
    robot4.move(-angle,0,NaN);
```

move the omnidrive forward by 250 degrees. Since joint 1 of `robot1` and `robot3` are now on the left side of the omnidrive, they rotate by a positive angle value in order to move forward. `robot2` and `robot4` are

now on the right side of the omnidrive, so they need to rotate by a negative angle value in order to move the omnidrive forward.

Do Exercise 11 on page 336.

16.2.1 Summary

1. Call the **CLinkbotI** member function

   ```
   robot.holdJoint(id);
   ```

   to hold a joint of a Linkbot specified by its argument `id`.

2. Call the **CLinkbotI** member function

   ```
   robot.setJointSafetyAngle(angle);
   ```

   to set the joint safety angle.

3. Most of the blocking and non-blocking **CLinkbotI** member functions are also available for the **CLinkbotL** class.

16.2.2 Terminology

connected Linkbots-Ls.

16.2.3 Exercises

1. Write a program `stand2.ch` to control two connected Linkbot-Ls with the standing motion. Joint 2 of the 1st robot is connected with a square faceplate. Joint 1 of two robots are connected by a bridge connector as shown in the figure. The program will move joints for two robots in the following sequences. Move joint 1 of robot1 to -90 degrees, joint 1 of robot2 to 90 degrees, joint 2 of robot1 to 45 degrees, then joint 1 of robot1 to 0 degree.

2. Pose teach two linked Linkbot-Ls from the flat position to stand up as shown Figure 16.5 on page 319. Play the recorded motion. Save the motion in a program `teachstand.ch` and run the program in ChIDE.

3. Pose teach two linked Linkbot-Ls to bow as shown in Figure 16.6 on page 321. Play the recorded motion. Save the motion in a program `teachbow.ch` and run the program in ChIDE. Compare the program `teachbow.ch` with the program `bow.ch` in Program 16.5.

4. Write a program `inchworm2.ch` to control two Linkbot-Ls in an inchworm motion. Joint 2 of the the two robots are connected with a square faceplate. Joint 1 of the two robots are connected by a bridge connector. The program will move joint 1 of each robot in the following sequences. To move the inchworm left move joint 1 of robot1 to -50 degrees, joint 1 of robot2 to 40 degrees, joint 1 of robot1 to 0 degrees, and joint 1 of robot2 to 0 degrees. To move the inchworm right move joint 1 of robot2 to 40 degrees, joint 1 of robot1 to -50 degrees, joint 1 of robot2 to 0 degrees, and joint 2 of robot1 to 0 degrees. Joint 2 of robot1 and robot2 will not move.

5. Run the program `gripper.ch` using a Linkbot-L to pick up a highlighter pen and hackysack as shown in Figure 16.10. The gripper is opened by 90 degrees first, joint 2 rotates by 90 degrees, then close the gripper. Write a program `gripper2.ch` using a Linkbot-I to open the gripper by 90 degrees first, then close the gripper.

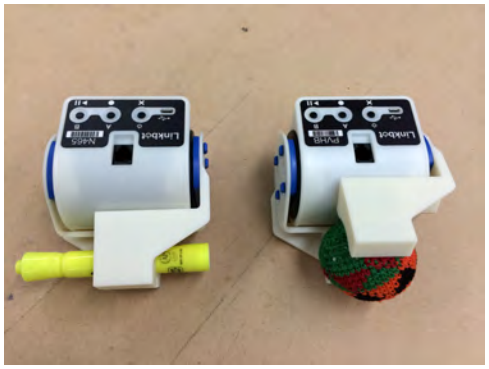


Figure 16.10: A Linkbot-I picks up a pen and hackysack.

6. The program `opengripper.ch` illustrates how the program `gripper.ch` in Exercise 6 can be implemented similarly without using the member functions **openGripper**() and **closeGripper**().

```
/* File: opengripper.ch
   Grip an object using a Linkbot-L.
   When mounting the gripper, make sure joint 1 is in zero position
   if the gripper is closed */
#include <linkbot.h>
CLinkbotL gripper;
double gripperAngleNew, gripperAngleOld;

gripper.resetToZero();

/* Open the gripper by moving joint1 to -90 and joint2 to 90 */
gripperAngleOld = 0;        // initialize the variable
gripperAngleNew = -90;
gripper.moveTo(gripperAngleNew, 90, NaN);

/* Close the gripper to grab an object */
while(abs(gripperAngleNew - gripperAngleOld) > 0.1) {
    printf("gripperAngleNew = %lf\n", gripperAngleNew);
    gripperAngleOld = gripperAngleNew;    // update the old position
    gripper.moveJointNB(JOINT1, 8); // move 8 degrees
    gripper.delaySeconds(1);              // closing for 1 second
    gripper.getJointAngle(JOINT1, gripperAngleNew); // get the new position
}
gripper.moveJointNB(JOINT1, 8); // try to move another 8 degrees
```

```
gripper.delaySeconds(1);          // closing for 1 second
gripper.holdJoint(JOINT1);        // hold the object
gripper.holdJointsAtExit();       // hold the object when the program ends
```

Program 16.9: Opening the gripper by 90 degrees, rotating joint 2 by 90 degrees, then closing the gripper using a Linkbot-L.

The statements

```
    gripperAngleOld = 0;          // initialize the variable
    gripperAngleNew = -90;
```

initialize the variable `gripperAngleOld` with zero and `gripperAngleNew` with -90.

The statement

```
    /* Open the gripper by moving joint1 to -90 and joint2 to 90 */
    gripper.moveTo(gripperAngleNew, 90, NaN);
```

rotates the gripper sideways by 90 degrees, while the claw rotates open by -90 degrees.

Then the next lines of the program

```
    /* Close the gripper to grab an object */
    while(abs(gripperAngleNew - gripperAngleOld) > 0.1) {
        gripperAngleOld = gripperAngleNew;     // update the old position
        gripper.moveJointNB(JOINT1, 8); // move 8 degrees
        gripper.delaySeconds(1);                // closing for 1 second
        gripper.getJointAngle(JOINT1, gripperAngleNew);// get the new position
    }
    gripper.moveJointNB(JOINT1, 8); // try to move another 8 degrees
    gripper.delaySeconds(1);               // closing for 1 second
    gripper.holdJoint(JOINT1);             // hold the object
```

closes the claw of the `gripper` around the highlighter pen. The code uses a **while** loop to repeatedly move the joint 1 of the gripper by 8 degrees till the absolute value for the difference between the two subsequent readings is smaller than 0.1 degree. The function **abs**() obtains the absolute value of its argument. The motion for joint 1 is accomplished by the member function **moveJointNB**() for 1 second by the member function **delaySeconds**(). Then, the joint angle is obtained by the member function **getJointAngle**(). For a firm hold on the pen, after the **while** loop, joint 1 for the gripper is moved again for 1 second and set to the hold state by the member function **holdJoint**().

(a) Run the program `opengripper.ch` to open and close a gripper as shown in Figure 16.10 to grab an object.

(b) Write a similar program `opengripper2.ch` using a Linkbot-I to grab an object. The program will open the gripper by 90 degrees using the following statements.

```
    /* Open the gripper by moving joint1 to -45 and joint3 to -45 */
    gripperAngleOld = 0;          // initialize the variable
    gripperAngleNew = -45;
    gripper.moveTo(gripperAngleNew, NaN, -45);
```

It then closes the gripper.

7. Run the program `explorer.ch` in Program 16.7 in both hardware and RoboSim (in Preconfigured Linkbot Configurations in explorer.ch).

8. Modify Program 16.7 as `explorer2.ch` to pick up a hackysack as shown in Figure 16.11. The code for getting the arm ready should be as follows.

```
/* arm ready */
robot3.moveToNB(50, NaN, -50);
robot4.moveToNB(-100, NaN, 100);
robot3.moveWait();
robot4.moveWait();
```

Run the program `explorer2.ch` in both hardware and RoboSim (in Preconfigured Linkbot Configurations in explorer.ch).

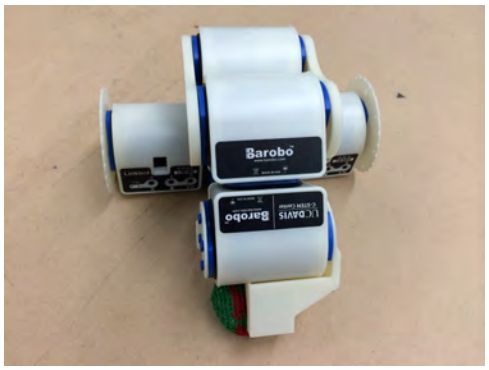
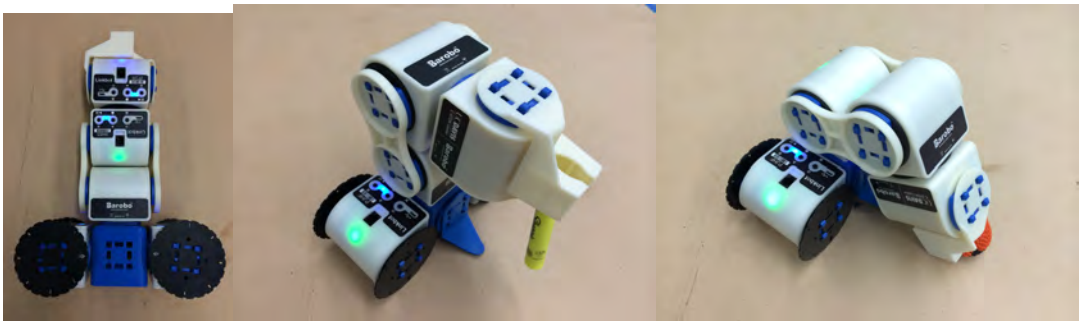


Figure 16.11: An explorer picks up a hackysack.

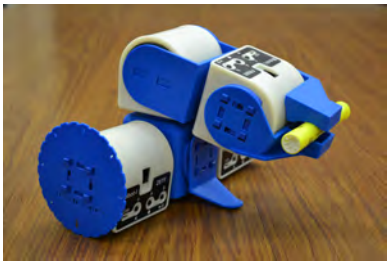
9. An explorer with a four-wheel drive is shown in Figure 16.12. Write a program `fourwheelexplorer.ch`, similar to Program 16.7. The program shall be able to pick up a highlighter pen and hackysack using such a four-wheel drive explorer. Run the program in both hardware and RoboSim (in Preconfigured Linkbot Configurations in fourwheelexplorer.ch).



(a) Initial position (b) Picking up a pen (c) Picking up a hackysack

Figure 16.12: An explorer with four Linkbot-Is and one Linkbot-L using a four-wheel drive.

10. Pose teach an explorer with four Linkbots as shown in the figure below to move towards an object and pick it up. Play the recorded motion. Save the motion in a program `teachexplorer.ch` and run the program in ChIDE.



11. Write a program `omnidrive2.ch` by appending the following code to the program `omnidrive.ch` in Program 16.8 an omnidrive with four Linkbot-Ls shown in Figure 16.9. Run the program in ChIDE in debug mode.

```
/* Move Backward */
robot1.moveNB(-angle,0,NaN);
robot2.moveNB(angle,0,NaN);
robot3.moveNB(-angle,0,NaN);
robot4.move(angle,0,NaN);

/* Sharp Turn Back */
robot1.moveJointToNB(JOINT2, 0);
robot2.moveJointToNB(JOINT2, 0);
robot3.moveJointToNB(JOINT2, 0);
robot4.moveJointTo(JOINT2, 0);

/* Sharp Turn 45 */
robot1.moveJointToNB(JOINT2, 45);
robot2.moveJointToNB(JOINT2, 45);
robot3.moveJointToNB(JOINT2, 45);
robot4.moveJointTo(JOINT2, 45);

/* Move Forward */
robot1.moveNB(angle,0,NaN);
robot2.moveNB(angle,0,NaN);
robot3.moveNB(-angle,0,NaN);
robot4.move(-angle,0,NaN);

/* Sharp Turn -45 */
robot1.moveJointToNB(JOINT2, -45);
robot2.moveJointToNB(JOINT2, -45);
robot3.moveJointToNB(JOINT2, -45);
robot4.moveJointTo(JOINT2, -45);

/* Move Forward */
robot1.moveNB(angle,0,NaN);
robot2.moveNB(angle,0,NaN);
robot3.moveNB(-angle,0,NaN);
robot4.move(-angle,0,NaN);

/* Sharp Turn Back */
robot1.moveJointToNB(JOINT2, 0);
robot2.moveJointToNB(JOINT2, 0);
robot3.moveJointToNB(JOINT2, 0);
robot4.moveJointTo(JOINT2, 0);
```

16.3 Control a Group of Connected Linkbots with Identical Movements

In Section 16.1 and Section 16.2, we learned how to control multiple connected Linkbots. In this section we will learn how to control a group of connected Linkbots in identical movements. Four Linkbot-Is can be connected in a four-bot drive configuration, which is similar to the omnidrive configuration from Program 16.8. Groups can be used to control the left and right wheels of the four-bot drive. Figure 16.13 shows the four-bot drive configuration. To assemble this configuration, connect joint 2 of each Linkbot-I to each corner of an H-connector. Attach wheels to joint 1 of each Linkbot-I on the left hand side, and attach wheels to joint 3 of each Linkbot-I on the right hand side.

Figure 16.13: A four-bot drive with four Linkbot-Is.

```
/* File: fourbotdrive.ch
 *
 *       Top View
 *
 *       |++| |++|
 *       |++| |++|
 *
 *       |++| |++|
 *       |++| |++|
 *
 */
#include <linkbot.h>
CLinkbotI robotLeftFront, robotLeftRear;
CLinkbotI robotRightFront, robotRightRear;
CLinkbotIGroup robotLeft, robotRight;

robotLeft.addRobot(robotLeftFront);
robotLeft.addRobot(robotLeftRear);
robotRight.addRobot(robotRightFront);
robotRight.addRobot(robotRightRear);

/* move forward */
robotLeft.moveNB(360, NaN, 0);
robotRight.moveNB(0, NaN, -360);
robotLeft.moveWait();
robotRight.moveWait();

/* move backward */
robotLeft.moveNB(-360, NaN, 0);
```

```
robotRight.moveNB(0, NaN, 360);
robotLeft.moveWait();
robotRight.moveWait();

/* turn right */
robotLeft.moveNB(360, NaN, 0);
robotRight.moveNB(0, NaN, 360);
robotLeft.moveWait();
robotRight.moveWait();

/* turn left */
robotLeft.moveNB(-360, NaN, 0);
robotRight.moveNB(0, NaN, -360);
robotLeft.moveWait();
robotRight.moveWait();
```

Program 16.10: Controlling four Linkbot-Is as a four-bot drive vehicle.

Program 16.10 controls the wheels on the left side of the four-bot drive using one group and the wheels on the right side using another group. The lines

```
    CLinkbotI robotLeftFront, robotLeftRear;
    CLinkbotI robotRightFront, robotRightRear;
```

declare variables to control each Linkbot-I. The variables `robotLeftFront`, `robotLeftRear`, `robotRightFront`, and `robotRightRear` control the Linkbot-Is in the corresponding positions of the four-bot configuration. The next line

```
    CLinkbotIGroup robotLeft, robotRight;
```

declares two variables, `robotLeft` and `robotRight`, to control the motions of Linkbot-Is on the left and right sides of the four-bot drive, respectively. The following lines

```
    robotLeft.addRobot(robotLeftFront);
    robotLeft.addRobot(robotLeftRear);
```

add the Linkbot-Is on the left side of the four-bot drive to the `robotLeft` group, one at a time. Then the lines

```
    robotRight.addRobot(robotRightFront);
    robotRight.addRobot(robotRightRear);
```

add the Linkbot-Is on the right side of the four-bot drive to the `robotRight` group. The non-blocking member function **resetToZeroNB()** is used to reset the joints of all Linkbot-Is to the zero position simultaneously. The next lines

```
    robotLeft.moveNB(360, NaN, 0);
    robotRight.moveNB(0, NaN, -360);
```

move the four-bot drive forward by 360 degrees. Joint 1 of each Linkbot-I in the `robotLeft` group is rotated by 360 degrees, and joint 3 of each Linkbot-I in the `robotRight` group is rotated by -360 degrees. Since the non-blocking **moveNB()** function is used for both groups, all four Linkbot-Is move simultaneously. Because no wheel is attached to joint 3 of each Linkbot-I in the `robotLeft` group, this joint is rotated by 0 degrees. Similarly, because no wheel is attached to joint 1 of each Linkbot-I in the `robotRight` group, this joint is also rotated by 0 degrees. The resulting forward movement of the four-bot drive is equivalent to the movement of one Linkbot-I using the following function call

```
    robot.move(360, NaN, -360);
```

The function **moveWait()** is called on each group to pause the program until the forward movement is finished. Then the lines

```
    robotLeft.moveNB(-360, NaN, 0);
    robotRight.moveNB(0, NaN, 360);
```

move the `robotLeft` and `robotRight` groups simultaneously, with the result that the four-bot drive backward by 360 degrees. The movement of the four-bot drive in this case is equivalent to the movement of one Linkbot-I using the function call

```
    robot.move(-360, NaN, 360);
```

Then the function **moveWait()** is called on each group to pause the program until the backward movement is finished. The following lines

```
    robotLeft.moveNB(360, NaN, 0);
    robotRight.moveNB(0, NaN, 360);
```

cause joint 1 of each Linkbot-I in the `robotLeft` group to rotate 360 degrees. This happens at the same time that joint 3 of each Linkbot-I in the `robotRight` group rotates 360 degrees. The resulting movements of the groups `robotLeft` and `robotRight` cause the four-bot drive to make a full right turn. The motion of the four-bot drive in this case is equivalent to the movement of one Linkbot-I using the function call

```
    robot.turnRight(360, radius, trackwidth);
```

where `radius` is the length of the wheel radius and `trackwidth` is the distance between the two wheels. The function **moveWait()** is again called on both groups to pause the program until the right turn is finished. Then the next statements

```
    robotLeft.moveNB(-360, NaN, 0);
    robotRight.moveNB(0, NaN, -360);
```

cause joint 1 of each Linkbot-I in the `robotLeft` group to rotate -360 degrees. At the same time, joint 3 of each Linkbot-I in the `robotRight` group also rotates -360 degrees. The resulting simultaneous movements of both groups cause the four-bot drive to make a full left turn. The motion of the four-bot drive is equivalent to the movement of one Linkbot-I using the function call

```
    robot.turnLeft(360, radius, trackwidth);
```

where `radius` is the length of the wheel radius and `trackwidth` is the distance between the two wheels. Finally, the function **moveWait()** is called on both groups to allow the left turn to complete before the program finishes execution.

Do Exercise 1 on page 342.

We can also make a group of connected Linkbot-Ls perform identical bowing motions. Figure 16.14 shows a group of connected Linkbot-Ls, assembled as described in Section 16.2. Be sure that `robot3` and `robot4` are assembled in the same manner as `robot1` and `robot2`. Also be sure to place the faceplates of `robot1` and `robot3` on the ground before running Program 16.11.

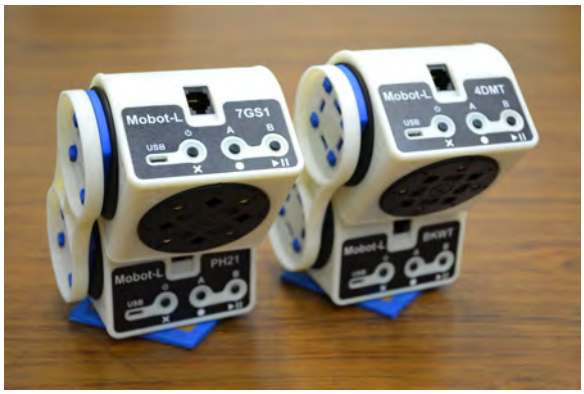


Figure 16.14: Two Linkbot-Ls bow at the same time.

```
/* File: groupbow.ch
   Bow a group of two, each with two linked Linkbot-Ls.
   Joint 2 of robot1 is connected to a square faceplate, placed on the ground.
   Joint 1 of robot1 and robot2 are connected by a bridge connector.
   rohot3 and robot4 are connected in the same manner as robot1 and robot2 */
#include <linkbot.h>
CLinkbotL robot1, robot2, robot3, robot4;
CLinkbotLGroup group1, group2;

/* add robot1 and robot3 to the group1
   add robot2 and robot4 to the group2 */
group1.addRobot(robot1);
group1.addRobot(robot3);
group2.addRobot(robot2);
group2.addRobot(robot4);

/* move to the zero position at the same time. */
group1.resetToZeroNB();
group2.resetToZero();

/* set speed ratio to 0.25 for slow motion */
group1.setJointSpeedRatios(0.25, 0.25, NaN);
group2.setJointSpeedRatios(0.25, 0.25, NaN);

/* bow */
group1.moveTo(0, 45, NaN);
group2.moveToNB(90, 0, NaN);
group1.moveTo(-45, 45, NaN);

/* back to the original position */
group1.resetToZeroNB();
group2.resetToZero();
```

Program 16.11: Bowing a group of Linkbot-Ls for a robot dance.

Program 16.11 controls groups of connected modules in bowing motions. A ***connected module*** refers to a single Linkbot that occupies a specific position in a configuration, such as the `gripper` of the explorer from Program 16.7. A ***group of connected modules*** refers to multiple corresponding Linkbots that occupy a specific position in multiple identical configurations. An example would be a group of grippers from

multiple explorers. A group of connected modules can be controlled using member functions of either the **CLinkbotIGroup** or **CLinkbotLGroup** classes. Like the **CLinkbotIGroup** class, the **CLinkbotLGroup** class is defined in the **linkbot.h** header file. The line

```
CLinkbotLGroup group1, group2;
```

declares two variables, `group1` and `group2`, of class **CLinkbotLGroup**. These will be used to control the groups of connected modules. Then the lines

```
group1.addRobot(robot1);
group1.addRobot(robot3);
```

add `robot1` and `robot3` to `group1` using the **CLinkbotLGroup** version of the member function **addRobot()**. The syntax is the same as the syntax for the **CLinkbotIGroup** version, which is listed in Section 7.1. Since `robot1` and `robot3` are the feet in each of the identical configurations, they will be controlled by the same group. The next lines

```
group2.addRobot(robot2);
group2.addRobot(robot4);
```

add `robot2` and `robot4` to `group2`. Since `robot2` and `robot4` are the heads in each of the identical configurations, they will be controlled by a different group than `robot1` and `robot3`. The next lines

```
group1.resetToZeroNB();
group2.resetToZero();
```

reset all joints of all Linkbot-Ls to the zero position. The **CLinkbotLGroup** versions of the member functions **resetToZero()** and **resetToZeroNB()** are used in this case, with syntax similar to that listed in Section 7.1. The next lines

```
group1.setJointSpeedRatios(0.25, 0.25, NaN);
group2.setJointSpeedRatios(0.25, 0.25, NaN);
```

set the speed ratios for all joints of all Linkbot-Ls in `group1` and `group2` to the value 0.25, for slow motion. The general syntax for the **CLinkbotLGroup** version of the member function **setJointSpeedRatios()** is similar to that listed in Section 10.7.4, with two differences. The first difference is that, because **setJointSpeedRatios()** is being used for Linkbot-Ls, the arguments will be slightly different. The first and second arguments will be joint speed ratio values, and the third argument will be NaN since joint 3 does not move. the second difference is that, because **setJointSpeedRatios()** is being used for groups, the joint speed ratios of multiple Linkbot-Ls will be set using a single function call. The next line

```
group1.moveTo(0, 45, NaN);
```

rotates the faceplates attached to joint 2 of the Linkbot-Ls in `group1` by 45 degrees. This partially turns the two standing configurations so that they are facing left. Since the **CLinkbotLGroup** version of **moveTo()** is used, multiple Linkbot-Ls are turned at the same time. Then the line

```
group2.moveToNB(90, 0, NaN);
```

rotates joint 1 of the Linkbot-Ls in `group2` by 90 degrees. This will cause `robot2` and `robot4`, the heads of the bowing configurations, to fold forward. Since the statement uses a non-blocking function, then the next line will execute simultaneously with it. This synchronous statement

```
group1.moveTo(-45, 45, NaN);
```

holds the position of joint 2 of the Linkbot-Ls in `group1` at the absolute joint angle position of 45 degrees, while rotating joint 1 of both `robot1` and `robot3` forward by 45 degrees. These motions of `group1` will be performed at the same time as the motions from the previous statement. After this line executes, then both configurations in the group will be bowing at the same time, as shown in Figure 16.14. Finally, the lines

```
group1.resetToZeroNB();
group2.resetToZero();
```

resets joints 1 and 2 of all Linkbot-Ls in both groups simultaneously to the zero position. Both of the identical configurations will now be standing up straight and facing forward as the program finishes execution.

Do Exercises 2 and 3 on page 343.

16.3.1 Summary

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

   ```
   #include <linkbot.h>
   CLinkbotIGroup group;
   ```

   for controlling a group of Linkbot-Is.

2. Include the header file **linkbot.h** and use the class **CLinkbotLGroup** to declare a variable `group` by the following two statements

   ```
   #include <linkbot.h>
   CLinkbotLGroup group;
   ```

   for controlling a group of Linkbot-Ls.

3. Call the **CLinkbotIGroup** member function

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

   to add a single Linkbot-I to a group.

4. Call the **CLinkbotLGroup** member function

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

   to add a single Linkbot-L to a group.

5. Use a group of connected modules to control multiple connected modules in identical motions.

6. Most of the blocking and non-blocking **CLinkbotIGroup** member functions are also available for the **CLinkbotLGroup** class.

16.3.2 Terminology

CLinkbotIGroup, **CLinkbotLGroup**, connected modules, group of connected modules.

16.3.3 Exercises

1. Write a program `fourbotdrive2.ch` to control groups of Linkbot-Is in a four-bot drive configuration. The program will move the four-bot drive in the following sequence. Move backward 360 degrees, turn left 720 degrees, move forward 360 degrees, and finally turn right 720 degrees.

2. Write a program `groupbow2.ch` for four Linkbots to bow simultaneously.

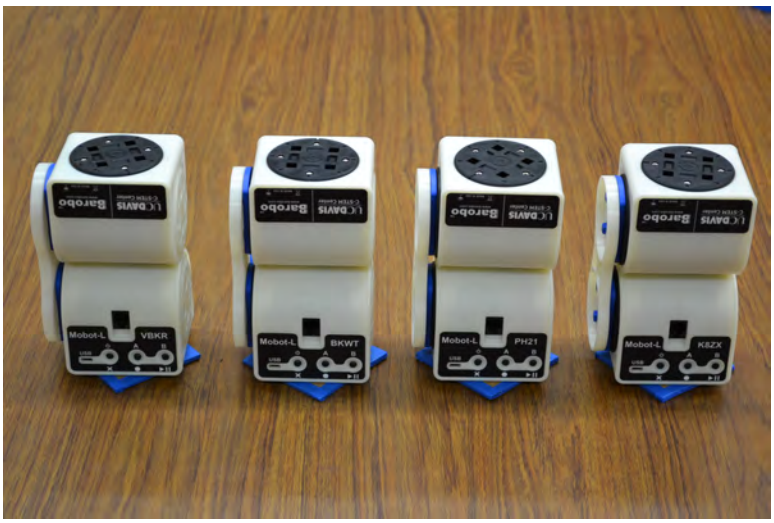


Figure 16.15: A group of Linkbots bow simultaneously.

3. Write a program `groupbow3.ch` for three Linkbots to bow in the following manner. The middle Linkbot bows first, delay for 0.5 second, then two other Linkbots bow at the same time. Treat the two other Linkbots as a group.

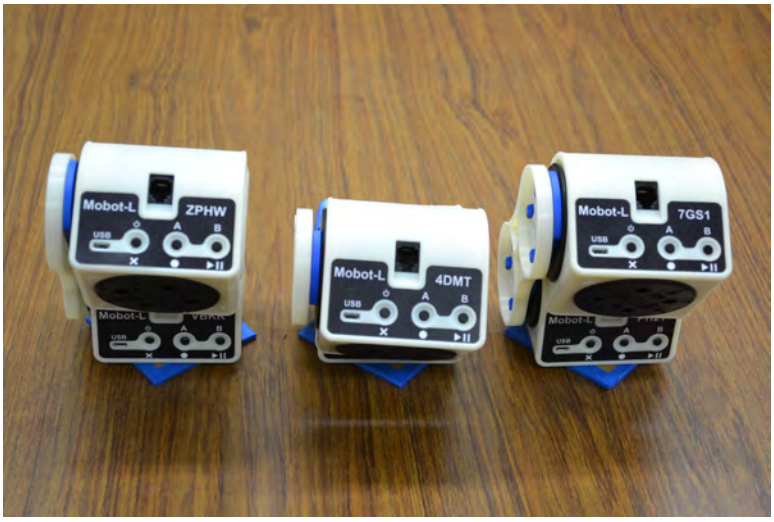


Figure 16.16: Three Linkbots bow together, with the middle one bowing first, followed by other two Linkbots.

APPENDIX A

Using Advanced Programming Features

This appendix lists a few sample programs using advanced programming features not covered in this book. You may learn more about these advanced features in the book *Learning Computer Programming in Ch for the Absolute Beginner*.

A.1 Make a Decision Using an if and else if Statements

The general syntax of an **if**, **else if**, and **else** statement is as follows:

```
if(expression1) {
   /* statements1; */
}
else if(expression2){
   /* statements2; */
}
else {
   /* statements3; */
}
```

The **if-else** statement is a common way to make decisions in a program. The expressions in parentheses, `expression1` and `expression2` are called the ***controlling expressions*** because they control which part of the **if-else** statement is used. If the controlling expression `expression1` is true, then the `statements1` following the **if** statement are executed and the statements following the **else if** and **else** statements get skipped. If the controlling expression `expression1` is false, then the `statements1` following the **if** statement are skipped and the next controlling expression, `expression2` is evaluated. If `expression2`

is true, then the `statements2` following the **else if** statement are executed and the `statements3` following the **else** statement get skipped. If `expression2` is false, however, then the `statements2` following the **else if** statement get skipped and the `statements3` following the **else** statement are executed. An **if-else** statement always has an **if** statement. The **else if** and **else** statements are optional.

Program 13.17 copies the position of the one Linkbot to the other Linkbot. Program A.1 controls the joint speed for a Linkbot by the corresponding joint angle of a controlled Linkbot. In Program A.1, the statement

```
robot1.relaxJoints();
```

relaxes joints 1 and 3 of `robot1`. This allows the user to turn each joint freely. The statement

```
robot2.driveForeverNB();
```

drives joints 1 and 3 of `robot1` forever.

A **while** loop is used to control the speed of joints 1 and 3 for `robot2` based on the joint angles of joints 1 and 3 of `robot1`. Inside the **while** loop, first the statement

```
speed1 = angle1;
```

obtains the joint angles for joints 1 and 3. The statement

```
speed1 = angle1;
```

then assigns the value of the angular position for joint 1 as the value of the speed for joint 1. The statements

```
if(speed1 > 200) {
   speed1 = 200;
}
else if(speed1 < -200)
   speed1 = -200;
}
```

compare the value of `speed1` with the maximum positive and negative speeds that a Linkbot joint can move. If the first controlling expression `speed1 > 200` is true, then the variable `speed1` is changed to the maximum speed of 200 degrees per second. If `speed1 > 200` is false, however, then the program skips to the **else if** statement and evaluates the second controlling expression `speed1 < -200`. If this expression is true, then `speed1` is changed to the value -200 degrees per second. If both controlling statements prove to be false, then `speed1` is unchanged. The **if** statement and **if else** statements that check the value of `speed1` are used as boundary checks. This helps ensure that the speed of joint 1 is not set higher than the maximum speed of 200 degrees per second or lower than the speed of -200 degrees per second. A similar speed boundary check is performed for joint 3 in the program.

```
/* Filename: speedcontrol.ch
   The joint angles for robot1 (green) will determine
   the speed of joints for robot2 (red). */
#include <linkbot.h>
CLinkbotI robot1, robot2;
double angle1, angle3;
double speed1, speed3;

/* Set colors for robots. Use the green robot to control the red one */
robot1.setLEDColor("green");
robot2.setLEDColor("red");

/* relax all joints of robot1 */
robot1.relaxJoints();

/* set joints of robot2 to move forward */
robot2.driveForeverNB();

while(1) {
    robot1.getJointAngles(angle1, NaN, angle3);

    speed1 = angle1;
    if(speed1 > 200) {
        speed1 = 200;
    }
    else if(speed1 < -200) {
        speed1 = -200;
    }

    speed3 = angle3;
    if(speed3 > 200) {
        speed3 = 200;
    }
    else if(speed3 < -200) {
        speed3 = -200;
    }

    robot2.setJointSpeeds(speed1, NaN, -speed3);
}
```

Program A.1: Controlling the joint speed for a Linkbot by the corresponding joint angle of a controlled Linkbot.

Do Exercise 2, on page 350.

A.2 Use a while-Loop for Repeating Motions

```
/* File: whileloop.ch
   repeat motions using a while-loop */
#include <linkbot.h>
CLinkbotI robot;
int i, num;

/* repeat rolling forward and backward 3 times */
num = 3;
i=0;
while(i<num)  {
    robot.driveAngle(360);   // drive forward 360 degrees
    robot.driveAngle(-360);  // drive backward 360 degrees
    i++;                       // increment i
}
```

Program A.2: Rolling a Linkbot-I forward and backward three times using a **while** loop.

A **while** loop in Ch can be used to repeat actions. Program A.2 uses a **while** loop to repeat the motion of rolling forward 360 degrees and rolling backward 360 degrees. The **while** loop consists of the code fragment below.

```
    /* repeat rolling forward and backward 3 times */
    num = 3;
    i=0;
    while(i<num)  {
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
        i++;
    }
```

The variable `num` with the value of 3 is the number of the repetitions. The variable `i` is the loop control variable. It is initialized to zero. The loop control expression `i<num` is the loop continuation condition. If the control variable `i` is less than `num`, the statements below in the loop body are executed.

```
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
        i++;                              // increment i;
```

At the end of the loop body, the variable `i` is incremented by 1. While the value of `i` is less than the value of `num`, the loop iterates. Because `num` is 3, the motion of rolling forward 360 degrees and rolling backward 360 degrees will be repeated three times.

E Do Exercise 3, on page 350.

A.3 Use a for-Loop for Repeating Motions

```
/* File: forloop.ch
   repeat motions using a for-loop */
#include <linkbot.h>
CLinkbotI robot;
int i, num;

/* repeat rolling forward and backward 3 times */
num = 3;
for (i=0; i<num; i++) {
    robot.driveAngle(360);   // roll forward 360 degrees
    robot.driveAngle(-360);  // roll backward 360 degrees
}
```

Program A.3: Rolling a Linkbot-I forward and backward three times using a **for** loop.

A **for** loop in Ch can also be used to repeat actions. Similar to Program A.2, Program A.3 uses a **for** loop to repeat the motion of rolling forward 360 degrees and rolling backward 360 degrees three times. The **for** loop consists of the code fragment below.

```
    /* repeat rolling forward and backward 3 times */
    num = 3;
    for (i=0; i<num; i++) {
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
    }
```

The variable `num` with the value of 3 is the number of the repetitions. There are three expressions in the **for** loop control structure

```
    for (i=0; i<num; i++)
```

The first expression `i=0` initializes the loop control variable `i` with 0. The second expression `i<num` is the loop continuation condition. The loop iterates when the control variable `i` is less than `num`. The third expression `i++` increments the control variable after the loop body

```
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
```

is executed.

The above **for** loop is the same as the following **while** loop presented in section A.2.

```
    /* repeat rolling forward and backward 3 times */
    num = 3;
    i=0;
    while(i<num)  {
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
        i++;
    }
```

E Do Exercise 4, on page 350.

A.4 Use a Function with an Argument of the CLinkbotI Class

```
/* File: function.ch
   repeat motions using the functions roll() */
#include <linkbot.h>
CLinkbotI robot;

/* the function prototype for roll() */
void roll(CLinkbotI &robot);

/* call the function roll() three times
   to repeat the motions defined in the function three times */
roll(robot);
roll(robot);
roll(robot);

/* define the function roll() */
void roll(CLinkbotI &robot) {
    robot.driveAngle(360);   // drive forward 360 degrees
    robot.driveAngle(-360);  // drive backward 360 degrees
}
```

Program A.4: Rolling a Linkbot-I forward and backward using the function `roll()`.

A function can be written in Ch to also repeat actions. Program A.4 uses a function `roll()` for the motion of rolling forward 360 degrees and rolling backward 360 degrees. The function prototype for the function `roll()` is as follows.

```
    /* the function prototype for roll() */
    void roll(CLinkbotI &robot);
```

Since the data type **void** is the return type for the function `roll()`, the function does not return a value. The function uses an argument of reference type of **CLinkbotI**. Therefore, the variable name `robot` can be used inside the function to access the member functions of **CLinkbotI**. The function `roll()` is defined as

```
    /* define the function roll() */
    void roll(CLinkbotI &robot) {
        robot.driveAngle(360);   // drive forward 360 degrees
        robot.driveAngle(-360);  // drive backward 360 degrees
    }
```

When the function `roll()` is called, it will roll forward 360 degrees and roll backward 360 degrees. The program calls this function three times by the following statements.

```
    roll(robot);
    roll(robot);
    roll(robot);
```

Therefore, the motion of rolling forward 360 degrees and rolling backward 360 degrees will be repeated three times.

E Do Exercises 5, 6, and 7 on page 350.

A.4.1 Exercises

1. Watch the following video tutorials for RoboBlockly in http://roboblockly.ucdavis.edu/videos/.
   (a) C6. Repeating with a Loop
   (b) C7. Using a Variable with a Loop
   (c) C12. Making Decision Using if-Statement
   (d) C13. Making Decision Using if-else if-else statement

2. Write a program `speedcontrol2.ch`, based on the program `speedcontrol.ch` in Program A.1, to control joint speeds of the controller Linkbot-I by the joint angles of the controlled Linkbot-I by changing the statements

```
    speed1 = angle1;
    speed3 = angle3;
```

   to

```
    speed1 = 2*angle1;
    speed3 = 2*angle3;
```

   The controlled robot will move faster by the same joint angle of the controlling robot. Test the program by changing the statements to

```
    speed1 = angle1/2;
    speed3 = angle3/2;
```

3. Write a program `whileloop2.ch` using a **while** loop to repeat the motion of rotating joint 1 by 360 degrees and then by -360 degrees for five times. Use the member function **robot.moveJoint**().

4. Write a program `forloop2.ch` using a **for** loop to repeat the motion of rotating joint 1 by 360 degrees and then by -360 degrees for five times.

5. Write a program `function2.ch` using a function `rotateJoint1()` for the motion of rotating joint 1 by 360 degrees and then by -360 degrees. Call the function `rotateJoint1()` five times to repeat the motion defined in the function.

6. The function `forwardturn()` is defined as follows:

```
void forwardturn(CLinkbotI &robot) {
    double radius = 1.75;
    double trackwidth = 3.69;

    robot1.driveDistance(5);
    robot2.turnRight(90, radius, trackwidth);
}
```

   Write a program `square.ch` to call the function `forwardturn()` four times for a robot to move along the path of a square.

7. The function `wait5robots()` is defined as follows:

```
void wait5robots(CLinkbotI &robot1, CLinkbotI &robot2,  CLinkbot &robot3,
                 CLinkbotI &robot4, CLinkbotI &robot5) {
    robot1.moveWait();
    robot2.moveWait();
```

```
    robot3.moveWait();
    robot4.moveWait();
    robot5.moveWait();
}
```

Modify the program `snake.ch` in Program 16.3 to call the function `wait5robots()` once using

```
wait5robots(robot1, robot2, robot3, robot4, robot5);
```

instead of calling the member function **moveWait()** five times.

APPENDIX B

Controlling Mindstorms and Linkbots in a Single Program

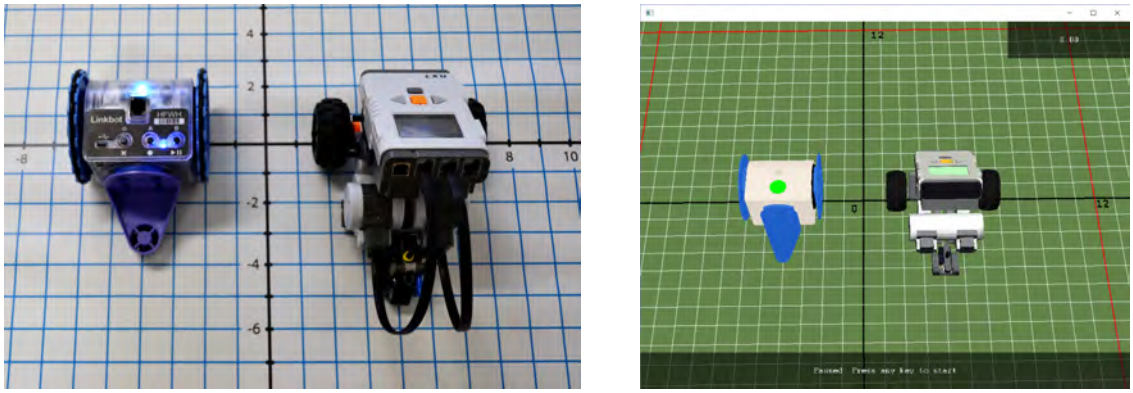


Figure B.1: Linkbot and Mindstorms controlled together.

In order to use Linkbots and Mindstorms NXT/EV3 together, connect to the robots as you usually would, through Linkbot Labs for Linkbot and Ch Mindstorms Controller for Mindstorms. Keep the robots connected as you run the programs. Virtual Linkbots and Mindstorms can be controlled in RoboSim, by adding a robot of each type from the "Individual Robots" panel on the left side of the RoboSim window.

We will illutrate how to control both Linkbots and Mindstorms together through a single program by solving the following problem.

Problem Statement:
A Linkbot-I and a Mindstorms are congured as two-wheel robots as shown in Figure B.1. The radii of the Linkbot wheels are 1.75 inches, and the radii of the Mindstorms wheels are 1.1 inches. Write a program `linkbotmindstorms1.ch` for two robots racing in the following

manner. Both robots will drive from a starting line at different times. The Linkbot drives for 16 seconds at the speed of 1.5 inches per second. Then 8 seconds after the Linkbot has left the starting line, the Mindstorms races at the speed of 3 inches per second for 8 seconds.

This problem statement is simila to the one presented in section 13.8. The only difference is that now the second robot is a Mindstorms. Based on Program 13.11, we developed Program B.1 to solve the above problem.

```
/* File: linkbotmindstorms.ch
   Linkbot moves first for 16s at 1.5in/s.
   8 seconds later, a NXT/EV3 moves for 8s at 3in/s.
   The equations of motions are
        d = 1.5t
        d = 3(t-8) */
#include <linkbot.h>
#include <mindstorms.h>
CLinkbotI robot1;
CMindstorms robot2;
double speed1=1.5, speed2=3;
double radius1=1.75, radius2=1.1;
double time1=16, delaytime=8, time2=8;

robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveTime(time2);
robot1.moveWait();
```

Program B.1: Conrolling a Linkbot-I and a Mindstorms with a single program.

In Program B.1, statements

```
#include <linkbot.h>
#include <mindstorms.h>
CLinkbotI robot1;
CMindstorms robot2;
double speed1=1.5, speed2=3;
double radius1=1.75, radius2=1.1;
```

include the header file **linkbot.h** for Linkbot and **mindstorms.h** for Mindstorms. Variables `robot1` and `robot2` are declared for two different types of robots. The radii of different robots are set. The remaining part of Program B.1 is the same as that in Program 13.11.

Similar to Program 13.12, Program B.2 extends Program B.1 to record and plot the distances versus time for the two robots, as shown in Figure B.2.

```
/* File: linkbotmindstorms2.ch
   A Linkbot moves first for 16s at 1.5in/s.
   8 seconds later, a NXT/EV3 moves for 8s at 3in/s.
   The equations of motions are
          d = 1.5t
          d = 3(t-8)
   Record time and distances using driveTime() and driveTimeNB().
   plot the acquired data for two robots */
#include <linkbot.h>
#include <mindstorms.h>
#include <chplot.h>
CLinkbotI robot1;
CMindstorms robot2;
double speed1=1.5, speed2=3; // speeds of robots in inches per second
double radius1=1.75, radius2=1.1; // the radii of two wheels of robot1 and 2
double time1=16, time2=8;    // motion time in seconds for robot1 and robot2
double delaytime=8;          // delay time for robot2
double timeInterval = 0.1;   // time interval in 0.1 second
int numDataPoints1, numDataPoints2; // number of data points recorded
robotRecordData_t timedata1, distances1; // recorded time and distances for robot1
robotRecordData_t timedata2, distances2; // recorded time and distances for robot2
CPlot plot;                  // plotting class

/* move to the zero position at the same time. */
robot1.resetToZeroNB();  robot2.resetToZeroNB();
robot1.moveWait();       robot2.moveWait();

/* set the speeds for robot1 and robot2 */
robot1.setSpeed(speed1, radius1);
robot2.setSpeed(speed2, radius2);

/* disable record data shift and begin recording time and distance */
robot1.recordNoDataShift();
robot2.recordNoDataShift();
robot1.recordDistanceBegin(timedata1,distances1,radius1,timeInterval);
robot2.recordDistanceBegin(timedata2,distances2,radius2,timeInterval);

/* robot1 drives first for a total of 'time1' seconds. 'delaytime' seconds later,
   robot2 drives for 'time2' seconds while robot1 also drives */
robot1.driveTimeNB(time1);
robot2.delaySeconds(delaytime);
robot2.driveTime(time2);
robot1.moveWait();  // wait till robot1 moved 'time1' seconds

/* end recording time and distance */
robot1.recordDistanceEnd(numDataPoints1);
robot2.recordDistanceEnd(numDataPoints2);

/* plot the data */
plot.title("Distances versus time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Distances (inches)");
plot.data2DCurve(timedata1, distances1, numDataPoints1);
plot.legend("Distance for robot 1");
plot.data2DCurve(timedata2, distances2, numDataPoints2);
plot.legend("Distance for robot 2");
plot.plotting();
```

Program B.2: Plotting the distances versus time for the Linkbot and Mindstorms.

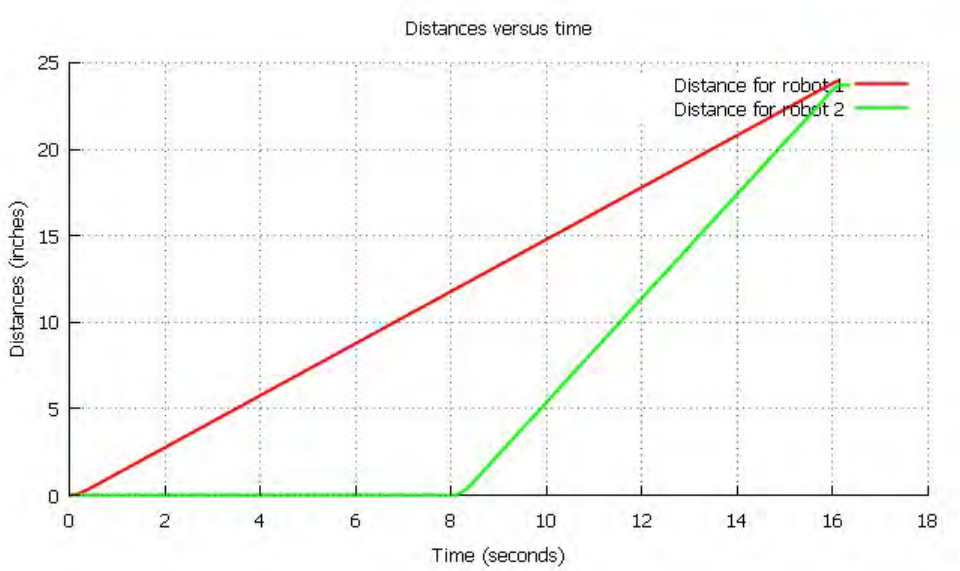


Figure B.2: The plot for the distances versus time from Program B.2.

B.0.1 Exercises

1. A Linkbot-I and a Mindstorms are configured as two-wheel robots. The radii of the Linkbot wheels are 1.75 inches, and the radii of the Mindstorms wheels are 1.1 inches. Write a program `linkbotmindstorms3.ch` for two robots racing in the following manner. The robots will drive from a starting line at different times. The Linkbot drives for 10 seconds at the speed of 1.2 inches per second. Then 5 seconds after the Linkbot has left the starting line, the Mindstorms races at the speed of 2.4 inches per second for 5 seconds.

2. Modify the program `linkbotmindstorms3.ch` in Exercise 1 as the program `linkbotmindstorms4.ch` to produce a plot shown below.

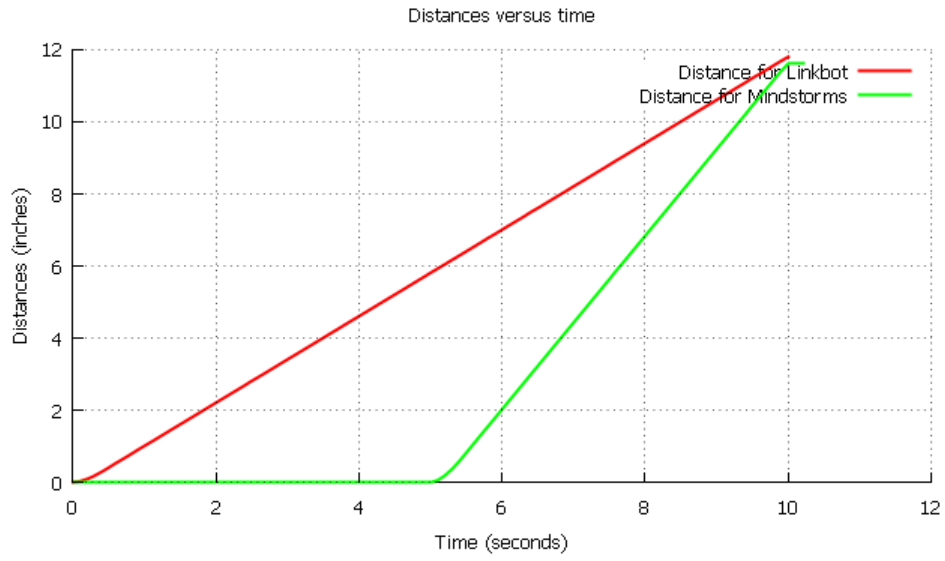


Figure B.3: The plot for the distances versus time for the program linkbotmindstorms4.ch.

3. A Linkbot-I and a Mindstorms are configured as two-wheel robots. The radii of the Linkbot wheels are 1.75 inches, and the radii of the Mindstorms wheels are 1.1 inches. Write a program `linkbotmindstorms5.ch` for two robots racing in the following manner. Both robots will drive from a starting line at different times. The Mindstorms drives for 24 inches at the speed of 1.5 inches per second. Then, 8 seconds after the Mindstorms has left the starting line, the Linkbot races for 24 inches at the speed of 3 inches per second. Produce a plot of distance versus time for the robots as shown below.

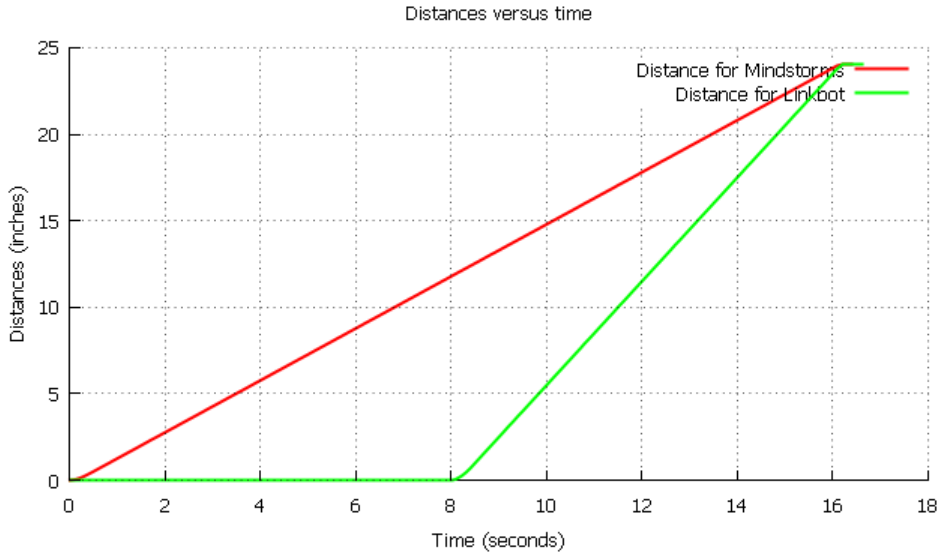


Figure B.4: The plot for the distances versus time from the program `linkbotmindstorms5.ch`.

APPENDIX C

Colors Available for Use with the Member Functions setLEDColor() and getLEDColor()

This appendix lists all the possible color names that can be used to set and get the LED color using member functions **setLEDColor()** and **getLEDColor()**. Each name must be enclosed with quotation marks when used as the argument for **setLEDColor()**. The name of each color is listed along with its corresponding RGB values in Table C.1 below.

Table C.1: The colors available for use with **setLEDColor()** and **getLEDColor()**.

Color Name	Red	Green	Blue
aliceBlue	240	248	255
antiqueWhite	250	235	215
aqua	0	255	255
aquamarine	127	255	212
azure	240	255	255
beige	245	245	220
bisque	255	228	196
black	0	0	0
blanchedAlmond	255	235	205
blue	0	0	255
blueViolet	138	43	226

(Continued)

Table C.1: (Continued)

Color Name	Red	Green	Blue
brown	165	42	42
burlyWood	222	184	135
cadetBlue	95	158	160
chartreuse	127	255	0
chocolate	210	105	30
coral	255	127	80
cornflowerBlue	100	149	237
cornSilk	255	248	220
crimson	220	20	60
darkBlue	0	0	139
darkCyan	0	139	139
darkGoldenrod	184	134	11
darkGray	169	169	169
darkGreen	0	100	0
darkKhaki	189	183	107
darkMagenta	139	0	139
darkOliveGreen	85	107	47
darkOrange	255	140	0
darkOrchid	153	50	204
darkRed	139	0	0
darkSalmon	233	150	122
darkSeaGreen	143	188	143
darkSlateBlue	72	61	139
darkSlateGray	47	79	79
darkTurquoise	0	206	209
darkViolet	148	0	211
deepPink	255	20	147
deepSkyBlue	0	191	255
dimGray	105	105	105
dodgerBlue	30	144	255
fireBrick	178	34	34
floralWhite	255	250	240
forestGreen	34	139	34
fuchsia	255	0	255
gainsboro	220	220	220
ghostWhite	248	248	255
gold	255	215	0
goldenrod	218	165	32
gray	128	128	128
green	0	255	0
greenYellow	173	255	47

(Continued)

Table C.1: (Continued)

Color Name	Red	Green	Blue
honeydew	240	255	240
hotPink	255	105	180
indigo	75	0	130
ivory	255	255	240
khaki	240	230	140
lavender	230	230	250
lavenderBlush	255	240	245
lawnGreen	124	252	0
lemonChiffon	255	250	205
lightBlue	173	216	230
lightCoral	240	128	128
lightCyan	224	255	255
lightGoldenrodYellow	250	250	210
lightGray	211	211	211
lightGreen	144	238	144
lightPink	255	182	193
lightSalmon	255	160	122
lightSeaGreen	32	178	170
lightSkyBlue	135	206	250
lightSlateGray	119	136	153
lightSteelBlue	176	196	222
lightYellow	255	255	224
limeGreen	50	205	50
linen	250	240	230
magenta	255	0	255
maroon	128	0	0
mediumAquamarine	102	205	170
mediumBlue	0	0	205
mediumOrchid	186	85	211
mediumPurple	147	112	219
mediumSeaGreen	60	179	113
mediumSlateBlue	123	104	238
mediumSpringGreen	0	250	154
mediumTurquoise	72	209	204
mediumVioletRed	199	21	133
midnightBlue	25	25	112
mintCream	245	255	250
mistyRose	255	228	225
moccasin	255	228	181
navajoWhite	255	222	173
navy	0	0	128
oldLace	253	245	230
olive	128	128	0

(Continued)

Table C.1: (Continued)

Color Name	Red	Green	Blue
oliveDrab	107	142	35
orange	255	165	0
orangeRed	255	69	0
orchid	218	112	214
paleGoldenrod	238	232	170
paleGreen	152	251	152
paleTurquoise	175	238	238
paleVioletRed	219	112	147
papayaWhip	255	239	213
peachPuff	255	218	185
peru	205	133	63
pink	255	192	203
plum	221	160	221
powderBlue	176	224	230
purple	128	0	128
red	255	0	0
rosyBrown	188	143	143
royalBlue	65	105	225
saddleBrown	139	69	19
salmon	250	128	114
sandyBrown	244	164	96
seaGreen	46	139	87
seaShell	255	245	238
sienna	160	82	45
silver	192	192	192
skyBlue	135	206	235
slateBlue	106	90	205
slateGray	112	128	144
snow	255	250	250
springGreen	0	255	127
steelBlue	70	130	180
tan	210	180	140
teal	0	128	128
thistle	216	191	216
tomato	255	99	71
turquoise	64	224	208
violet	238	130	238
wheat	245	222	179
white	255	255	255
whiteSmoke	245	245	245
yellow	255	255	0
yellowgreen	154	205	50

APPENDIX D

Melodies and Music Notes Defined in Ch

D.1 Melodies in Ch

Table D.1 lists all pre-composed melodies in Ch available for robots.

Table D.1: Names of songs and their corresponding melody names in Ch.

Name of Song	Name of Melody in Ch
All Star	AllStar
The Ants Go Marching	AntsGoMarching
The Ants Go Marching (high pitch)	AntsGoMarchingHighPitch
B-I-N-G-O	Bingo
Burn (Hamilton)	Burn
Colors of the Wind	ColorsOfTheWind
Despacito	Despacito
Do Re Mi	DoReMi
Happy Birthday	HappyBirthday
Hedwig's Theme	HedwigsTheme
How Far I'll Go	HowFarIllGo
I See the Light	ISeeTheLight
The Ice Cream Truck Jingle	IceCream
Jingle Bells	JingleBells
Let It Go	LetItGo
Marry Had A Little Lamb	LittleLamb
Twinkle Twinkle Little Star	LittleStar
Mario Theme	MarioTheme
Mario Underwordl	MarioUnderworld
Merry Christmas	MerryChristmas
Mii Theme	MiiTheme
Never Gonna Give You Up	NeverGonnaGiveYouUp
Old Mc Donald Had A Farm	OldMcDonald
Pirates Theme	PirateTheme
A Typical Phone Ring Tone	RingTone
Row Your Boat	RowYourBoat
Star Wars Theme	StarWarsTheme
Dance Music	Techno
The Star-Spangled Banner	TheStarSpangledBanner
The Wheels On The Bus	WheelsOnTheBus

D.2 Music Notes Defined in Ch

Table D.2 lists all macros defined in Ch for music notes. Notice that not all frequencies for a robot can be played due to the limitation of the hardware. For example, Lego NXT can only accept frequencies from 200 Hz to 14000 Hz whereas EV3 can only play notes from C4 to C7 (262 Hz to 2093 Hz). Therefore, check the frequency range of your robot before composing a melody. Otherwise, your robot may not be able to correctly play notes out-of-range.

Table D.2: Music notes defined in Ch

Ch Macro	Letter	Frequency (Hz)	Ch Macro	Letter	Frequency (Hz)
NOTE_B0	B0	31	NOTE_GS4	GS4	415
NOTE_C1	C1	33	NOTE_A4	A4	440
NOTE_CS1	CS1	35	NOTE_AS4	AS4	466
NOTE_D1	D1	37	NOTE_B4	B4	494
NOTE_DS1	DS1	39	NOTE_C5	C5	523
NOTE_E1	E1	41	NOTE_CS5	CS5	554
NOTE_F1	F1	44	NOTE_D5	D5	587
NOTE_FS1	FS1	46	NOTE_DS5	DS5	622
NOTE_G1	G1	49	NOTE_E5	E5	659
NOTE_GS1	GS1	52	NOTE_F5	F5	698
NOTE_A1	A1	55	NOTE_FS5	FS5	740
NOTE_AS1	AS1	58	NOTE_G5	G5	784
NOTE_B1	B1	62	NOTE_GS5	GS5	831
NOTE_C2	C2	65	NOTE_A5	A5	880
NOTE_CS2	CS2	69	NOTE_AS5	AS5	932
NOTE_D2	D2	73	NOTE_B5	B5	988
NOTE_DS2	DS2	78	NOTE_C6	C6	1047
NOTE_E2	E2	82	NOTE_CS6	CS6	1109
NOTE_F2	F2	87	NOTE_D6	D6	1175
NOTE_FS2	FS2	93	NOTE_DS6	DS6	1245
NOTE_G2	G2	98	NOTE_E6	E6	1319
NOTE_GS2	GS2	104	NOTE_F6	F6	1397
NOTE_A2	A2	110	NOTE_FS6	FS6	1480
NOTE_AS2	AS2	117	NOTE_G6	G6	1568
NOTE_B2	B2	123	NOTE_GS6	GS6	1661
NOTE_C3	C3	131	NOTE_A6	A6	1760
NOTE_CS3	CS3	139	NOTE_AS6	AS6	1865
NOTE_D3	D3	147	NOTE_B6	B6	1976
NOTE_DS3	DS3	156	NOTE_C7	C7	2093
NOTE_E3	E3	165	NOTE_CS7	CS7	2217
NOTE_F3	F3	175	NOTE_D7	D7	2349
NOTE_FS3	FS3	185	NOTE_DS7	DS7	2489
NOTE_G3	G3	196	NOTE_E7	E7	2637
NOTE_GS3	GS3	208	NOTE_F7	F7	2794
NOTE_A3	A3	220	NOTE_FS7	FS7	2960
NOTE_AS3	AS3	233	NOTE_G7	G7	3136
NOTE_B3	B3	247	NOTE_GS7	GS7	3322
NOTE_C4	C4	262	NOTE_A7	A7	3520
NOTE_CS4	CS4	277	NOTE_AS7	AS7	3729
NOTE_D4	D4	294	NOTE_B7	B7	3951
NOTE_DS4	DS4	311	NOTE_C8	C8	4186
NOTE_E4	E4	330	NOTE_CS8	CS8	4435
NOTE_F4	F4	349	NOTE_D8	D8	4699
NOTE_FS4	FS4	370	NOTE_DS8	DS8	4978
NOTE_G4	G4	392			

APPENDIX E

Quick References to Ch

This appendix summarizes Ch features used in this book.

E.1 Ch Language

E.1.1 Reserved Keywords

double else for if int return while

E.1.2 Reserved Names

CPlot M_PI NaN printf pow randint scanf sqrt

E.1.3 Punctuators

```
+   -   *   /   (   )   .   \   "   #   &   {   }
```

E.1.4 Comments

```
/* This is a comment */
int i;       // this is also a comment
```

E.1.5 Header Files

```
#include <chplot.h>   // for the plotting class CPlot
#include <linkbot.h>  // for the robot class CLinkbotI and CLinkbotL
```

E.1.6 Declaration of Variables for Integers and Decimals

```
int i, n=10;          // variables for integers
double x, y=1.2;      // variables for decimals
```

E.1.7 Declaration of Arrays

```
array double a[32], b[2]={1,2};    // variables for arrays
a[0] = a[31];  // accessing 1st and last elements of an array
```

E.1.8 Declaration of Functions

```
/* define a function func(x) for f(x)=2x-3 */
double func(double x) {
    return 2*x - 3;
}
```

E.1.9 Define Macros

```
#define N 10
```

E.1.10 The if-else Statement

```
if(x<0) {
    y = 2*x;
}
else {
    y = 3*x;
}
```

E.1.11 The while Loop

```
i=0;
while(i < num) {
    sum = sum + i;
    i = i + 1;
}
```

E.1.12 Input Function scanf() and Output Function printf()

```
printf("i= %d, x = %lf\n", i, x);  // print integer i and decimal x
printf("%.2lf\n", x);     // print two digits after the decimal point
printf("%E\n", x);        // print x in scientific notation
scanf("%d", &i);          // get the user input for i
scanf("%d%lf", &i, &x);   // get the user input for i and x
```

Print special characters using `"\n"` for a newline character and `"%%"` for the percent symbol `%`.

```
printf("Hello, World\n");
printf("We sell Yogurt with 10%% discount\n"); // with 10% discount
```

E.1.13 Math Operators

Description	Operator in math	Operator in Ch	Operation
addition	$+$	+	$x + y$
subtraction	$-$	-	$x - y$
multiplication	$\times$	*	$x * y$
multiplication	$\cdot$	*	$x * y$
division	$\div$	/	x / y

Unless the result of a division operation is a whole number, to get the correct numerical result, one of the numbers must be inputted as a decimal number. For example, the math expression $\frac{2(3+4)}{5-6}$ can be inputted in Ch as `2.0*(3+4)/(5-6)`.

E.1.14 Relational Operators

```
<    <=    ==    >=    >    !=
```

E.1.15 Math Functions abs(), sqrt(), pow(), and hypot()

The absolute function **abs**(x) is for $|x|$. The square root function **sqrt**(x) is for $\sqrt{x}$. The exponential function **pow**(x, y) is for x^y. The hypotenuse function **hypot**(a, b) is for $\sqrt{a^2 + b^2}$.

```
y = abs(x);
y = sqrt(x);
y = pow(x, 3);
c = hypot(a, b);
```

E.2 Member Functions of the Plotting Class CPlot

This document uses several member functions of the plotting class **CPlot** to create and manipulate plots. Detailed information for member functions of the class **CPlot** can be found in the section for the header file **chplot.h** in *Ch Reference Guide*, which is available in a PDF file `chref.pdf` by clicking the command `Help->Help` in ChIDE. The usages of the member functions in this document are summarized in this appendix.

Math Symbols

The Ch plotting uses the enhanced text for strings such as title, labels, and legends. Special symbols for the enhanced text can be found in *Ch Reference Guide*. The strings `"<="` and `">"` represent $\leq$ and $\geq$, respectively. To print a superscript on a plot, the symbol `^` is used, such as `x^2` for x^2. To print a subscript on a plot, the symbol `_` is used, such as `x_2` for x_2. The string `"M_PI"` is for π. The string `"{/Symbol @{\\140}\\326}x"` is used to print a square root function sign $\sqrt{x}$.

Handling Outline and Filled Color of Objects

The member functions **circle**(), **rectangle**(), **rectanglexy**(), **quad**(), **polygon**(), **regularPolygon**(), and **triangle**() can add correponding objects to a 2D plot. These objects are not counted as a data set for later calls to **CPlot**::**legend**().

By default, the line width of the outline of an object is 2. The color of the outline of an object is red, green, blue, magenta, cyan, yellow, black, orange, or grey. The color will be cycled if more than eight objects are added. The default line width and color of the outline of an object can be changed by **strokeWidth**() and **strokeColor**(), respectively. The outline of an object can be disabled by **noStrokeColor**().

By default, the interior of an object is not filled. The interior of an object can be filled with a solid color by **fillColor**(). The opacity of the filled color can be set by **fillOpacity**() in the range from 0 to 1, with 0 for complete transparency and 1 for solid color. The member function **noFill**() will disable the filled color.

plot.arrow(double x_tail, double y_tail, double x_head, double y_head)

adds an arrow to a plot. The arrow points from (*x_tail*, *y_tail*) to (*x_head*, *y_head*). For example,

```
plot.arrow(x1, y1, x2, y2);
```

This member function can be used to draw arrows in x and y axes.

plot.axisRange(int axis, double minimum, double maximum)

sets the minimum and maximum value for an axis. The first argument `axis` specifies which axis. The commonly used macros for the argument `axis` are **PLOT_AXIS_X** and **PLOT_AXIS_Y** for x-axis and y-axis, respectively. The second argument `minimum` is for the minimum value of the axis. The third argument `maximum` is for the maximum value of the axis. For example,

```
plot.axisRange(PLOT_AXIS_X, -10, 10);
plot.axisRange(PLOT_AXIS_Y, -10, 10);
```

plot.circle(double x, double y, double radius)

draws a circle centered at the point `(x, y)` with the radius `r`. For example,

```
plot.circle(0, 0, 3);
```

plot.data2DCurve(double x, double y, int n)

plots `n` data points stored in arrays `x` and `y` in a line plot. For example,

```
plot.data2DCurve(x, y, 100);
```

plot.dataFile(string_t filename)

plots data in the data file `filename`. For example,

```
plot.dataFile("position.txt");
```

plot.expr(double x0, double xf, int num, string_t expx)

plots a function defined as an expression `expx` in terms of the variable `x` in the range `[x0, xf]` with the number of points specified by the argument `num`. For example,

```
plot.expr(-10, 10, 500, "2*x+1");
```

plot.exprPoint(double x0, double xf, int num, string_t expx)

plots a function defined as an expression `expx` in terms of the variable `x` in the range `[x0, xf]` with the number of points specified by the argument `num` in scatter plot. For example,

```
plot.exprPoint(-10, 10, 500, "2*x+1");
```

plot.fillColor(string_t color)

sets the color to fill all objects. For example

```
plot.fillColor("yellow");
```

plot.fillCurve(string_t option, string_t color)
changes the previously added line or curve with the filled color based on the boundary specified in the option. It must be called before a curve is added. For example,

```
plot.fillCurve("y1=-20", "yellow");
```

specifies that the plot fills the area with the color yellow between the curve, created by the data points in next data set, and the horizontal line $y = -20$.

plot.fillOpacity(double opacity)
sets the opacity for the filled color for the interior of an object. For example,

```
plot.fillOpacity(0.5);
```

plot.func2D(double x0, double xf, int num, double func())
plots a function defined as `func()` in the range `[x0, xf]` with the number of points specified by the argument `num`. For example,

```
plot.func2D(-10, 10, 500, func);
```

plot.func2DPoint(double x0, double xf, int num, double func())
plots a function defined as `func()` in the range `[x0, xf]` with the number of points specified by the argument `num` in scatter plot. For example,

```
plot.func2DPoint(-10, 10, 500, func);
```

plot.label(int axis, string_t label)
adds a label to an axis. For example,

```
plot.label(PLOT_AXIS_X, "x");
plot.label(PLOT_AXIS_Y, "y=2x+5");
```

plot.legend(string_t legend)
adds a legend to the plot. For example,

```
plot.legend("y=2x+5");
```

plot.line(double x1, double y1, double x2, double y2)
draws a straight line between the two end points `(x1, y1)` and `(x2, y2)`. For example,

```
plot.line(-10, 150, 10, 250);
```

plot.lineStyle(string_t style)
specifies a line style. For example,

```
plot.lineStyle("."); // for dotted line
plot.lineStyle("-"); // for solid line
```

plot.noFill()
does not fill objects with color. For example,

```
plot.noFill();
```

plot.noStrokeColor()
disables the coloring of the outline of objects. For example,

```
plot.noStrokeColor();
```

plot.numberLine(double x0, double x1, ...)
draws a direction line to connect two adjacent points on the number line. Each direction line has a vertical offset from the number line. For example,

```
plot.numberLine(x0, x1, x2);
```

plot.numberLineScattern(double x[ ], int n)
plots data in array `x` with the `n` number of elements as a scatter plot with an offset on the number line. For example,

```
plot.numberLineScattern(x, n);
```

plot.plotting()
generates a plot. For example,

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

plot.point(double x, double y)
draws a point at the coordinate `(x, y)`. For example,

```
plot.point(-10, 10);
```

plot.pointStyle(string_t style)
specifies the point style, appliable to **point**(), **scattern**(), etc. For example,

```
plot.pointStyle(".");
plot.pointStyle("+");
```

plot.polygon(double x[ ], double y[])
draws a polygon. For example,

```
plot.polygon(x, y);
```

plot.quad(double x1, double y1, double x2, double y2, double x3, double y3, double x4, double y4)
draws a quadrilateral with four points `(x1, y1)`, `(x2, y2)`, `(x3, y3)`, and `(x4, y4)`. For example,

```
plot.quad(1, 2, 3, 4, 5, 6, 7, 8);
```

plot.rectangle(double x, double y, double width, double height)
draws a rectangle with the lower left corner at the point `(x, y)` and specified `width` and `height`. For example,

```
plot.rectangle(1, 2, 3, 4);
```

plot.rectanglexy(double x1, double y1, double x2, double y2)
draws a rectangle with the lower left corner at the point `(x1, y1)` and the upper right corner at the point `(x2, y2)`. For example,

```
plot.rectangle(1, 2, 4, 6);
```

plot.regularPolygon(double x, double y, int n, double length, double angle)
draws plot a regular polygon. For example,

```
plot.regularPolygon(0, 0, 8, 3, 30);
```

plot.scattern(double x[ ], double y[], int n)

plot.scattern(double x[ ], double y[], int n, string_t color)
plots `n` data points stored in arrays `x` and `y` in a scatter plot with a default or user specified color. For example,

```
plot.scattern(x, y, 100);
plot.scattern(x, y, 100, "green");
```

plot.sizeRatio(double ratio)
sets the x and y aspect ratio for the plot. For example,

```
plot.sizeRatio(1);
```

plot.strokeColor(string_t color)
sets the color for lines and points, including the outline of objects. For example,

```
plot.strokeColor("blue");
```

plot.strokeWidth(int width)
sets the width in pixel for lines and points, including the outline of objects. For example,

```
plot.strokeWidth(3);
```

plot.text(string_t string, int just, double x, double y)
adds the text at (x, y). For example,

```
plot.text("x", PLOT_TEXT_CENTER, x, y);
```

plot.ticsRange(int axis, double incr)
specifies the incremental value for tick marks for an axis.

```
plot.ticsRange(PLOT_AXIS_X, 1);
plot.ticsRange(PLOT_AXIS_Y, 5);
```

plot.title(string_t title)
adds a title, specified in a string, to the plot. For example,

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

plot.triangle(double x1, double y1, double x2, double y2, double x3, double y3)
draws a triangle with three points `(x1, y1)`, `(x2, y2)`, and `(x3, y3)`. For example,

```
plot.triangle(1, 2, 3, 4, 5, 6);
```

APPENDIX F

Quick Reference to Linkbot Member Functions

Some data types and member functions of the classes **CLinkbotI**, **CLinkbotL**, **CLinkbotIGroup**, and **CLinkbotLGroup** are defined in the header file **linkbot.h**. This appendix summarizes these data types and member functions used to control the Linkbots.

F.1 Data Types

The data types defined in the header file **linkbot.h** are described in this section. These data types are used by the Linkbot library to represent certain values, such as joint id's and motor directions.

Data Type	Description
robotJointId_t	An enumerated value that indicates a Linkbot joint.
robotRecordData_t	Recorded time, angle, distance, x or y coordinate.
note_t	Music note

The data type **robotJointId_t** is used to identify a joint on the Linkbot. Valid values for this type are listed in the table below:

Value	Description
JOINT1	Joint number 1 on the Linkbot.
JOINT2	Joint number 2 on the Linkbot.
JOINT3	Joint number 3 on the Linkbot.

The data type **robotRecordData_t** is used for the member functions **recordDistanceBegin**(), **recordAngleBegin**(), **recordAnglesBegin**(), and **recordxyBegin**().

The data type **note_t** is used for the member functions **playNotes**() and **playNotesNB**().

F.2 Member Functions Available in both Classes CLinkbotI and CLinkbotL

Member functions available in both classes CLinkbotI and CLinkbotL are listed in this section. A page number following a member function indicates where the member function is introduced. If there is no page number following a member function, the function has not be presented in this book.

robot.blinkLED(double delay, int numBlinks)

blinks the on-board LED. The argument `delay` is the amount of time between blinks. The argument `numBlinks` is the number of times to blink the LED. For example,

```
robot.blinkLED(0.1, 10);
```

robot.connectWithSerialID("serialID")

connects a robot to be controlled using the serial ID. The program then does not depend on the configuration set up by CMC. For example,

```
robot.connectWithSerialID("BNC2");
```

int robot.closeGripper(), page 328

closes a gripper. For example,

```
robot.closeGripper();
```

void robot.delaySeconds(double seconds), page 179

makes a program pause for the number of seconds specified before moving to the next line of code. For example,

```
robot.delaySeconds(3);
```

void robot.disconnect()

disconnects from a remote Linkbot. For example,

```
robot.disconnect();
```

void robot.getAccelerometerData(int port, double &x, double &y, double &z), page 224

measures the magnitude of gravitational forces in the X, Y and Z directions and stores values in variables `x`, `y`, and `z`.

```
robot.getAccelerometer(x, y, z);
```

void robot.getBatteryVoltage(double &voltage), page 225

measures the battery voltage and stores value in variable `voltage`.

```
robot.getBatteryVoltage(voltage);
```

void robot.getLEDColor(string_t color), page 203

passes the color of the robot's LED to the variable `color`. For example,

```
string_t color;
robot.getLEDColor(color);
```

void robot.getLEDColorRGB (int &r, int &g, int &b), page 204

passes the color of the robot's LED in RGB values to variables `r`, `g`, and `b`. For example,

```
robot.getLEDColorRGB (r, g, b);
```

void robot.getJointAngle(int id, double &angle), page 174

gets the joint angle of a Linkbot 10 times and stores the average of the 10 joint angles (in degrees) in the variable `angle`. `id` is used to specify which joint to monitor and is written as **JOINT1**, **JOINT2**, or **JOINT3** for the respective joints. For example,

```
robot.getJointAngle(JOINT2, angle);
```

void robot.getJointAngles(double &angle1, double &angle2, double &angle3), Page 176

is the same as **getJointAngle**(), but it gets the average angle value for both movable joints of a robot. Values for the average angle of each joint will be stored in the respective variables `angle1`, `angle2`, and `angle3`. For example,

```
robot.getJointAngles(NaN, angle2, angle3);
```

void robot.getJointAngleInstant(int id, double &angle)

gets a joint angle of a Linkbot (in degrees) in the variable `angle`. `id` is used to specify which joint to monitor and is written as **JOINT1**, **JOINT2**, or **JOINT3** for the respective joints. For example,

```
robot.getJointAngleInstant(JOINT2, angle);
```

void robot.getJointAnglesInstant(double &angle1, double &angle2, double &angle3)

is the same as **getJointAngle**(), but it gets the angle values for both movable joints of a robot. Values for the angle of each joint will be stored in the respective variables `angle1`, `angle2`, and `angle3`. For example,

```
robot.getJointAnglesInstant(NaN, angle2, angle3);
```

void robot.getJointSafetyAngle(double &angle)

gets the current joint safety angle limit of a robot. The default joint safety angle is 10 degrees. For example,

```
robot.getJointSafetyAngle(angle);
```

void robot.getJointSafetyAngleTimeout(double &seconds)

gets the current joint safety limit timeout of a robot. The default joint safety limit timeout is 0.5 second. For example,

```
robot.getJointSafetyAngleTimeout(seconds);
```

void robot.getJointSpeed(int id, double &speed), page 181

gets the speed (deg/sec) of a robot's joint and stores the value in the variable `speed`.

```
robot.getJointSpeed(JOINT2, speed);
```

void robot.getJointSpeedRatio(int id, double &ratio), page 183

gets the speed ratio of a robot's joint and records value in the variable `ratio`. Speed ratio is the current speed compared to the maximum speed (200 degrees per second). For example,

```
robot.getJointSpeedRatio(JOINT3, ratio);
```

void robot.getJointSpeedRatios(double &ratio1, double &ratio2, double &ratio3), page 184
gets the speed ratios of moving joints of a robot and records values in the variables `ratio1` and `ratio3`. For example,

```
robot.getJointSpeedRatios(NaN, ratio2, ratio3);
```

void robot.getJointSpeeds(double &speed1, double &speed2, double &speed3), page 182
gets the speed of both moving joints of a robot and stores value in the respective variable for that joint. For example,

```
robot.getJointSpeeds(NaN, speed2, speed3);
```

void robot.holdJoint(int id), page 327
holds a joint of the robot. For example,

```
robot.holdJoint(JOINT2);
```

void robot.holdJoints(), page 261
holds all the joints of the robot. For example,

```
robot.holdJoints();
```

void robot.holdJointsAtExit(), page 198
holds all the joints of the robot when the program finishes (exits). For example,

```
robot.holdJointsAtExit();
```

int robot.isMoving()
checks if the robot is moving any of its joints. For example,

```
if(robot.isMoving()) {
    printf("robot is moving\n");
}
```

int robot.move(double angle1, double angle2, double angle3), page 169
moves a robot's joints relative to their current positions by corresponding values of `angle1`, `angle2`, and `angle3`. For example,

```
linkboti.move(90, NaN, -90);
linkbotl.move(NaN, 30, 90);
```

int robot.moveForeverNB()
moves joints forever, while allowing the next lines of code to begin before the function **moveForeverNB**() has finished executing. For example,

```
robot.moveForeverNB();
```

int robot.moveJoint(int id, double angle), page 178
is the same as the **move**() function but allows you to specify just one joint to move. For example,

```
robot.moveJoint(JOINT3, 90);
```

int robot.moveJointNB(int id, double angle), page 232
A non-blocking version of **moveJoint**().

int robot.moveJointByPowerNB(int id, int power), page 281
moves a joint by setting the motor power of the joint. The specified power can be an integer from -100 to +100. For example,

```
robot.moveJointByPowerNB(JOINT2, 90);
```

int robot.moveJointForeverNB(int id), page 261
moves joints forever, while allowing the next lines of code to begin before the function **moveJointForeverNB**() has finished executing. For example,

```
robot.moveJointForeverNB(JOINT2);
```

int robot.moveJointTime(int id, double seconds), page 197
move a joint for a specified time in seconds. For example,

```
robot.moveJointTime(JOINT2, 5);
```

int robot.moveJointTimeNB(int id, double seconds), page 255
A non-blocking version of **moveJointTime**().

int robot.moveJointTo(int id, double angle), page 177
is the same as the **moveTo**() function but allows you to specify just one joint to move. For example,

```
robot.moveJointTo(JOINT2, 90);
```

int robot.moveJointToNB(int id, double angle), page 243
A non-blocking version of **moveJointTo**().

int robot.moveJointToByTrackPos(int id, double angle)
is the same as the **moveToByTrackPos**() function but allows you to specify just one joint to move. For example,

```
robot.moveJointToByTrackPos(JOINT2, 90);
```

int robot.moveJointToByTrackPosNB(int id, double angle)
A non-blocking version of **moveJointToByTrackPos**().

int robot.moveJointWait(int id), page 233
pauses the program until the movement of the specified joint, `id`, has stopped its current motion. For example,

```
robot.moveJointWait(JOINT2);
```

int robot.moveNB(double angle1, double angle2, double angle3), page 241
A non-blocking version of **move**().

int robot.moveTime(double seconds), page 196
moves joints for a specified time in seconds. For example,

```
robot.moveTime(5);
```

int robot.moveTimeNB(double seconds)
A non-blocking version of **moveTime**().

int robot.moveTo(double angle1, double angle2, double angle3), page 171
moves the respective joints of a robot to the absolute position of the specified angles. This is different from the **move**() function which moves the joints an angle relative to its current position. For example,

```
robot.moveTo(120, NaN, -120);
```

int robot.moveToNB(double angle1, double angle2, double angle3), page 243
A non-blocking version of **moveTo**().

int robot.moveToByTrackPos(double angle1, double angle2, double angle3), page 279
moves the respective joints of a robot to the absolute position of the specified angles by tracking the positions during the motion. This is different from the **moveTo**() function which moves the joints based on the specified joint speed. For example,

```
robot.moveToByTrackPos(NaN, 120, -120);
```

int robot.moveToByTrackPosNB(double angle1, double angle2, double angle3), page 279
A non-blocking version of **moveToByTrackPos**().

void robot.moveToZero()
moves all joints of a robot to their zero position. Unlike **resetToZero**(), **moveToZero**() will rewind each joint to zero position even for multiple turns. For example,

```
robot.moveToZero();
```

void robot.moveToZeroNB()
A non-blocking version of **moveToZero**().

void robot.moveWait(), page 234
pauses the program until the movement of *all* the joints of a robot have finished moving.

int robot.openGripper(double angle), page 328
opens a gripper with a specified angle in degrees. For example,

```
robot.openGripper(90);
```

int robot.playMelody(note_t song(int), double speedFactor), page 41
plays a song in melody. For example,

```
robot.playMelody(HappyBirthDay, 0.5); // faster
robot.playMelody(HappyBirthDay, 1);   // normal speed
robot.playMelody(HappyBirthDay, 2);   // slower
```

int robot.playMelodyNB(note_t song(int), double speedFactor), page 213
A non-blocking version of **playMelody**().

int robot.playNotes(note_t song[:], double speedFactor), page 218
plays music notes. For example,

```
robot.playNotes(notes, 1);   // normal speed
```

int robot.playNotesNB(note_t song[:], double speedFactor), page 222
A non-blocking version of **playNotes**().

int robot.playNotesWait(), page 213
pauses the program until the melody or music notes have finished playing.

int robot.recordAngleBegin(int id, double* &timedata, double* &angledata, double interval), page 193

begins recording data points of time and joint angle stored in the variables, `timedata` and `angledata`. `id` specifies which joint to record data for and `interval` is used to tell how often the program should record data points (in seconds). For example,

```
robot.recordAngleBegin(JOINT2, timedata, angledata, 0.1);
```

int robot.recordAngleEnd(int id, int &numDataPoints), page 193
ends the recording started by the member function **recordAngleBegin**() and passes the number of data points collected in the variable `numDataPoints`. For example,

```
robot.recordAngleEnd(JOINT2, numDataPoints);
```

int robot.recordAnglesBegin(double* &timedata, double* &angledata1, double* &angledata2,

double interval)
begins recording data points of time and two joint angles stored in the variables, `timedata`, `angledata1`, and `angledata2`. `angledata1` contains the data for joint 2. `angledata2` has the data for joint 3. `interval` is used to tell how often the program should record data points (in seconds). For example,

```
robot.recordAnglesBegin(timedata, angledata1, angledata2, 0.1);
```

int robot.recordAnglesEnd(int &numDataPoints)
ends the recording started by the member function **recordAnglesBegin**() and passes the number of data points collected in the variable `numDataPoints`. For example,

```
robot.recordAnglesEnd(numDataPoints);
```

void robot.relaxJoint(int id)
relaxes a joint of robot. For example,

```
robot.relaxJoint(JOINT2);
```

void robot.relaxJoints(), page 280
relaxes all joints of robot. For example,

```
robot.relaxJoints();
```

void robot.resetToZero(), page 92
resets the robot's joints to their absolute zero positions.

void robot.resetToZeroNB(), page 243
A non-blocking version of **resetToZero**().

void robot.setBuzzerFrequency(int frequency, double time), page 207
turns on the robot buzzer with a specified frequency (in Hz) for a time duration. For example,

```
robot.setBuzzerFrequency(450, 1.5);
```

void robot.setBuzzerFrequencyOff(), page 208
turns off the robot buzzer. For example,

```
robot.setBuzzerFrequencyOff();
```

void robot.setBuzzerFrequencyOn(int frequency), page 208
turns on the robot buzzer with a specified frequency (in Hz). For example,

```
robot.setBuzzerFrequencyOn(450);
```

void robot.setLEDColor(string_t "color"), pages 40
sets the robot LED to the specified color. For example,

```
robot.setLEDColor("blue");
```

void robot.setLEDColorRGB(int r, int g, keyredint b), page 204
sets the color of the robot's LED in RGB values. For example,

```
robot.setLEDColorRGB(160, 32, 240);
```

void robot.setJointSafetyAngle(double angle), page 329
sets the joint safety angle in degrees for a robot. The default joint safety angle is 10 degrees. For example,

```
robot.setJointSafetyAngle(50);
```

void robot.setJointSafetyAngleTimeout(double seconds)
sets the joint safety angle limit timeout of a robot in seconds. The default joint safety angle limit timeout is 0.5 seconds. For example,

```
robot.setJointSafetyAngleTimeout(1.5);
```

int robot.setJointSpeed(int id, double speed), page 181
sets the speed of a robot joint in degrees per second. For example,

```
robot.setJointSpeed(JOINT2, 90);
```

int robot.setJointSpeedRatio(int id, double ratio), page 183
sets the speed ratio of a robot joint. For example,

```
robot.setJointSpeedRatio(JOINT3, speedratio);
```

int robot.setJointSpeedRatios(double ratio1, double ratio2, double ratio3, double ratio4), page 184
sets the speed ratios of moving joints of a robot. For example,

```
robot.setJointSpeedRatios(NaN, ratio2, ratio3, NaN);
```

int robot.setJointSpeeds(double speed1, double speed2, double speed3), page 182
sets the speed of both moving joints of a robot in degrees per second. For example,

```
robot.setJointSpeeds(NaN, speed2, speed3, NaN);
```

void robot.systemTime(double &time), page 96
records the time in seconds since the system last started (Windows) or since 00:00:00 January 1, 1970 (Mac OS X and Linux). For example,

```
robot.systemTime(time1);
... (Code to be timed) ...
robot.systemTime(time2);
elapsedtime = time2 - time1;
```

F.3 Member Functions Available only in the Class CLinkbotI as a Two-Wheel Robot

Member functions available only in class CLinkbotI for a two-wheel robot are listed in this section.

All robot.drive*() functions treat the robot as a vehicle with the speed as setup by setSpeed(). To control each joint with a different speed, use robot.move*() functions such as robot.moveForeverNB() instead of robot.driveForeverNB().

void robot.driveAngle(double angle), page 35

drives both joints of a Linkbot forward or backward by the specified angle. For example,

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

void robot.driveAngleNB(double angle), page 247

A non-blocking version of **driveAngle**().

void robot.driveDistance(double distance, double radius), page 38

drives a two wheel robot a specified distance given its wheel radius. For example,

```
robot.driveDistance(10, radius);
```

void robot.driveDistanceNB(double distance, double radius), page 246

A non-blocking version of **driveDistance**().

void robot.driveForeverNB(), page 260

drives a Linkbot forever, while allowing the next lines of code to begin before the function **driveForeverNB**() has finished executing. For example,

```
robot.driveForeverNB();
```

void robot.driveTime(double seconds), page 123

drives a Linkbot for a specified time in seconds. For example,

```
robot.driveTime(5);
```

void robot.driveTimeNB(double seconds), page 255

A non-blocking version of **driveTime**().

void robot.getDistance(double &distance, double radius), page 92

records the distance that a Linkbot has moved using a specified wheel radius. For example,

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

void robot.recordDataShift(), page 251

enables the shifting of recorded data. By default, the record functions only record data while the Linkbot is in motion (data shifted). For example,

```
robot.recordDataShift();
```

int robot.recordDistanceBegin(double* &time, double* &distance, double radius, double timeInterval), page 103

begins recording the time and distance of a robot joint in variables `time` and `distance` (of type robotRecordData_t) given the specified wheel radius. `timeInterval` is the time interval between measurements. For example,

```
robot.recordDistanceBegin(timedata, distances, radius, 0.1);
```

int robot.recordDistanceEnd(int &numDataPoints), page 103

ends the **recordDistanceBegin**() function and passes the number of data points collected in the variable `numDataPoints`. For example,

```
robot.recordDistanceEnd(numDataPoints);
```

int robot.recordDistanceOffset(double offset), page 112

creates an offset so that when the **recordDistanceBegin**() function is called, each distance value will add an offset value. For example,

```
robot.recordDistanceOffset(2);
```

void robot.recordNoDataShift(), page 251

disables the shifting of recorded data. By default, the record functions only record data while the Linkbot is in motion (data shifted). For example,

```
robot.recordNoDataShift();
```

void robot.setSpeed(double speed, double radius), page 90

sets joints 2 and 3 of a Linkbot to the desired speed with the specified wheel radius. For example,

```
robot.setSpeed(4, radius);
```

void robot.turnLeft(double angle, double radius, double trackwidth), page 63

turns a two wheel robot left by a specified angle, given the wheel radius and trackwidth (distance between the two wheels). For example,

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

void robot.turnLeftNB(double angle, double radius, double trackwidth), page 247

A non-blocking version of **turnLeft**().

void robot.turnRight(double angle, double radius, double trackwidth), page 63

turns a two wheel robot left or right by a specified angle, given the wheel radius and trackwidth. For example,

```
robot.turnRight(90, radius, trackwidth);
```

void robot.turnRightNB(double angle, double radius, double trackwidth), page 247

A non-blocking version of **turnRight**().

F.4 Member Functions Available in Both Classes CLinkbotIGroup and CLinkbotLGroup

Member functions available only in classes **CLinkbotIGroup** and **CLinkbotLGroup** are listed first in this section. Most member functions in classes **CLinkbotI** and **CLinkbotL** are available in classes **CLinkbotIGroup** and **CLinkbotLGroup**. They are listed in a table in this section.

int group.addRobot(CLinkbotI &name), page 83

adds a robot using the robot's declared name to the class **CLinkbotIGroup**. For example,

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

int group.addRobots(array CLinkbotI name), page 87

adds an array of robots to a group using the robot array's declared name. For example,

```
CLinkbotI robot[4];
group.addRobots(robot);
```

Member Function Name	Description
addRobot()	Add a Linkbot as a member of the Linkbot group.
addRobots()	Add Linkbot to the Linkbot group.
closeGripper()	Close the gripper.
delaySeconds()	Pauses program for number of seconds specified.
driveAngle()	Drive each Linkbot forward or backward.
driveAngleNB()	Identical to **driveAngle()** but non-blocking.
driveDistance()	Drive each Linkbot in the group a certain distance.
driveDistanceNB()	Identical to **driveDistance()** but non-blocking.
driveForeverNB()	Drive each Linkbot forever until stopped.
driveTime()	Drive each Linkbot for a specified time.
driveTimeNB()	Identical to **driveTime()** but non-blocking.
holdJoint()	Hold a joint for all Linkbot in the group.
holdJoints()	Hold all the joints for all Linkbot in the group.
holdJointsAtExit()	Hold all the joints at exit for all Linkbot in the group.
isMoving()	Check if any joint of a robot is moving.
move()	Move two joints of each Linkbot by specified angles.
moveForeverNB()	Move joints forever untill stopped.
moveNB()	Identical to **move()** but non-blocking.
moveJoint()	Move a joint from its current position by an angle.
moveJointNB()	Identical to **moveJoint()** but non-blocking.
moveJointByPowerNB()	Move a joint by setting the power of the joint motor.
moveJointForeverNB()	Move a joint forever. A joint will move until stopped.
moveJointTime()	Move a joint for a specified time.
moveJointTimeNB()	Identical to **moveJointTime()** but non-blocking.
moveJointTo()	Set the desired joint position for a joint.
moveJointToNB()	Identical to **moveJointTo()** but non-blocking.
moveJointToByTrackPos()	Move a joint to the specified position by tracking its position.
moveJointToByTrackPosNB()	Identical to **moveJointToByTrackPos**(), but non-blocking.
moveJointWait()	Wait until the specified motor has stopped moving.
moveTime()	Move joints for a specified time.
moveTimeNB()	Identical to **moveJointTime()** but non-blocking.
moveTo()	Move two joints of each Linkbot to specified absolute angles.
moveToNB()	Identical to **moveTo()** but non-blocking.
moveToByTrackPos()	Move two joints to the specified positions by tracking positions.
moveToByTrackPosNB()	Identical to **moveToByTrackPos**(), but non-blocking.
moveToZero()	Instructs all motors to go to their zero positions.
moveToZeroNB()	Identical to **moveToZero()** but non-blocking.
moveWait()	Wait until all motors have stopped moving.
openGripper()	Open the gripper.
relaxJoint()	Relax a joint of the Linkbot.
relaxJoints()	Relax all the joints of the Linkbot.

Member Function Name	Description
setLEDColor()	Set the LEDs of all robots to the specified color.
setLEDColorRGB()	Set the LEDs of all robots to the specified color in RGB.
setJointSpeed()	Set a motor's speed in degrees per second.
setJointSpeeds()	Set all motor speeds in degrees per second.
setJointSpeedRatio()	Set a joint speed to a fraction of its maximum speed.
setJointSpeedRatios()	Set joint speeds to a fraction of their maximum speeds.
setSpeed()	Move the Linkbot at a constant velocity.
turnLeft()	Rotate the Linkbot counterclockwise.
turnLeftNB()	Identical to **turnLeft()** but non-blocking.
turnRight()	Rotate the Linkbot clockwise.
turnRightNB()	Identical to **turnRight()** but non-blocking.

F.5 Member Functions Using x-y Coordinates for Two-Wheel Robots

Member functions for Linkbot-I using x-y coordinates are listed in this section.

void robot.drivexy(double x, double y, double radius, double trackwidth), page 145

drives a Linkbot by x and y relative to its current position in the x-y coordinate system given the radius and track width of the Linkbot. For example,

```
robot.drivexy(3, 4, radius, trackwidth);
```

void robot.drivexyNB(double x, double y, double radius, double trackwidth), page 285

A non-blocking version of **drivexy**().

void robot.drivexyTo(double x, double y, double radius, double trackwidth), page 134

drives a Linkbot to the location of (x,y) in the x-y coordinate system given the radius and trackwidth. For example,

```
robot.drivexyTo(3, 4, radius, trackwidth);
```

void robot.drivexyToExpr(double x0, double xf, double num, double expr, double radius,

double trackwidth), page 149

drives a Linkbot based on an expression with the variable x, for x from `x0` to `xf` with `num` number of points. The expression `t expr` should be a valid Ch expression in terms of the variable `x`. The radius and track width of the robot are specified by the last two arguments. For example,

```
robot.drivexyToExpr(0, 5, 100, "2+3*x", radius, trackwidth);
```

void robot.drivexyToExprNB(double x0, double xf, double num, double expr, double radius,

double trackwidth) page 294

A non-blocking verson of drivexyToExpr().

void robot.drivexyToFunc(double x0, double xf, int num, double f(double), double radius,

double trackwidth), page 151

drives a Linkbot based on a function `func()`, for x from `x0` to `xf` with `num` number of points. The radius and track width of the Linkbot are specified by the last two arguments. For example,

```
double func(double x) {
    return 2+3*x;
}
robot.drivexyToFunc(0, 5, 100, func, radius, trackwidth);
```

void robot.drivexyToFuncNB(double x0, double xf, int num, double f(double), double radius,

double trackwidth) page 295

A non-blocking version of drivexyToFunc().

void robot.drivexyToNB(double x, double y, double radius, double trackwidth), page 283

A non-blocking version of **drivexyTo**().

void robot.drivexyWait(), page 285

pauses the program until the movement of a Linkbot by the function **drivexyNB**() or **drivexyToNB**() is finished moving. For example,

```
robot.drivexyWait();
```

int robot.getPosition(double &x, double &y, double &angle), page 138

gets the position of a Linkbot in the x-y coordinate system and stores the coordinates in variables `x`, `y`, and `angle`. For example,

```
robot.getPosition(x, y, angle);
```

int robot.getxy(double &x, double &y), page 135

gets the position of a Linkbot in the x-y coordinate system and stores the coordinates in variables `x` and `y`. For example,

```
robot.getxy(x, y);
```

int robot.initPosition(double x, double y, double angle), page 137

sets the position of a Linkbot in the x-y coordinate system and sets the angle relative to the x-axis. For example,

```
robot.initPosition(0, 0, 90);
```

int robot.recordxyBegin(double* &xdata, double* &ydata, double timeInterval), page 158

begins recording the position (x, y) in the x-y coordinate system for a Linkbot variables `xdata` and `ydata` (of type **robotRecordData_t**). `timeInterval` is the time interval between position readings. For example,

```
robot.recordxyBegin(xdata, ydata, 0.1);
```

int robot.recordxyEnd(int &numDataPoints)), page 158

ends the **recordxyBegin**() function and passes the number of data points collected in the variable `numDataPoints`. For example,

```
robot.recordxyEnd(numDataPoints);
```

F.6 Member Functions Available only in RoboSim and RoboBlockly

Member functions available only in RoboSim and RoboBlockly are listed in this section.

void robot.traceColor(string_t color, int width)
sets the color and width of the traced line for a Linkbot.

```
robot.traceColor("red", 2);
```

This feature is only available in RoboBlockly.

void robot.traceOff(), page 156
turns off tracing for a Linkbot.

```
robot.traceOff();
```

void robot.traceOn(), page 156
turns on tracing for a Linkbot.

```
robot.traceOn();
```

F.7 Member Functions Not Presented in This Book

blinkLED(), **getJointAngleInstant**(), **getJointAnglesInstant**(), **getJointSafetyAngle**(), **getJointSafetyAngleTimeout**(), **isMoving**(), **moveJointToByTrackPos**(), **moveJointToByTrackPosNB**(), **moveTimeNB**(), **moveForeverNB**(), **moveToZero**(), **moveToZeroNB**(), **relaxJoint**(), **setJointSafetyAngleTimeout**().

APPENDIX G

Common Mistakes in Writing Ch Programs

This appendix summarizes common mistakes in writing programs in Ch.

1. Divide two integers. Note that 3/2 is 1 and 3.0/2 is 1.5.
2. Miss a semicolon ';' at the end of a programming statement. For example

```
int i, j
i = 120
```

3. Miss a newline character '\n'. For example,

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

4. Use '/n' for a newline character. For example,

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

5. Use '\n' and '.2lf' in the function **scanf**(). For example,

```
scanf("%d\n", &n);     // scanf("%d", &n);
scanf("%.2lf", &var); // scanf("%lf", &var);
```

6. Miss the header file **chplot.h**

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

   for using the plotting class **CPlot**.

7. Use variables without declaring them first.
   (a) Use a variable, y1, not defined.
   (b) Use a variable of capital letter. For example,

```
Printf("Hello, world\n");
double subtotal;
Subtotal = 10;
```

8. Use the conversion specifier `"%lf"`, instead of `"%d"`, to output from or input into a variable **int** type. For example,

```
int n;
printf("n = %lf\n", n);
scanf("%lf", &n);
```

9. Use the conversion specifier `"%d"`, instead of `"%lf"`, to output from or input into a variable **double** type. For example,

```
double x;
printf("x = %d\n", n);
scanf("%d", &x);
```

10. Miss a **moveWait**() statement at the end of a program. For example,

```
robot1.driveAngleNB(360);
robot2.driveAngle(180);
/* without the statement below,
   robot1 will only drive forward 180 degrees. */
robot1.moveWait();
```

Index