



---

# Mobot User's Guide

Version 1.7



## How to Contact Barobo

Mail Barobo, Inc.  
813 Harbor Blvd, Suite 335  
West Sacramento, CA 95691-2201  
Phone + 1 916 596-3050  
Web <http://www.barobo.com>  
Email info@barobo.com

Copyright ©2012 by Barobo, Inc. All rights reserved.  
July 5, 2012

Permission is granted for users to make one copy for their own personal use. Further reproduction, or any copying of machine-readable files (including this one) to any server computer, is strictly prohibited.

Barobo, Inc. is the holder of the copyright to the Mobot software and Mobot User's Guide described in this document, including without limitation such aspects of the system as its code, structure, sequence, organization, programming language, header files, function and command files, object modules, static and dynamic loaded libraries of object modules, compilation of command and library names, interface with other languages and object modules of static and dynamic libraries. Use of the system unless pursuant to the terms of a license granted by Barobo or as otherwise authorized by law is an infringement of the copyright.

**Barobo, Inc. makes no representations, expressed or implied, with respect to this documentation, or the software it describes, including without limitations, any implied warranty merchantability or fitness for a particular purpose, all of which are expressly disclaimed. Users should be aware that included in the terms and conditions under which Barobo is willing to license the Mobot software as a provision that Barobo , and their distribution licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of the Mobot software and that liability for direct damages shall be limited to the amount of purchase price paid for Mobot and Mobot software.**

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. Barobo shall not be responsible under any circumstances for providing information on or corrections to errors and omissions discovered at any time in this documentation or the software it describes, even if Barobo has been advised of the errors or omissions.

Barobo, Mobot, iMobot, RobotController, RoboMancer, and SnapConnector are either registered trademarks or trademarks of Barobo, Inc. in the United States and/or other countries. Ch, ChIDE, and SoftIntegration are trademarks of SoftIntegrtation, Inc. Microsoft, MS-DOS, Windows, Windows 2000, Windows XP, Windows Vista, and Windows 7 are trademarks of Microsoft Corporation. Linux is a trademark of Linus Torvalds. Mac OS X and Darwin are trademarks of Apple Computers, Inc. All other trademarks belong to their respective holders.

# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
<b>2</b>	<b>Configuring Mobots for Remote Control</b>	<b>11</b>
2.1	Bluetooth Pairing for Mac OS X . . . . .	12
2.2	Adding Bluetooth Addresses of Robots in RoboMancer. . . . .	15
<b>3</b>	<b>The “Motion Control” Tab</b>	<b>19</b>
3.1	The Mobot Diagram and “Move To Zero” Button . . . . .	19
3.2	Individual Joint Control . . . . .	20
3.3	Rolling Control . . . . .	20
3.4	Joint Speeds . . . . .	20
3.5	Joint Positions . . . . .	20
3.5.1	Joint Limits . . . . .	20
3.6	Motions . . . . .	20
<b>4</b>	<b>The “Pose Teaching” Tab</b>	<b>21</b>
<b>5</b>	<b>Getting Started with Programming the Mobot</b>	<b>21</b>
5.1	start.ch, A Basic Ch Mobot Program . . . . .	21
5.1.1	start.ch Source Code . . . . .	21
5.1.2	Demo Code for start.ch Explained . . . . .	22
5.2	returnval.ch, A Basic Ch Mobot Program Which Checks Return Values . . . . .	23
5.2.1	Source Code . . . . .	23
5.2.2	returnval.ch Explained . . . . .	24
5.3	getJointAngle.ch, A Basic Ch Mobot Program Which Retrieves a Joint Angle . . . . .	25
5.3.1	Source Code . . . . .	25
5.3.2	getJointAngle.ch Explained . . . . .	25
<b>6</b>	<b>Controlling the Speed of Mobot Joints</b>	<b>26</b>
6.1	setspeed.ch Source Code . . . . .	26
6.2	setspeed.ch Source Code Explanation . . . . .	26
<b>7</b>	<b>Preprogrammed Motions</b>	<b>28</b>
7.1	inchworm.ch: A Demo using the motionInchwormLeft() Preprogrammed Motion . . . . .	29
7.1.1	inchworm.ch Source Code . . . . .	29
7.1.2	inchworm.ch Explained . . . . .	29
7.2	stand.ch: A Demo Using the motionStand() Preprogrammed Motion . . . . .	30
7.2.1	stand.ch Source Code . . . . .	30
7.2.2	stand.ch Explained . . . . .	30
7.3	tumble.ch: A Demo Using the motionTumbleLeft() Preprogrammed Motion . . . . .	30
7.3.1	tumble.ch Source Code . . . . .	30
7.3.2	tumble.ch Explained . . . . .	31
7.4	motion.ch: A Demo Using Multiple Preprogrammed Motions . . . . .	31
7.4.1	motion.ch Source Code . . . . .	31
7.4.2	motion.ch Explained . . . . .	32

<b>8 Detailed Examples of Preprogrammed Motions and Writing Customized Motions</b>	<b>32</b>
8.1 Inchworm Gait Demo . . . . .	33
8.1.1 <code>inchworm2.ch</code> Source Code . . . . .	33
8.1.2 Demo Code for <code>inchworm2.ch</code> Explained . . . . .	33
8.2 Standing Demo . . . . .	34
8.2.1 <code>stand2.ch</code> Source Code . . . . .	34
8.2.2 <code>stand2.ch</code> Explained . . . . .	35
<b>9 Blocking and Non-Blocking Functions</b>	<b>36</b>
9.1 List of Blocking Movement Functions . . . . .	36
9.2 List of Non-Blocking Movement Functions . . . . .	37
9.3 Blocking and Non-Blocking Demo Programs . . . . .	38
9.3.1 <code>nonblock.ch</code> Source Code . . . . .	38
9.3.2 <code>nonblock.ch</code> Source Code Explanation . . . . .	38
9.3.3 <code>nonblock2.ch</code> Source Code . . . . .	38
9.3.4 <code>nonblock2.ch</code> Source Code Explanation . . . . .	39
9.3.5 <code>nonblock3.ch</code> Source Code . . . . .	39
9.3.6 <code>nonblock3.ch</code> Source Code Explanation . . . . .	39
9.4 Preprogrammed Motion Demos with Non-Blocking Functions . . . . .	40
9.4.1 <code>unstand2.ch</code> Source Code . . . . .	40
9.4.2 <code>unstand2.ch</code> Source Code Explanation . . . . .	40
9.4.3 <code>tumble2.ch</code> Source Code . . . . .	41
9.4.4 <code>tumble2.ch</code> Source Code Explanation . . . . .	41
<b>10 Controlling Multiple Modules</b>	<b>43</b>
10.1 <code>twoModules.ch</code> Source Code . . . . .	43
10.2 Demo Explanation . . . . .	43
10.3 Controlling Multiple Connected Modules . . . . .	45
10.3.1 <code>lift.ch</code> , Lifting Demo . . . . .	45
10.3.2 <code>lift.ch</code> Source Code Explanation . . . . .	46
<b>11 Commanding Multiple Mobots to Perform Identical Tasks</b>	<b>47</b>
11.1 Demo program <code>group.ch</code> . . . . .	48
11.1.1 Source Code . . . . .	48
11.1.2 Demo Explanation . . . . .	49
11.2 Demo Program <code>groups.ch</code> . . . . .	49
11.2.1 <code>groups.ch</code> Source Code . . . . .	49
11.2.2 Demo Explanation . . . . .	51
<b>12 Data Acquisition, Data Processing, and Application Examples for Learning Algebra</b>	<b>52</b>
12.1 Example 1 . . . . .	52
12.1.1 Problem Statement . . . . .	52
12.1.2 <code>dataAcquisition.ch</code> Source Code . . . . .	53
12.1.3 <code>dataAcquisition.ch</code> Explained . . . . .	54
12.1.4 <code>dataAcquisition1.ch</code> Source Code . . . . .	59
12.2 Example 2 . . . . .	60
12.2.1 Problem Statement . . . . .	60
12.2.2 <code>dataAcquisition2.ch</code> Source Code . . . . .	61
12.2.3 <code>dataAcquisition2.ch</code> Explained . . . . .	62
12.3 Example 3 . . . . .	64
12.3.1 Problem Statement . . . . .	64
12.3.2 <code>dataAcquisition3.ch</code> Source Code . . . . .	64

12.3.3 <code>dataAcquisition3.ch</code> Explained . . . . .	65
<b>13 Using the On-Board Buttons</b>	<b>68</b>
13.1 Running the On-Board Joint Test program . . . . .	68
13.2 Recalibrating the Mobot's Zero Position . . . . .	68
13.3 Testing the Mobot's Zero Position . . . . .	69
<b>A CMobot Class</b>	<b>70</b>
A.1 Data Types . . . . .	70
A.2 <code>mobotJointId_t</code> . . . . .	70
A.3 <code>mobotJointState_t</code> . . . . .	70
A.4 CMobot API Functions . . . . .	70
<b>blinkLED()</b>	<b>74</b>
<b>connect()</b>	<b>74</b>
<b>connectWithAddress()</b>	<b>75</b>
<b>disconnect()</b>	<b>75</b>
<b>driveJointToDirect()</b>	<b>76</b>
<b>driveJointToDirectNB()</b>	<b>76</b>
<b>driveToDirect()</b>	<b>76</b>
<b>driveToDirectNB()</b>	<b>76</b>
<b>getJointAngle()</b>	<b>77</b>
<b>getJointMaxSpeed()</b>	<b>78</b>
<b>getJointSafetyAngle()</b>	<b>78</b>
<b>getJointSafetyAngleTimeout()</b>	<b>79</b>
<b>getJointSpeed()</b>	<b>79</b>
<b>getJointSpeedRatio()</b>	<b>80</b>
<b>getJointSpeedRatios()</b>	<b>80</b>
<b>getJointSpeeds()</b>	<b>81</b>
<b>getJointState()</b>	<b>81</b>
<b>isConnected()</b>	<b>82</b>
<b>isMoving()</b>	<b>82</b>
<b>motionArch()</b>	<b>83</b>
<b>motionArchNB()</b>	<b>83</b>

<b>motionInchwormLeft()</b>	84
<b>motionInchwormLeftNB()</b>	84
<b>motionInchwormRight()</b>	84
<b>motionInchwormRightNB()</b>	84
<b>motionRollBackward()</b>	85
<b>motionRollBackwardNB()</b>	85
<b>motionRollForward()</b>	85
<b>motionRollForwardNB()</b>	85
<b>motionSkinny()</b>	86
<b>motionSkinnyNB()</b>	86
<b>motionStand()</b>	87
<b>motionStandNB()</b>	87
<b>motionTumbleLeft()</b>	87
<b>motionTumbleLeftNB()</b>	87
<b>motionTumbleRight()</b>	88
<b>motionTumbleRightNB()</b>	88
<b>motionTurnLeft()</b>	89
<b>motionTurnLeftNB()</b>	89
<b>motionTurnRight()</b>	89
<b>motionTurnRightNB()</b>	89
<b>motionUnstand()</b>	90
<b>motionUnstandNB()</b>	90
<b>motionWait()</b>	91
<b>move()</b>	91
<b>moveNB()</b>	91
<b>moveContinuousNB()</b>	92
<b>moveContinuousTime()</b>	93
<b>moveJoint()</b>	93

<b>moveJointNB()</b>	93
<b>moveJointContinuousNB()</b>	94
<b>moveJointContinuousTime()</b>	95
<b>moveJointTo()</b>	96
<b>moveJointToNB()</b>	96
<b>moveJointToAbs()</b>	97
<b>moveJointToAbsNB()</b>	97
<b>moveJointToDirect()</b>	98
<b>moveJointToDirectNB()</b>	98
<b>moveJointWait()</b>	99
<b>moveTo()</b>	99
<b>moveToNB()</b>	99
<b>moveToAbs()</b>	100
<b>moveToAbsNB()</b>	100
<b>moveToDirect()</b>	101
<b>moveToDirectNB()</b>	101
<b>moveToZero()</b>	102
<b>moveToZeroNB()</b>	102
<b>moveWait()</b>	103
<b>recordAngle()</b>	103
<b>recordAngleBegin()</b>	104
<b>recordAngleEnd()</b>	104
<b>recordAngles()</b>	105
<b>recordAnglesBegin()</b>	106
<b>recordAnglesEnd()</b>	107
<b>recordWait()</b>	107
<b>setJointSafetyAngle()</b>	108
<b>setJointSafetyAngleTimeout()</b>	108

setJointSpeed()	109
setJointSpeedRatio()	109
setJointSpeedRatios()	110
setJointSpeeds()	110
setTwoWheelRobotSpeed()	111
stopAllJoints()	111
stopOneJoint()	111
stopTwoJoints()	111
stopThreeJoints()	111
<b>B CMobotGroup API</b>	<b>112</b>
addRobot()	115
motionArch()	115
motionArchNB()	115
motionInchwormLeft()	116
motionInchwormLeftNB()	116
motionInchwormRight()	116
motionInchwormRightNB()	116
motionRollBackward()	117
motionRollBackwardNB()	117
motionRollForward()	117
motionRollForwardNB()	117
motionSkinny()	118
motionSkinnyNB()	118
motionStand()	119
motionStandNB()	119
motionTumbleLeft()	119
motionTumbleLeftNB()	119
motionTumbleRight()	120

<b>motionTumbleRightNB()</b>	<b>120</b>
<b>motionTurnLeft()</b>	<b>121</b>
<b>motionTurnLeftNB()</b>	<b>121</b>
<b>motionTurnRight()</b>	<b>121</b>
<b>motionTurnRightNB()</b>	<b>121</b>
<b>motionUnstand()</b>	<b>122</b>
<b>motionUnstandNB()</b>	<b>122</b>
<b>motionWait()</b>	<b>122</b>
<b>move()</b>	<b>123</b>
<b>moveNB()</b>	<b>123</b>
<b>moveContinuousNB()</b>	<b>124</b>
<b>moveContinuousTime()</b>	<b>124</b>
<b>moveJoint()</b>	<b>125</b>
<b>moveJointNB()</b>	<b>125</b>
<b>moveJointContinuousNB()</b>	<b>126</b>
<b>moveJointContinuousTime()</b>	<b>127</b>
<b>moveJointTo()</b>	<b>128</b>
<b>moveJointToNB()</b>	<b>128</b>
<b>moveJointWait()</b>	<b>128</b>
<b>moveTo()</b>	<b>129</b>
<b>moveToNB()</b>	<b>129</b>
<b>moveToZero()</b>	<b>130</b>
<b>moveToZeroNB()</b>	<b>130</b>
<b>moveWait()</b>	<b>130</b>
<b>setJointSpeed()</b>	<b>131</b>
<b>setJointSpeedRatio()</b>	<b>131</b>
<b>setJointSpeedRatios()</b>	<b>132</b>
<b>setJointSpeeds()</b>	<b>132</b>

setTwoWheelRobotSpeed()	133
stop()	134
C Miscellaneous Utility Functions	134
angle2distance()	135
deg2rad()	135
delay()	136
distance2angle()	136
processRecordedAngles()	137
rad2deg()	137
shiftTime()	138

# 1 Introduction

The Mobot is a breakthrough modular robot. A single Mobot module is a fully functional robot capable of performing many possible motions. The Mobot can also be used as a building block to create robots with different geometric configurations. This documentation introduces the basic computer setup required for controlling the Mobot, as well as several demo programs and a complete reference for all API function provided with the `CMobot` and `CMobotGroup` library.

The `CMobot` library is a collection of functions geared towards controlling the motors and reading sensor values of a Mobot module via the Bluetooth wireless protocol. The functions are designed to be intuitive and easy to use. Various functions are provided to control or obtain the speed, direction, and position of the motors. The API includes C++ classes called `CMobot` and `CMobotGroup` to facilitate control of single and multiple Mobs.

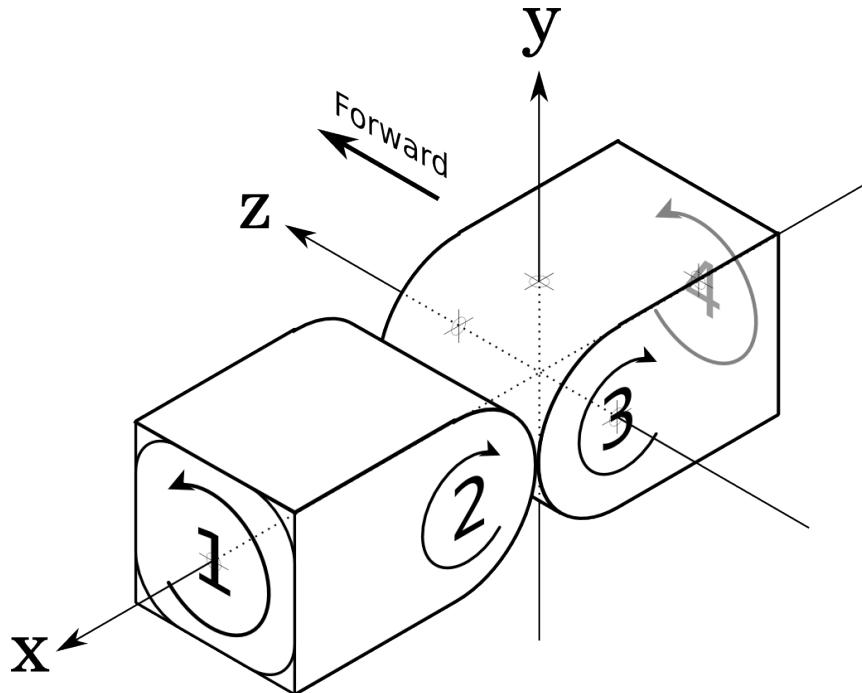


Figure 1: A schematic diagram of a Mobot module.

Figure 1 shows a schematic diagram displaying the locations and positive directions of the four joints of a Mobot module. The joints 1 and 4 shown in the figure are fully rotational and have no joint limits. Joints 2 and 3, however, can only move in the range -90 to +90 degrees.

## 2 Configuring Mobs for Remote Control

Mobot modules should be configured the first time they are used with a new computer. The process informs the computer which Mobs it is allowed to connect to. This is also necessary for certain functions in the `CMobot` API, such as `connect()`, to determine which mobs to connect to.

The configuration is performed through the Barobo RoboMancer program. The remainder of the section contains step-by-step instructions and screenshots showing how to configure your Mobs.

## 2.1 Bluetooth Pairing for Mac OS X

For the Mac OS X operating system, an additional initial step needs to be performed. Windows users may skip these steps directly to section 2.2 on page 15.

To begin the pairing process, click on the Bluetooth icon on the applet tray, typically located on the top right portion of the screen. Figure 2 shows the Bluetooth icon on a typical Mac OS X system. Click on the icon to drop down a menu.



Figure 2: The Bluetooth Icon on a Mac OS X System.

The menu that is dropped down should appear similar to the one shown in Figure 3. Next, click on the menu item labeled “Set Up Bluetooth Device...”.

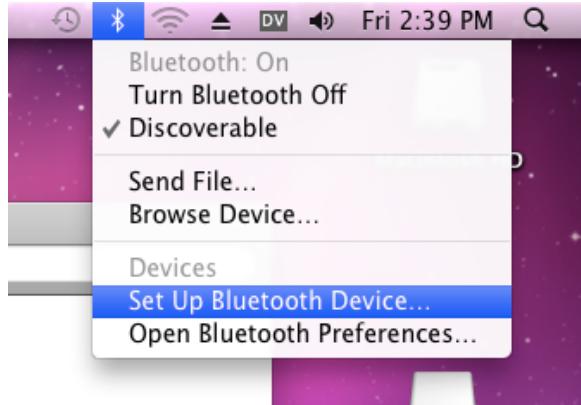


Figure 3: The Bluetooth Icon Dialog.

The previous step should start a dialog used to find and pair with Bluetooth devices. Make sure the Mobot is turned on, and it should appear as a device in the dialog, as shown in Figure 4. If there are multiple Mobot devices, as shown in the figure, you may pinpoint the specific Mobot to pair with by looking inside the battery compartment on the side of the Mobot with the power switch. Inside, there is a sticker with the Mobot’s unique ID number. The last four digits of the ID number will coincide with the Mobot’s Bluetooth ID number. Once the correct Mobot is selected, click on the “Continue” button.

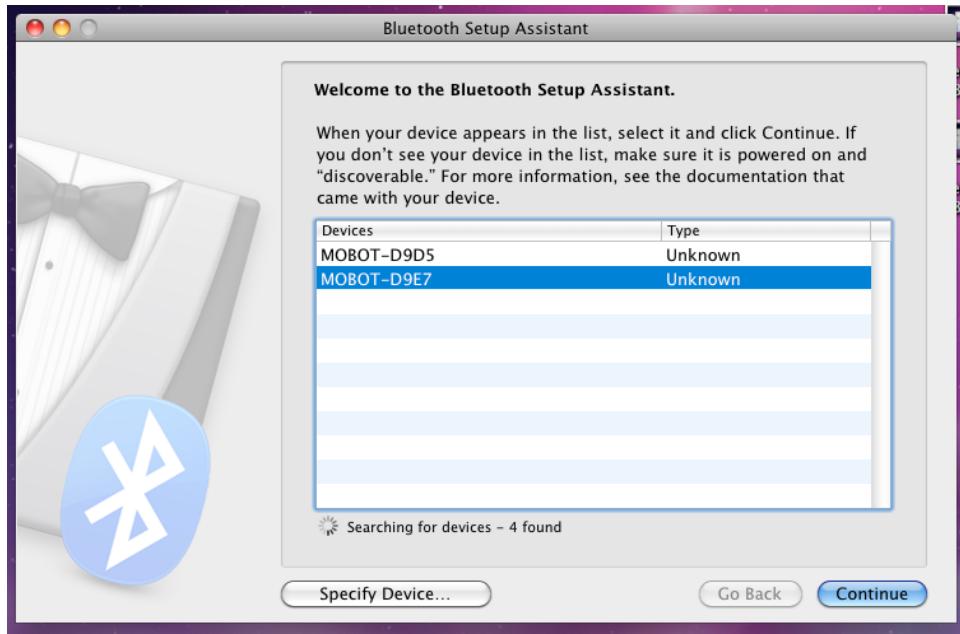


Figure 4: The Bluetooth Pairing Dialog.

At this point, the pairing dialog will attempt to pair with the Mobot using a default pairing key, “0000”, as shown in Figure 5. This initial pairing attempt will fail because the Mobot pairing key is hard-coded to a value of “1234”. Click on the button labeled “Passcode Options...” in order to modify the passcode options. This should bring up the dialog shown in Figure 6.

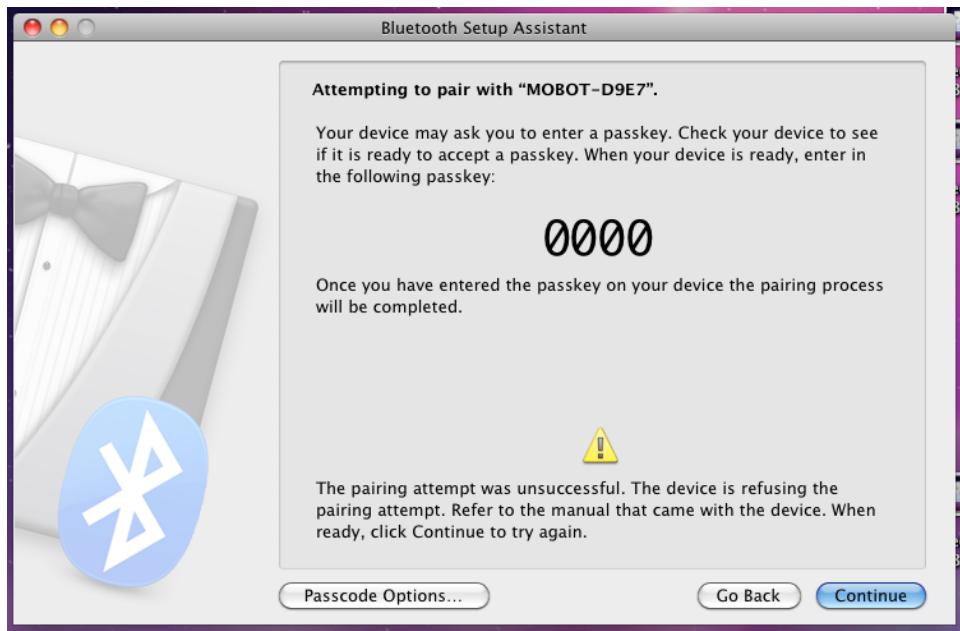


Figure 5: The Bluetooth Pairing Dialog.

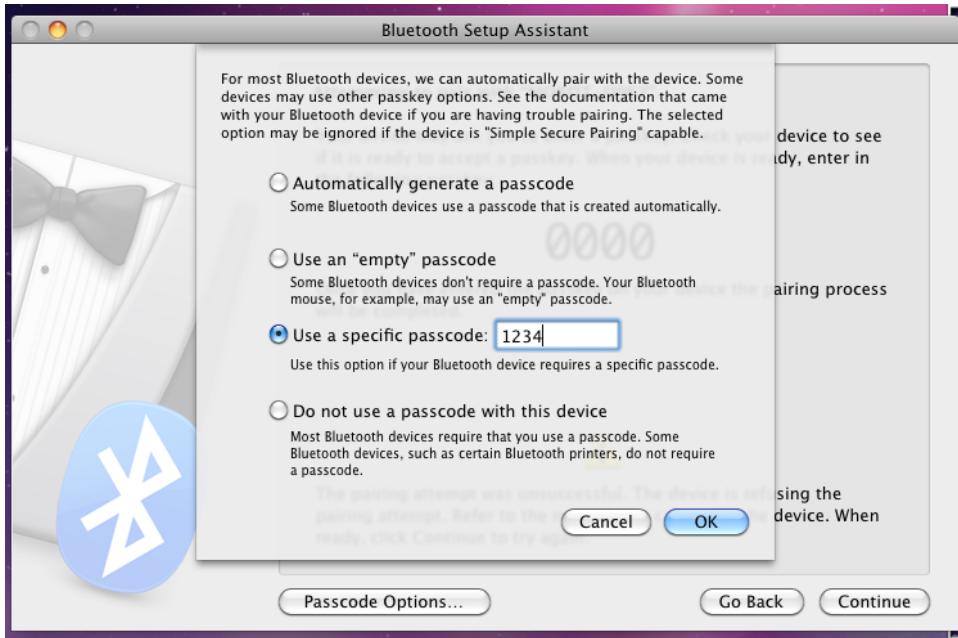


Figure 6: The Bluetooth Pairing Dialog.

In the dialog shown in Figure 6, select the option to “use a specific passcode”, and enter the value “1234”. Next, click on the ”OK” button.

You should be greeted with the dialog shown in Figure 7, which indicates that the Mobot has successfully paired. At this point, you may continue on with the rest of the pairing process, or repeat this step for all Mobots that need to be paired before continuing.



Figure 7: The Bluetooth Pairing Success Dialog.

## 2.2 Adding Bluetooth Addresses of Robots in RoboMancer.

First, start the provided Barobo RoboMancer application. In Windows, start the provided Barobo Robot Control Program by clicking on the icon labeled “RoboMancer” on your desktop, as shown in 8. On Mac OS X systems, the RoboMancer application is located inside the “Applications” folder in Finder. The control dialog as shown in Figure 9 should pop up.



Figure 8: The icon for Barobo RoboMancer.

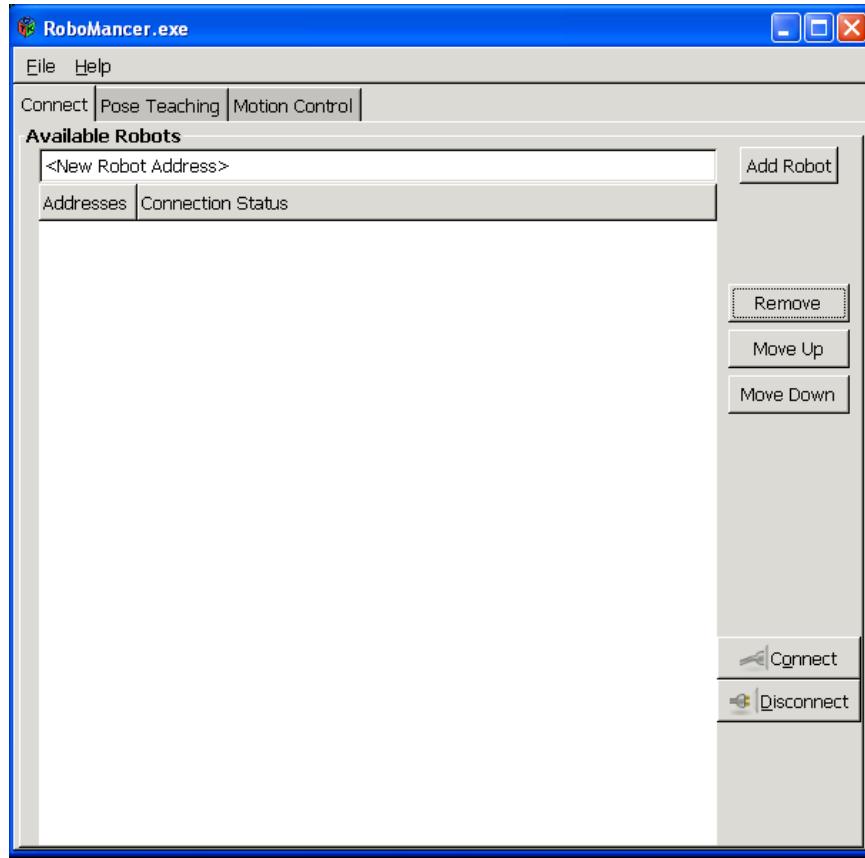


Figure 9: The opening screen for RoboMancer.

The RoboMancer application, as seen in Figure 9, is organized into several different sections, each denoted by a tab. Upon startup, the default tab that is selected is the “Connect” tab. This section of the application allows you to manage, connect, and disconnect from Mobots.

First, add your Mobot address(es) into robomancer by typing in the address and clicking on the “Add Robot” button. The dialog should now something like the figure shown in Figure 10.

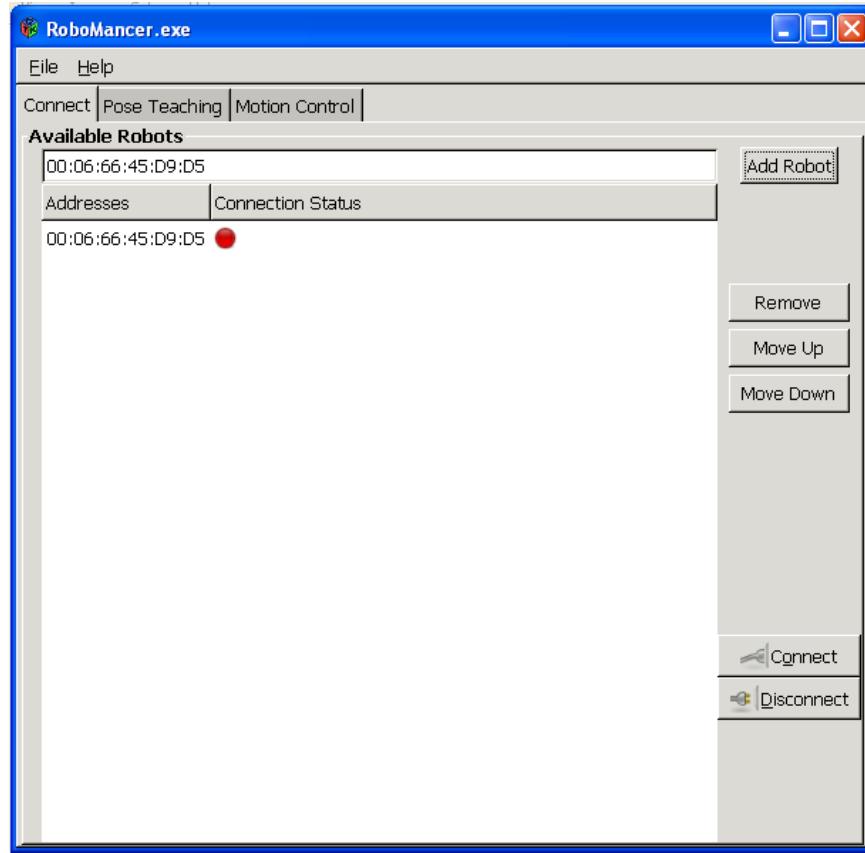


Figure 10: Configuring robot bluetooth connection.

Once new addresses are added, RoboMancer will remember the addresses in the future. The red dot next to the added address in the “Connection Status” column indicates the current connection status of the robot. The red dot indicates that the Mobot is not currently connected, and a green dot indicates that the Mobot is properly connected.

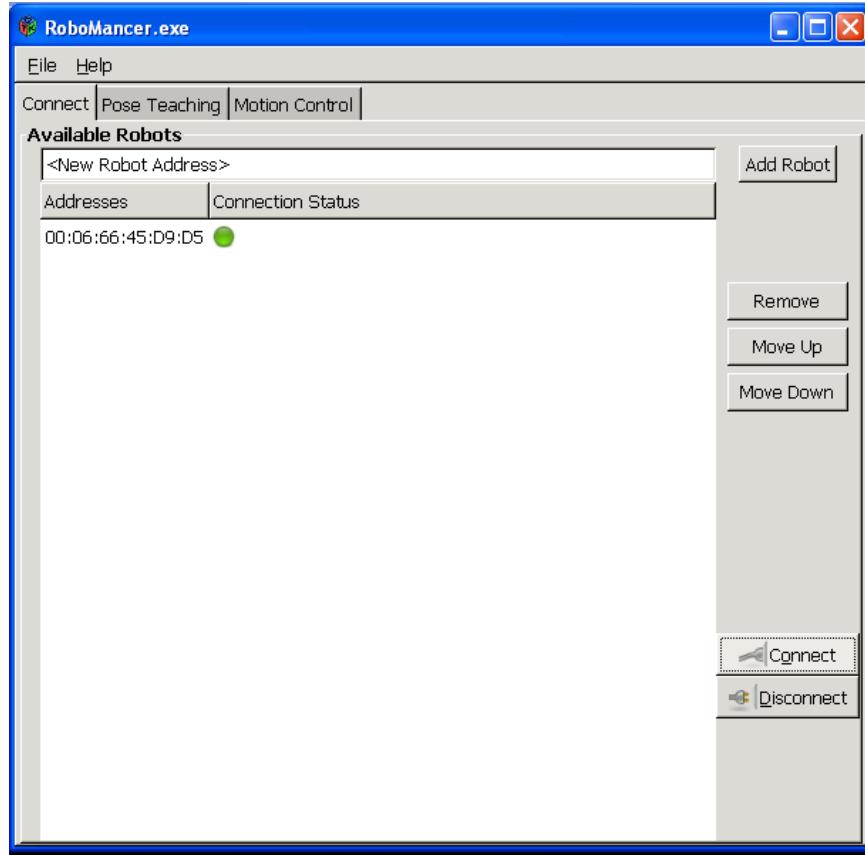


Figure 11: The Connection dialog showing a connected Mobot.

To connect to your Mobot(s), select the Mobot you wish to connect to and click on the button labelled “Connect”. If the connection succeeds, the red dot should turn green to indicate that a connection has been established, as shown in Figure 11. Should the connection fail, please check that the address has been entered correctly, the Mobot is on, your computer has Bluetooth capability, and that the Mobot is within range of your computer, and try again.

Furthermore, please note that Bluetooth devices have a maximum limit of connected devices. The maximum limit is 7 devices connected simultaneously. This means that a maximum of 7 Mobots may be connected to a computer simultaneously, if no other devices are connected. If, for instance, two other devices are currently connected, such as a bluetooth enabled phone and a bluetooth mouse, then the maximum number of connected Mobots decreases to 5, for a total of 7 connected devices.

### 3 The “Motion Control” Tab

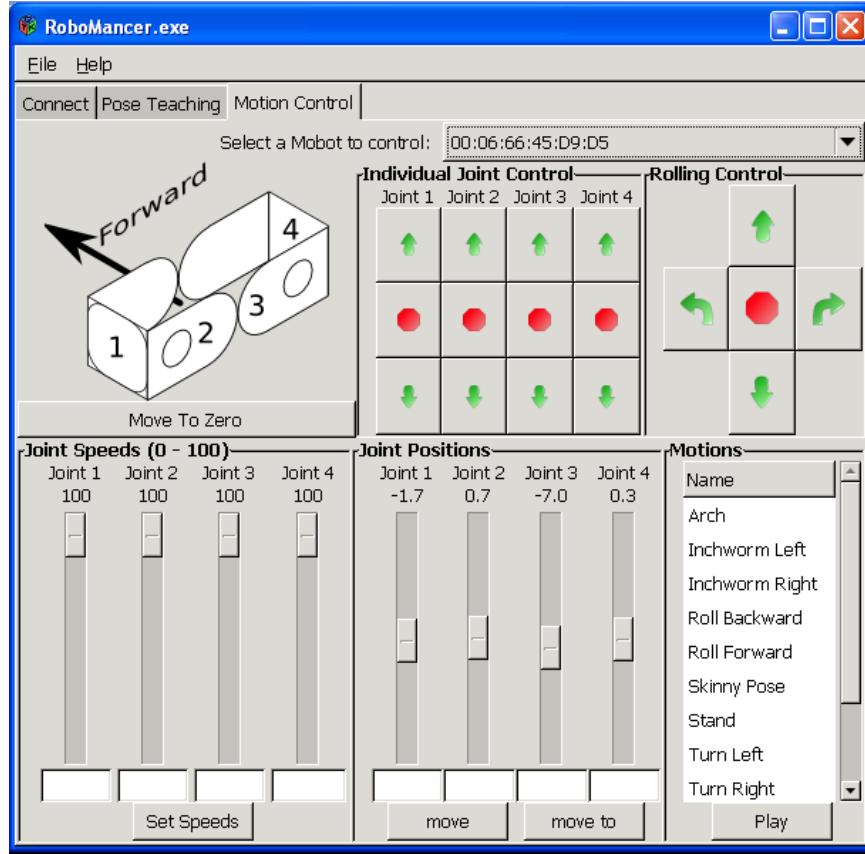


Figure 12: The graphical display of RoboMancer while connected to a robot.

Once a robot is connected to RoboMancer, the joint angles and speeds of the robot are displayed as shown in Figure 12. This dialog is located under the third tab of RoboMancer, labeled “Motion Control”. The “Motion Control” tab can be used to display information about the Mobot’s joint positions, and also control the speeds and positions of the Mobot’s joints. The interface is divided up into six sections; three on the top half of the interface, and three on the bottom half.

#### 3.1 The Mobot Diagram and “Move To Zero” Button

The first section of the GUI located on the top left of the interface displays a schematic diagram of the Mobot, displaying motor positions. Underneath the diagram, there is a large button with the text “Move To Zero”. When clicked, this button will command the connected Mobot to rotate all of its joints to a flat “Zero” position.

It is also possible to move the Mobot into zero position using the on-board buttons. Please refer to Section 13.3 on page 69 for more details.

## 3.2 Individual Joint Control

The second section, located at the top-middle section of the interface, is the “Individual Joint Control” section. These buttons command the Mobot to move individual joints. When the up or down arrows are clicked, the Mobot 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.

If the joint encounters any obstacle that prevents it from moving, the joint will automatically disengage power to the joint. This may happen, example, if a body joint attempts to rotate beyond its limits, or if it collides with the other corresponding body joint.

## 3.3 Rolling Control

This section contains buttons for controlling the Mobot as a two wheeled mobile mobot. The up and down buttons cause the Mobot to roll forward or backward. The left and right buttons cause the Mobot to rotate towards the left, or towards the right. The stop button in the middle causes the Mobot to stop where it is.

## 3.4 Joint Speeds

The “Joint Speeds” section, located at the bottom left of the interface, displays and controls the current joint speeds of the Mobot. The joint speeds are in units of degrees per second. To set a specific desired joint speed for a particular joint, the joint speed may be typed directly into the edit boxes below the sliders, and the “Set” button should be clicked.

## 3.5 Joint Positions

This section, located in the bottom-middle of the interface, is used to display and control the positions of each of the four joints of a Mobot. The joint positions are displayed in the numerical text located above each vertical slider. The displayed joint positions are in units of degrees.

The method of controlling the joints is by using the vertical sliders. Each vertical slider’s position represents a joint’s angle. The sliders for the two end joints vary from -180 degrees to 180 degrees, representing one complete rotation. The angles for the two body joints vary from -90 to 90 degrees. When the position of the slider is moved, the Mobot will move its joints to match the sliders.

Underneath the sliders, there are four text entry boxes. The text boxes accept specific angles for each joint which the user may type in. If the “MoveTo” button is clicked, each joint will move to their respective desired absolute positions. If any text entry is left blank, the corresponding joint will not move.

If the “Move” button is clicked, the program treats the angles entered by the user as a relative amount to move. For instance, if the value “360” is entered into the box for joint 1 and “Move” is clicked, the joint will rotate one full rotation, no matter where the joint was when the motion began.

### 3.5.1 Joint Limits

Joints 1 and 4 are fully rotational and have no joint limits. Joints 2 and 3, however, are limited to a range of -90 to +90 degrees.

## 3.6 Motions

This section, located on the bottom right of the interface, contains a set of preprogrammed motions for the Mobot. To execute a preprogrammed motion, simply click on the name of the motion you wish to execute, and then click the button labeled “Play”.

## 4 The “Pose Teaching” Tab

“Pose Teaching” is the process of training a robot to perform motions by physically moving the robot into desired positions. To pose teach a robot or multiple robots, begin by connecting to all of the robots that will be trained by pose teaching in the “Connect” tab. Once all of the desired mobots are connected, switch to the “Pose Teaching” tab. Once you have switched tabs, the function of the mobot’s on-board buttons will change to Pose Teaching mode. In Pose Teaching mode, the mobot buttons have the following functions:

- Button A: Record the current positions of all connected mobots.
- Button B: Play the current recorded motions in order, or stop if motions are currently being played.
- Buttons A and B together: Clear all recorded motions

Each time a pose is recorded, an indicator for that pose will appear in the dialog. As the poses are played, the pose icon changes for each pose to indicate the current pose being played.

Delays may be added between poses by using the “Add Delay” button. The length of the delay (in seconds) should be entered in the text box next to the “Add Delay” button.

It is also possible to save the recorded motion as a program by clicking on the “Save to Program” button. The generated program is a standard Ch Mobot program which may be executed in ChIDE.

## 5 Getting Started with Programming the Mobot

RoboMancer can be used to control the Mobot for simple tasks and applications. For more complicated applications, computer programs are better suited for controlling the Mobot. Mobot can be controlled using a C/C++ program through Ch, a C/C++ interpreter. Ch Professional Edition or Ch Student Edition software is required to run the demo programs for controlling Mobot. Ch is available from SoftIntegration, Inc. at <http://www.softintegration.com>

Before the Ch program is executed, the Bluetooth addresses of the robots need to be added using RoboMancer as described in Section 2.

To help the user become acquainted with the Mobot control programs, sample programs will be presented in this section to illustrate the basics and minimum requirements of a Mobot control program. The sample programs are located at CHHOME/package/chmobot/demos, where CHHOME is the Ch home directory, such as C:\Ch for Windows. For Windows, it is located at C:\Ch\package\chmobot\demos by default. Alternatively, the demos may be accessed in RoboMancer application by clicking on the “Help → Demos” menu item, as shown in Figure 13.

The first demo presents a minimal program which connects to a Mobot and moves joints 1 and 4.

### 5.1 start.ch, A Basic Ch Mobot Program

#### 5.1.1 start.ch Source Code

```
/* Filename: start.ch
 * Move the mobot faceplates. */
#include <mobot.h>

CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();
```

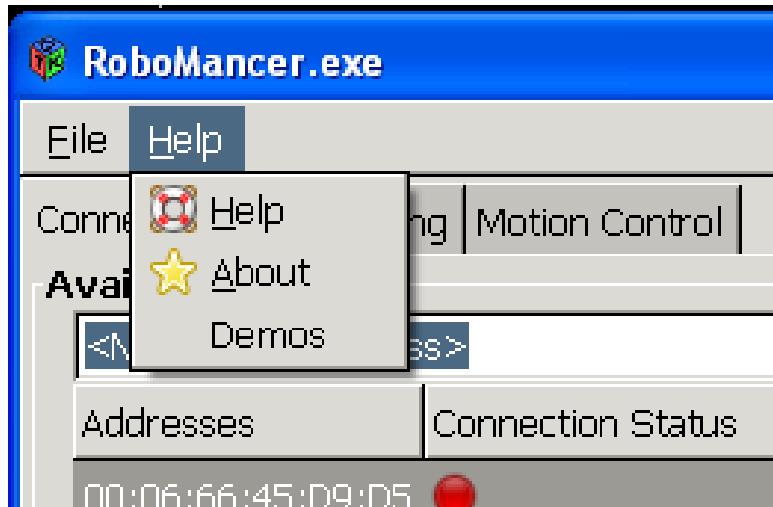


Figure 13: The “Help → Demos” menu item on the RoboMancer Application.

```
/* Rotate each of the faceplates by 360 degrees */  
mobot.move(360, 0, 0, 360);
```

### 5.1.2 Demo Code for start.ch Explained

The beginning of every Mobot control program will include header files. Each header file imports functions used for a number of tasks, such as printing data onto the screen or controlling the Mobot. The `mobot.h` header file must be included in order to use the `CMobot` class and related mobotic control functions.

```
#include <mobot.h> // Required for Mobot control functions
```

Next, we must initialize the C++ class used to control the Mobot.

```
CMobot mobot;
```

This line initializes a new variable named `mobot` which represents the remote Mobot module which we wish to control. This special variable is actually an instance of the `CMobot` class, which contains its own set of functions called “methods”, “member functions”, or simply “functions”.

The next line,

```
mobot.connect();
```

will connect our new variable, `mobot`, to a Mobot that has been previously configured with the computer in the process described in Section 2.

Note that there are two common methods to connect to a remote Mobot. The most common method, demonstrated in the previous line of code, is used to connect to a Mobot that is already paired to the computer. It is also possible to connect to Mobots which are not paired with the computer. This method is necessary for connecting to multiple Mobots simultaneously, as only a single Mobot may be paired with the computer at a time. The second method uses the function `connectWithAddress()`, and its default usage is as such:

```
string_t address = "11:22:33:44:55:66";
int defaultChannel = 1;
mobot.connectWithAddress(address, defaultChannel);
```



Figure 14: The mobot in zero position

The string "11:22:33:44:55:66" represents the Bluetooth address of the Mobot, which must be known in advance. The channel number 1 represents the Bluetooth channel to connect to. Channel 1 is the default channel Mobots listen on for incoming connections, but may be set to other values depending on the type of mobot. Detailed documentation for each of the Mobot functions, such as `connect()` and `connectAddress()`, are presented in Appendix A.4.

The next line,

```
mobot.moveToZero();
```

uses the `moveToZero()` member function. The `moveToZero` function causes the Mobot to move all of its motors to the zero position, as shown in Figure 14.

The next line of code command joints 1 and 4 to rotate 360 degrees.

```
mobot.move(360, 0, 0, 360);
```

Note that the member function `move()` expects input angles in degrees, so the angles in radians must be first be converted to degrees using the `rad2deg()` function. The `rad2deg()` function takes an angle in radians as its argument and returns the angle in degrees. The function is implemented in Ch with the code

```
#include <math.h> /* For M_PI */
double rad2deg(double radians)
{
    double degrees;
    degrees = radians * 180.0 / M_PI;
    return degrees;
}
```

If desired, values in radians may also be converted to degrees using the counterpart function, `deg2rad()`. Joints 1 and 4 are the faceplates of the Mobot which are sometimes used to act as "wheels".

## 5.2 returnval.ch, A Basic Ch Mobot Program Which Checks Return Values

### 5.2.1 Source Code

```
/* Filename: returnval.ch
 * Rotate the faceplates by 90 degrees */
#include <mobot.h>
```

```

CMobot mobot;
double angle1, angle4;

/* Connect to the paired Mobot */
if(mobot.connect())
{
    printf("Failed to connect to the mobot.\n");
    exit(-1);
}
/* Set the mobot to "home" position, where all joint angles are 0 degrees.*/
mobot.moveToZero();

/* Rotate each of the faceplates by 360 degrees */
angle1 = 360;
angle4 = 360;
mobot.move(angle1, 0, 0, angle4);
/* Move the motors back to where they were */
angle1 = -360;
angle4 = -360;
mobot.move(angle1, 0, 0, angle4);

```

### 5.2.2 returnval.ch Explained

The first portion of the code, the lines

```
#include <mobot.h>
CMobot mobot;
```

set up our program for controlling mobots as seen in previous demos. The next line,

```
double angle1, angle4;
```

declares two variables that will be used to hold angle values later in the program.

The next lines, which connect to the mobot, appear as such:

```
if(mobot.connect())
{
    printf("Failed to connect to the mobot.\n");
    exit(0);
}
```

This section connects to the remote mobot as in previous examples, but also does some error checking. The majority of the CMobot member functions return an integer value indicating whether or not the function succeeded. The CMobot member functions return 0 if they succeed, and -1 if any type of error has occurred. Errors may occur for any number of reasons, including lost connections, mechanical failure, and electrical interference. The demos up to this point have ignored the return values of the CMobot member functions.

Next following lines,

```
/* Set the mobot to "home" position, where all joint angles are 0 degrees.*/
mobot.moveToZero();

/* Rotate each of the faceplates by 360 degrees */
angle1 = 360;
angle4 = 360;
```

```

mobot.move(angle1, 0, 0, angle4);
/* Move the motors back to where they were */
angle1 = -360;
angle4 = -360;
mobot.move(angle1, 0, 0, angle4);

```

move the mobot into its zero position, rotates its end plates by one full rotation, and then rotates the end plates back to their original position. Unlike the previous demo, this demo uses variables to store the joint angle values. The variables are assigned values, and then used as the function arguments for the `move()` function.

### 5.3 getJointAngle.ch, A Basic Ch Mobot Program Which Retrieves a Joint Angle

#### 5.3.1 Source Code

```

/* Filename: getJointAngle.ch
 * Find the current joint angle of a joint. */
#include <mobot.h>
CMobot mobot;

/* Connect to a mobot */
mobot.connect();

/* Get the joint angle of the first joint */
double angle;
mobot.getJointAngle(MOBOT_JOINT1, angle);

/* Print out the joint angle */
printf("The current joint angle for joint 1 is %lf degrees.\n", angle);

```

#### 5.3.2 getJointAngle.ch Explained

The first portion of the program,

```

#include <mobot.h>
CMobot mobot;

/* Connect to a mobot */
mobot.connect();

```

initialize the `mobot` variable and connect to the remote mobot, as shown in the previous demo. Next, the line

```
double angle;
```

initializes a new variable called `angle`, which will be used to store the current angle of one of the mobotic joints. The next line,

```
mobot.getJointAngle(MOBOT_JOINT1, angle);
```

retrieves the current angle of joint 1, which is one of the faceplates of the mobot. `MOBOT_JOINT1` is an enumerated value defined in the header file `mobot.h`. Detailed information for all enumerated values defined in `mobot.h` can be found in Appendix A.1.

Finally, the last line of the program,

```
printf("The current joint angle for joint 1 is %lf degrees.\n", angle);  
prints the value of the variable onto the screen.
```

## 6 Controlling the Speed of Mobot Joints

### 6.1 setspeed.ch Source Code

```
/* Filename: setspeed.ch  
Move the two wheeled mobot with different speed. */  
#include <mobot.h>  
#include <math.h>  
CMobot mobot;  
  
/* Connect to the paired Mobot */  
mobot.connect();  
  
/* Set the mobot to "home" position, where all joint angles are 0 degrees. */  
mobot.moveToZero();  
  
double speed, radius;  
mobot.getJointMaxSpeed(MOBOT_JOINT1, speed);  
printf("The maximum speed is %lf degrees/s\n", speed);  
  
mobot.setJointSpeed(MOBOT_JOINT1, 60);  
mobot.setJointSpeed(MOBOT_JOINT4, 60);  
  
//mobot.setJointSpeedRatio(MOBOT_JOINT1, 0.5);  
//mobot.setJointSpeedRatio(MOBOT_JOINT4, 0.5);  
  
//mobot.setJointSpeeds(60, 0, 0, 60);  
  
printf("Roll forward 360 degrees.\n");  
mobot.motionRollForward(360);  
  
speed = 1.83; // = (3.5/2) * M_PI * 60/180 (inch/s)  
radius = 3.5/2; // radius is 1.75  
mobot.setTwoWheelRobotSpeed(speed, radius);  
  
printf("Move 360 degrees.\n");  
mobot.move(360, 0, 0, 360);  
  
/* move at 1.83inch/sec with the radius 3.5 inches for 3 seconds */  
printf("Move continuously for 3 seconds.\n");  
mobot.moveContinuousTime(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_FORWARD, 3);
```

### 6.2 setspeed.ch Source Code Explanation

The first several lines,

```
#include <mobot.h>  
#include <math.h>
```

```

CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

```

initialize the program, connect to the mobot, and move the mobot into its zero position, similar to the previous demos. The next three lines,

```

double speed, radius;
mobot.getJointMaxSpeed(MOBOT_JOINT1, speed);
printf("The maximum speed is %lf degrees/s\n", speed);

```

initializes the variables `speed` and `radius`, retrieves the maximum joint speed for the first joint using the member function, `getJointMaxSpeed` and stores it in the variables named `speed`. The value of the maximum speed is then printed onto the screen using the `printf()` function.

The next two lines,

```

mobot.setJointSpeed(MOBOT_JOINT1, 60);
mobot.setJointSpeed(MOBOT_JOINT4, 60);

```

set the joint speed settings for the two faceplate joints to 90 degrees per second.

The next lines,

```

//mobot.setJointSpeedRatio(MOBOT_JOINT1, 0.5);
//mobot.setJointSpeedRatio(MOBOT_JOINT4, 0.5);

//mobot.setJointSpeeds(60, 0, 0, 60);

```

are two alternate ways of setting the joint speeds of the faceplate joints. The member function `setJointSpeedRatio()` sets the joint speeds as a ratio of the maximum speed. The function `setJointSpeeds()` is used to set all four joint speeds simultaneously. Note though, that there is a slight difference between using the `setJointSpeeds()` function as shown in this example compared to the other methods. The other methods do not alter the joint speeds for joints 2 and 3, while the `setJointSpeeds()` function used as shown in the example explicitly sets the joint speeds of joints 2 and 3 to zero.

The next two lines,

```

printf("Roll forward 360 degrees.\n");
mobot.motionRollForward(360);

```

print a message to the screen and rolls the mobot forward by rotating the faceplates 360 degrees.

The next lines,

```

speed = 1.83; // = (3.5/2) * M_PI * 60/180 (inch/s)
radius = 3.5/2; // radius is 1.75
mobot.setTwoWheelRobotSpeed(speed, radius);

```

use the `setTwoWheelRobotSpeed()` function to set the faceplate joint speeds for a mobot acting as a two wheeled car. The `setTwoWheelRobotSpeed()` function takes a desired speed and the radius of the wheels as arguments and calculates the necessary rotational speed of the faceplate wheels to achieve the desired speed. Note that the units for the speed must match the units for the radius. For instance, if the radius is provided in inches, the desired speed must be provided in inches per second. If the radius is provided in centimeters,

the speed must be provided in centimeters per second, and so on. In this example, specifying the speed of the two wheeled mobot at 1.83 inches per second is equivalent to setting the speeds of the two wheels at 60 degrees per second.

The following two lines,

```
printf("Move 360 degrees.\n");
mobot.move(360, 0, 0, 360);
```

rotate the faceplates forward at the necessary rate to achieve a forward speed of 1.83 inches per second.

Finally, the last two lines,

```
printf("Move continuously for 3 seconds.\n");
mobot.moveContinuousTime(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_FORWARD, 3);
```

roll the mobot forward for three seconds. The enumerated values MOBOT\_FORWARD and MOBOT\_BACKWARD indicate the forward and backward directions for each motor to turn, respectively. MOBOT\_NEUTRAL indicates that the motor should not turn, but should remain flexible and backdrivable. MOBOT\_HOLD indicates that the joint will not turn, and that the joint will be forcefully held in place at its current position. More information regarding these macros may be found in Section A.3. In the previous line of code, joints 1 and 4 move forward while joints 2 and 3 hold their current positions. The last argument of the function `moveContinuousTime()` specifies the duration of time to move or hold the motors in seconds.

## 7 Preprogrammed Motions

The mobot API contains functions for executing preprogrammed motions. The preprogrammed motions are motions which are commonly used for mobot locomotion. Following is a list of available functions and a brief description about their effect on the mobot.

- `motionArch()`: This function causes the mobot to arch up for better clearance.
- `motionInchwormLeft()`: This function causes the mobot to perform the inchworm gait once, moving the mobot towards its left.
- `motionInchwormRight()`: This function causes the mobot to perform the inchworm gait once, moving the mobot towards its right.
- `motionRollBackward()`: This function causes the mobot to rotate its faceplates, using them as wheels to roll backward.
- `motionRollForward()`: This function causes the mobot to rotate its faceplates, using them as wheels to roll forward.
- `motionSkinny()`: This function makes the mobot assume a skinnier rolling profile.
- `motionStand()`: This function causes the mobot to stand up onto a faceplate, assuming the camera platform position.
- `motionTumbleRight()`: This function causes the mobot to perform the tumbling motion, flipping end over end.
- `motionTumbleLeft()`: This function causes the mobot to perform the tumbling motion, flipping end over end.
- `motionTurnLeft()`: This function uses the mobot's faceplates as wheels, turning them in opposite directions in order to rotate the mobot towards its left.

- `motionTurnRight()`: This function uses the mobot's faceplates as wheels, turning them in opposite directions in order to rotate the mobot towards its right.
- `motionUnstand()`: This function causes the mobot to drop down from a standing position.

Note that all of the functions listed above are “blocking” functions, meaning they will not return until the motion has completed. These functions also have non-blocking equivalents which are discussed in Section 9.

## 7.1 inchworm.ch: A Demo using the `motionInchwormLeft()` Preprogrammed Motion

### 7.1.1 inchworm.ch Source Code

```
/* File: inchworm.ch
 * Perform the "inchworm" motion four times */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set mobot motors to speed of 0.50 */
mobot.setJointSpeedRatio(MOBOT_JOINT2, 0.50);
mobot.setJointSpeedRatio(MOBOT_JOINT3, 0.50);

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

/* Do the inchworm motion four times */
mobot.motionInchwormLeft(4);
```

### 7.1.2 inchworm.ch Explained

First, the header file `mobot.h` is included. This header file is required before usage of the `CMobot` class and its associated member functions can be used. Next, we create a variable to represent our mobot and connect to the mobot with the following lines.

```
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();
```

Next, we set the motor speeds to 50% speed with the following lines.

```
mobot.setJointSpeedRatio(MOBOT_JOINT2, 0.50);
mobot.setJointSpeedRatio(MOBOT_JOINT3, 0.50);
```

We then move the mobot to its zero position in preparation for the inchworm gait.

```
mobot.moveToZero();
```

Finally, we perform the inchworm gait four times. The argument for the function `motionInchwormLeft()` represents the number of times the gait should be performed.

```
mobot.motionInchwormLeft(4);
```

## 7.2 stand.ch: A Demo Using the motionStand() Preprogrammed Motion

This demo is a simple demonstration of the motionStand() member function.

### 7.2.1 stand.ch Source Code

```
/* Filename: stand.ch
 * Make a Mobot stand up on a faceplate */
#include <mobot.h>
CMobot mobot;

/* Connect to the Mobot */
mobot.connect();
/* Run the built-in motionStand function */
mobot.motionStand();
delay(3); // Stand still for three seconds
/* Spin the mobot around two revolutions while spinning the top faceplate*/
mobot.move(2*360, 0, 0, 2*360);
/* Lay the mobot back down */
mobot.motionUnstand();
```

### 7.2.2 stand.ch Explained

After the initialization and connection as seen in the previous demo, it executes the following line of code:

```
mobot.motionStand();
```

This line of code causes the Mobot to perform a sequence of motions causing it to stand up on a faceplate. After the mobot has stood up, the next line of code,

```
delay(3); // Stand still for three seconds
```

pauses the program for three seconds, causing the mobot to remain still for three seconds.

After the pause is over, the line

```
mobot.move(2*360, 0, 0, 2*360);
```

turns both faceplates of the mobot two full rotations, making the mobot spin in place while standing. Finally, the line

```
mobot.motionUnstand();
```

causes the mobot to drop back down into a flat position.

## 7.3 tumble.ch: A Demo Using the motionTumbleLeft() Preprogrammed Motion

### 7.3.1 tumble.ch Source Code

```
/* Filename: tumble.ch
 * Tumbling mobot */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();
/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
```

```

mobot.moveToZero();

/* Tumble two times */
mobot.motionTumbleLeft(2);

```

### 7.3.2 tumble.ch Explained

The first portion of the program,

```

/* Filename: tumble.ch
 * Tumbling mobot */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect()
/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

```

initialize the proper variables, connect to the remote mobot, and make it move to a flat zero position, similar to previous demos.

Next, we make the mobot perform the tumbling motion with the following line:

```
mobot.motionTumbleLeft(2);
```

The argument, “2”, indicates that the tumbling motion should be performed two times.

## 7.4 motion.ch: A Demo Using Multiple Preprogrammed Motions

### 7.4.1 motion.ch Source Code

```

/* Filename: motion.ch
 * Move the two wheeled mobot. */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

/* test all pre-programmed motions */
mobot.motionArch(90);
mobot.motionInchwormLeft(4);
mobot.motionInchwormRight(4);
mobot.motionRollBackward(360);
mobot.motionRollForward(360);
mobot.motionTurnLeft(360);
mobot.motionTurnRight(360);
mobot.motionStand();
mobot.move(360, 0, 0, 360);
mobot.motionUnstand();
mobot.motionTumbleLeft(2);

```

#### 7.4.2 motion.ch Explained

The first portion of the program initializes and connects to the remote mobot similar to the previous demos.

The next portion of code executes a number of different programmed motions to demonstrate the motion capabilities of the mobot. First, the “Arch” motion is demonstrated by the following line of code:

```
mobot.motionArch(15);
```

The parameter given to the function, “15” in this case, is the angle in degrees that the body joints should form in relation to each other.

The next couple lines of code,

```
mobot.motionInchwormLeft(4);  
mobot.motionInchwormRight(4);
```

make the mobot inchworm to the left four times, and then inchworm to the right four times.

Next, the lines

```
mobot.motionRollBackward(360);  
mobot.motionRollForward(360);  
mobot.motionTurnLeft(360);  
mobot.motionTurnRight(360);
```

make the mobot roll backward, forward, and then turn left, and turn right sequentially. For each of these functions, the parameter is the angle in degrees to turn the faceplates. For instance, the line rolls the mobot backward using its faceplates as wheels by rotating the faceplates 360 degrees.

Next, the mobot stands up by executing the line

```
mobot.motionStand();
```

While the mobot is standing, the line

```
mobot.move(360, 0, 0, 360);
```

rotates both faceplates one complete rotation. This causes the mobot to spin around in a circle, since it is currently standing on one of its faceplates.

Next, we lay the mobot back down into a prone position with the following line of code:

```
mobot.motionUnstand();
```

Finally, we perform the tumbling motion.

```
mobot.motionTumbleLeft(2);
```

The tumbling motion is a movement in which the mobot stands up and then flips, end over end. The argument provided to the function, “2” in this case, is the number of times to perform the motion.

## 8 Detailed Examples of Preprogrammed Motions and Writing Customized Motions

In the previous sections, preprogrammed motions have been demonstrated. In this section, the inner workings of the preprogrammed motions will be discussed, as well as various methods of designing and creating custom motions. Some more complex motions, such as “motionTumbleLeft()” and “motionUnstand()” will be discussed in Section 9.

## 8.1 Inchworm Gait Demo

The next demo will illustrate how a simple gait known as the “Inchworm” gait can be implemented.

### 8.1.1 inchworm2.ch Source Code

```
/* File: inchworm2.ch
 * Perform the "inchworm" motion four times */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set mobot motors to speed of 0.50 */
mobot.setJointSpeedRatio(MOBOT_JOINT2, 0.50);
mobot.setJointSpeedRatio(MOBOT_JOINT3, 0.50);

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

/* Do the inchworm motion four times */
int i, num = 4;
double angle2 = -45;
double angle3 = 45;
for(i = 0; i < num; i++) {
    mobot.moveJointTo(MOBOT_JOINT2, angle2); /* Move joint 2 */
    mobot.moveJointTo(MOBOT_JOINT3, angle3); /* Move joint 3 */
    mobot.moveJointTo(MOBOT_JOINT2, 0);      /* Move joint 2 */
    mobot.moveJointTo(MOBOT_JOINT3, 0);      /* Move joint 3 back to zero position */
}
```

### 8.1.2 Demo Code for inchworm2.ch Explained

The first portion of the code is identical to the previous demo, and performs the same function of declaring a Mobot variable and connecting to a paired Mobot.

```
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();
```

The next lines of code set the joint speeds for the two body joints, joints 2 and 3, to 50% speed. They are set to fifty percent speed in order to slow the motion down in order to minimize slippage.

```
/* Set mobot motors to speed of 0.50 */
mobot.setJointSpeedRatio(MOBOT_JOINT2, 0.50);
mobot.setJointSpeedRatio(MOBOT_JOINT3, 0.50);
```

Next, we move the mobot into a flat “zero” position, as shown in Figure 15a.

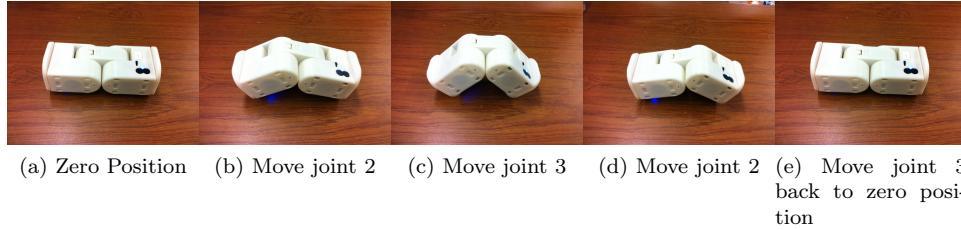


Figure 15: The inchworm motion

```
/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();
```

Finally, we perform the actual inchworm motion. The inchworm motion is a gait defined by a sequence of motions performed by the body joints. The motions are as such:

1. The first body joint, referred to as joint A, rotates towards the ground. This drags the Mobot towards the direction of joint A. (Figure 15b)
2. The other body joint, joint B, rotates towards the ground. Since the center of gravity is currently positioned over joint A, this causes the trailing body joint to slide toward joint A. (Figure 15c)
3. Joint A moves back to a flat position. (Figure 15d)
4. Joint B moves back to a flat position. (Figure 15e)
5. Repeat, if desired.

The direction of travel depends on the selection of the initial body joint. In the following code example, joint 2 is chosen as the initial body joint to move. In this case, the Mobot will traverse towards joint 2. The entire motion is encapsulated in a “for” loop which executes the entire motion four times.

```
/* Do the inchworm gait four times */
int i, num = 4;
double angle2 = -45;
double angle3 = 45;
for(i = 0; i < num; i++) {
    mobot.moveJointTo(MOBOT_JOINT2, angle2); /* Move joint 2 */
    mobot.moveJointTo(MOBOT_JOINT3, angle3); /* Move joint 3 */
    mobot.moveJointTo(MOBOT_JOINT2, 0); /* Move joint 2 */
    mobot.moveJointTo(MOBOT_JOINT3, 0); /* Move joint 3 back to zero position*/
}
```

The values of the variables `angle2` and `angle3` may also be modified to produce different variations of the inchworm gait to accomodate different terrain textures and ground surfaces.

## 8.2 Standing Demo

### 8.2.1 stand2.ch Source Code

```
/* Filename: stand2.ch
 * Make a Mobot stand up on a faceplate */
#include <mobot.h>
```

```

CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set mobot motors to speed of 90 degrees per second */
mobot.setJointSpeed(MOBOT_JOINT2, 90);
mobot.setJointSpeed(MOBOT_JOINT3, 90);
/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

/* Move the mobot into a fetal position */
mobot.moveJointTo(MOBOT_JOINT2, -85);
mobot.moveJointTo(MOBOT_JOINT3, 70);

/* Wait a second for the mobot to settle down */
delay(1);

/* Rotate the bottom faceplate by 45 degrees */
mobot.moveJointTo(MOBOT_JOINT1, 45);

/* Lift the body up */
mobot.moveJointTo(MOBOT_JOINT2, 20);

/* Pan the mobot around for 3 seconds at 45 degrees per second*/
mobot.setJointSpeed(MOBOT_JOINT1, 45);
mobot.moveContinuousTime(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_HOLD, 3);

```

### 8.2.2 stand2.ch Explained

The first portion of the program performs the necessary setup and connecting, similar to the previous demos. Similar to the previous inchworm demo, the motor speeds are set to a speed of 90 degrees per second, and the function `moveToZero()` is called to put the mobot into a flat position. Next, the following lines are executed:

```

mobot.moveJointTo(MOBOT_JOINT2, -85);
mobot.moveJointTo(MOBOT_JOINT3, 70);

```

These movement commands cause the Mobot to curl up into a fetal position with both of its faceplates facing toward the ground. The next line,

```
delay(1);
```

causes the program to pause for one second before continuing. This allows the mobot to settle down, in case it was still in motion from the last movement.

Next, the Mobot rotates one of the faceplates by 45 degrees.

```
mobot.moveJointTo(MOBOT_JOINT1, 45);
```

This endplate will eventually become the “foot” of the standing Mobot. Next, the Mobot lifts itself into a standing position, balancing on its endplate.

```
mobot.moveJointTo(MOBOT_JOINT2, 20);
```

Note that the previous joint angle for Joint 2, a body joint, was -85 degrees. This motion causes joint 2 to rotate all the way to a 20 degree position, which lift up the body of the Mobot such that the Mobot is balancing on faceplate joint 1.

Finally, we rotate joint 1, the foot joint, for three seconds which causes the entire Mobot to rotate in place. The speed is first set to 45 degrees per second to make the rotation a slow rotation. Next, the `moveContinuousTime` member function is used to continuously rotate a joint for a desired amount of time.

```
mobot.setJointSpeed(MOBOT_JOINT1, 45);  
mobot.moveContinuousTime(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_HOLD, 3);
```

## 9 Blocking and Non-Blocking Functions

All of the Mobot movement functions may be designated as either “blocking” functions or “non-blocking” functions. A blocking function is a function which does not return while operations are being performed. All standard C functions, such as `printf()`, are blocking functions. The `moveWait()` function is a blocking function. When called, the function will hang, or “block”, until all the joints have stopped moving. After all joints have stopped moving, the `moveWait()` function will return, and the rest of the program will execute.

Furthermore, some functions have both a blocking version and a non-blocking version. For these functions, the suffix NB denotes that the function is non-blocking. For instance, the function `motionStand()` is blocking, meaning the function will not return until the motion is completed, whereas the function `motionStandNB()` is non-blocking, meaning the function returns immediately and the mobot performs the “standing” motion asynchronously.

The function `moveNB()` is an example of a non-blocking function. When the `moveNB()` function is called, the function immediately returns as the joints begin moving. Any lines of code following the call to `moveNB()` will be executed even if the current motion is still in progress.

Demos for the non-blocking functions are located in the next section of this document.

### 9.1 List of Blocking Movement Functions

- `driveJointToDirect()`
- `driveToDirect()`
- `move()`
- `moveContinuousTime()`
- `moveJoint()`
- `moveJointTo()`
- `moveJointWait()`
- `moveTo()`
- `moveToZero()`
- `moveWait()`
- `motionArch()`
- `motionInchwormLeft()`
- `motionInchwormRight()`
- `motionRollBackward()`

- `motionRollForward()`
- `motionSkinny()`
- `motionStand()`
- `motionTumbleLeft()`
- `motionTumbleRight()`
- `motionTurnLeft()`
- `motionTurnRight()`
- `motionUnstand()`

## 9.2 List of Non-Blocking Movement Functions

- `driveJointToDirectNB()`
- `driveToDirectNB()`
- `moveNB()`
- `moveContinuousNB()`
- `moveJointNB()`
- `moveJointToNB()`
- `moveToNB()`
- `moveToZeroNB()`
- `motionArchNB()`
- `motionInchwormLeftNB()`
- `motionInchwormRightNB()`
- `motionRollBackwardNB()`
- `motionRollForwardNB()`
- `motionSkinnyNB()`
- `motionStandNB()`
- `motionTumbleLeftNB()`
- `motionTumbleRightNB()`
- `motionTurnLeftNB()`
- `motionTurnRightNB()`
- `motionUnstandNB()`

## 9.3 Blocking and Non-Blocking Demo Programs

### 9.3.1 nonblock.ch Source Code

```
/* File: nonblock.ch
   use the non-blocking functoin moveNB() . */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

mobot.moveToZero();

/* Rotate each of the faceplates by 720 degrees */
//mobot.move(720, 0, 0, 720); // Blocking version
mobot.moveNB(720, 0, 0, 720); // Non-Blocking version
while(mobot.isMoving()) {
    printf("mobot is moving ...\\n");
}
printf("move finished!\\n");
```

### 9.3.2 nonblock.ch Source Code Explanation

This demo gives an example of how non blocking functions operate. After the initial setup and initialization similar to the previous demos, the following line is executed:

```
mobot.moveNB(720, 0, 0, 720); // Non-Blocking version
```

The function `moveNB()` is a non-blocking function, which means that the program will continue executing even before the movement has completed. The next lines of code appear as such:

```
while(mobot.isMoving()) {
    printf("mobot is moving ...\\n");
}
```

The previous lines of code basically loops as long as the `mobot.isMoving()` function is returning true. In other words, in plain english, as long as the mobot is moving, the program will print the message “mobot is moving...”. As soon as the mobot completes its motion, the loop will break and the message “move finished!” is printed.

Alternatively, there is a commented line of code that appears in the program, which appears as

```
mobot.move(720, 0, 0, 720); // Non-Blocking version
```

If the `moveNB()` function is replaced with the `move()` function in this program, the message “mobot is moving...” is never printed to the screen. This is due to the fact that `move()` is a blocking function, and by the time the program continues past the `move()` statement, the mobot has already stopped moving.

### 9.3.3 nonblock2.ch Source Code

```
/* File: nonblock2.ch
   use the non-blocking functoin moveNB() . */
#include <mobot.h>
CMobot mobot;
```

```

/* Connect to the paired Mobot */
mobot.connect();

mobot.moveToZero();

/* Rotate each of the faceplates by 360 degrees */
//mobot.moveJoint(MOBOT_JOINT1, 360); // Blocking version
mobot.moveJointNB(MOBOT_JOINT1, 360); // Non-Blocking version
mobot.moveJoint(MOBOT_JOINT4, 360);

```

### 9.3.4 nonblock2.ch Source Code Explanation

The first block of the source code initialized and sets up the remote mobot similar to previous demos. The last two lines in the program appear like so:

```

mobot.moveJointNB(MOBOT_JOINT1, 360); // Non-Blocking version
mobot.moveJoint(MOBOT_JOINT4, 360);

```

The first line turns joint 1 for 360 degrees. However, since it is moved with a non-blocking function, the program immediately continues to the next line. The second line turns joint 4 for 360 degrees. Because computer programs execute so fast compared to the physical motion of the mobots, this program effectively begins rotating joints 1 and 4 simultaneously. Since the function `moveJoint()` is a blocking function, the program will not execute beyond that point until joint 4 has finished moving.

### 9.3.5 nonblock3.ch Source Code

```

/* File: nonblock3.ch
   Roll and arch simultaneously. */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

mobot.moveToZero();

printf("Rolling 360 degrees.\n");
mobot.motionRollForward(360);
printf("Rolling 360 degrees while arching.\n");
mobot.motionArchNB(15);
mobot.motionRollForwardNB(360);
mobot.motionWait();

```

### 9.3.6 nonblock3.ch Source Code Explanation

The first section of code initializes the necessary variables to control remote mobots as seen in previous demos. Next, we print a message on the screen and roll the mobot forward with the following two lines of code.

```

printf("Rolling 360 degrees.\n");
mobot.motionRollForward(360);

```

Next, we make two calls to non-blocking functions.

```

mobot.motionArchNB(15);
mobot.motionRollForwardNB(360);

```

The first call is to the function `motionArchNB()`, which arches the mobot up by moving joints 2 and 3. Since it is a non-blocking function, the program immediately continues on even before the arching motion is finished. The next call rolls the mobot forward by rotating joints 1 and 4. In effect, these two lines cause the mobot to roll forward and arch simultaneously. It is important to note that this compound motion works because the Arch motion only moves joints 2 and 3 while the rolling motions only move joints 1 and 4, so there are no conflicting motor commands.

In order to wait for all motions to finish, the last line of the program is

```
mobot.motionWait();
```

The `motionWait()` function will wait until all mobot motions are finished.

Note that if a program contains non-blocking functions, it is typically necessary to call a waiting function such as `moveWait()` or `motionWait()` before the program terminates. If a waiting function is not called, the program may terminate before the motion has been completed, which may halt the mobot in the middle of one of its motions.

## 9.4 Preprogrammed Motion Demos with Non-Blocking Functions

### 9.4.1 unstand2.ch Source Code

```

/* Filename: unstand.ch
 * Drop the mobot down while it is standing on faceplate 1. */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();
mobot.moveToZero();

mobot.moveJointToNB(MOBOT_JOINT2, -85);
mobot.moveJointToNB(MOBOT_JOINT3, 45);
mobot.moveWait();
mobot.moveToZero();

```

### 9.4.2 unstand2.ch Source Code Explanation

The first block of code,

```

/* Filename: unstand.ch
 * Drop the mobot down from a standing position. */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

```

initialize the program and connect to the remote mobot. Next, a series of non-blocking movements are performed:

```

mobot.moveJointToNB(MOBOT_JOINT2, -85);
mobot.moveJointToNB(MOBOT_JOINT3, 45);

```

Because both of these function calls are non-blocking, this function will effectively move joints 2 and 3 simultaneously. Since these are both non-blocking functions, a call to `moveWait()` is necessary to wait for the mobot to complete its motions, as done in the next line:

```
mobot.moveWait();
```

Finally, we move the mobot into a flat zero position with the following line:

```
mobot.moveToZero();
```

#### 9.4.3 tumble2.ch Source Code

```
/* Filename: tumble2.ch
 * Tumbling mobot */
#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

/* Begin tumbling for "num" times */
int i, num = 2;
for(i = 0; i < num; i++) {
    /* First lift and tumble */
    mobot.moveJointTo(MOBOT_JOINT2, -85);
    mobot.moveJointTo(MOBOT_JOINT3, 80);
    mobot.moveJointTo(MOBOT_JOINT2, 0);
    mobot.moveJointTo(MOBOT_JOINT3, 0);
    mobot.moveJointTo(MOBOT_JOINT2, 80);
    mobot.moveJointTo(MOBOT_JOINT2, 45);
    /* Second lift and tumble */
    mobot.moveJointTo(MOBOT_JOINT3, -85);
    mobot.moveJointTo(MOBOT_JOINT2, 80);
    mobot.moveJointTo(MOBOT_JOINT3, 0);
    mobot.moveJointTo(MOBOT_JOINT2, 0);
    mobot.moveJointTo(MOBOT_JOINT3, 80);
    if(i != (num-1)) { /* Do not perform this motion on the last tumble */
        mobot.moveJointTo(MOBOT_JOINT3, 45);
    }
}

/* Unstand the mobot */
mobot.moveJointToNB(MOBOT_JOINT2, 0);
mobot.moveJointToNB(MOBOT_JOINT3, 0);
mobot.moveWait();
mobot.moveToZero();
```

#### 9.4.4 tumble2.ch Source Code Explanation

The first lines of the program,

```

#include <mobot.h>
CMobot mobot;

/* Connect to the paired Mobot */
mobot.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot.moveToZero();

```

initialize the proper variables and connections as seen in previous demos.

Next, we begin the tumbling motion. We consider each tumbling motion to be the mobot flipping over twice. The reason the mobot flips twice per tumble is so that when the tumbling motion is done, the mobot ends right side up. A for loop is used to to tumble “num” times, as shown in the following code:

```

int i, num = 2;
for(i = 0; i < num; i++) {

```

These lines create two new variables, “i” and “num”, which are the loop counter, and the number of times to tumble, respectively.

Inside the loop, two tumbling motions are performed. The first tumbling motion appears as such:

```

/* First lift and tumble */
mobot.moveJointTo(MOBOT_JOINT2, -85);
mobot.moveJointTo(MOBOT_JOINT3, 80);
mobot.moveJointTo(MOBOT_JOINT2, 0);
mobot.moveJointTo(MOBOT_JOINT3, 0);
mobot.moveJointTo(MOBOT_JOINT2, 80);
mobot.moveJointTo(MOBOT_JOINT2, 45);

```

This movement is similar to the `motionStand()` motion, except that the mobot flips all the way over after standing up. After this motion is done, the mobot is balancing on joint 4. Next, we flip again so that the mobot is balancing on joint 1.

```

/* Second lift and tumble */
mobot.moveJointTo(MOBOT_JOINT3, -85);
mobot.moveJointTo(MOBOT_JOINT2, 80);
mobot.moveJointTo(MOBOT_JOINT3, 0);
mobot.moveJointTo(MOBOT_JOINT2, 0);
mobot.moveJointTo(MOBOT_JOINT3, 80);
if(i != (num-1)) { /* Do not perform this motion on the last tumble */
    mobot.moveJointTo(MOBOT_JOINT3, 45);
}
}

```

The `if` statement prevents the last motion from being executed when the mobot is on its last tumble. This is because the last motion in the loop is a preparatory motion for the next tumble, which is not needed if the mobot is currently performing its last tumble.

The entire motion, consisting of two flips, are performed “num” times. After the loop is completed, the mobot is made to fall back down into a prone positions with the following lines of code:

```

/* Unstand the mobot */
mobot.moveJointToNB(MOBOT_JOINT2, 0);
mobot.moveJointToNB(MOBOT_JOINT3, 0);
mobot.moveWait();
mobot.moveToZero();

```

## 10 Controlling Multiple Modules

The Mobot control software is designed to be able to control multiple modules simultaneously. There are some important differences in the program which enable the control of multiple modules. A small demo program which controls two modules simultaneously will first be presented, followed by a detailed explanation of the program elements.

### 10.1 twoModules.ch Source Code

```
/* Filename: twoModules.ch
 * Control two modules and make them stand and inchworm simultaneously. */
#include <mobot.h>
CMobot mobot1;
CMobot mobot2;

/* Connect mobot variables to the mobot modules. */
mobot1.connect();
mobot2.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot1.moveToZeroNB();
mobot2.moveToZeroNB();

mobot1.moveWait();
mobot2.moveWait();

/* Instruct the first mobot to stand and the second mobot to inchworm left four
 * times simultaneously. As soon as the first mobot stands up, rotate its joints
 * 1 and 4 360 degrees. */
mobot1.motionStandNB();
mobot2.motionInchwormLeftNB(4);
mobot1.motionWait();
mobot1.moveNB(360, 0, 0, 360);
mobot1.moveWait();
mobot2.motionWait();
/* Instruct the first mobot unstand and the second mobot inchworm right four
 * times simultaneously. */
mobot1.motionUnstandNB();
mobot2.motionInchwormRightNB(4);
mobot1.motionWait();
mobot1.motionTumbleLeft(1);
mobot2.motionWait();
```

### 10.2 Demo Explanation

The first two lines of interest appear as such:

```
CMobot mobot1;
CMobot mobot2;
```

These two lines declare two separate variables which will represent the two separate Mobot modules. Next, we need to connect each variable to a physically separate Mobot. This is done with the following lines.

```
mobot1.connect();  
mobot2.connect();
```

These two lines connect the mobots to the first two addresses of the known mobot addresses. The list of the computer's known mobot addresses may be configured in the process detailed in Section 2 on page 11. For each separate control program, the first call to the `connect()` member function will connect to the first mobot listed in the configuration file. Each successive call to the `connect()` function will connect to successive mobots listed in the configuration file. The order in which they are connected may be modified using the "Configure Mobot Bluetooth" dialog, as discussed in Section 2.

```
mobot1.moveToZeroNB();  
mobot2.moveToZeroNB();
```

These two lines command the two mobots to move to their zero positions. Note that these functions are non-blocking. This means that the `moveToZeroNB()` function will return immediately, and will not wait for the first mobot to finish completing the motion before commanding the second mobot to begin. In a normal program, this effectively causes both mobots to move to their zero positions simultaneously.

```
mobot1.moveWait();  
mobot2.moveWait();
```

Since the `moveToZeroNB()` functions are non-blocking, we would like the program to wait until the motions are complete before continuing. By calling `moveWait()` on both of the mobots, we can be assured that the mobots have finished moving before the program continues.

```
mobot1.motionStandNB();  
mobot2.motionInchwormLeftNB(4);
```

Similar to the calls to `moveToZeroNB()`, this block of code instructs the first Mobot to stand and the second Mobot to perform the inchworm motion four times. Note that we call the non-blocking versions of the functions, `motionStandNB()` and `motionInchwormLeftNB()`. Since these functions are non-blocking, both mobots will effectively perform the motions simultaneously.

When the first mobot finishes standing, we want it to spin around on its faceplate while the second mobot is still inchworming. We can accomplish this by first waiting for the standing motion to finish, and then moving the faceplate joints of the first mobot, as in the following lines of code.

```
mobot1.motionWait();  
mobot1.moveNB(360, 0, 0, 360);
```

Before we continue with the program, we wish to ensure that motion has stopped on both mobots. Since the last command sent to `mobot1` was a `moveNB()` command, we use the `moveWait()` function to wait for that movement to finish. Similarly, the last command sent to `mobot2` was a motion command, and so we use `motionWait()` to wait for the motion to finish. The two lines of code are seen in the program as follows:

```
mobot1.moveWait();  
mobot2.motionWait();
```

Finally, the following lines,

```
mobot1.motionUnstandNB();  
mobot2.motionInchwormRightNB(4);  
mobot1.motionWait();  
mobot1.motionTumbleLeft(1);  
mobot2.motionWait();
```

make the first mobot come back down from its standing position and the second mobot inchworm to the right four times simultaneously. Note that the function `motionWait()` is used to wait for motions, which are compound movements, to finish. This is similar to how the function `moveWait()` is used to wait for individual movements to finish. After `mobot1` finishes unstanding, it will begin a tumbling motion. Since the tumbling motion function is called before the call to `mobot2.motionWait()`, `mobot1` will begin tumbling even if `mobot2` has finished its previous motion, which was inchworming 4 times.

## 10.3 Controlling Multiple Connected Modules

### 10.3.1 lift.ch, Lifting Demo

```
/* Filename: lift.ch
Lift two connected mobots.
Joint 4 of the 1st mobot should be connected to
Joint 1 of the 2nd mobot as
      1st           2nd
|-----|-----|     |-----|-----|
1|   2     |   3     | 4 X 1|   2     |   3     | 4
|-----|-----|     |-----|-----|
*/
#include <mobot.h>
CMobot mobot1;
CMobot mobot2;

/* Connect mobot variables to the mobot modules. */
mobot1.connect();
mobot2.connect();

/* Set the mobot to "home" position, where all joint angles are 0 degrees. */
mobot1.moveToZeroNB();
mobot2.moveToZeroNB();

mobot1.moveWait();
mobot2.moveWait();

/* First lift */
mobot1.moveToNB(0, -90, 0, 0);
mobot2.moveToNB(0, 0, 90, 0);
mobot1.moveWait();
mobot2.moveWait();
delay(1);

/* Second lift */
mobot1.moveToNB(0, 0, 90, 0);
mobot2.moveToNB(0, -90, 0, 0);
mobot1.moveWait();
mobot2.moveWait();
delay(1);

/* Move to zero position */
mobot1.moveToZeroNB();
mobot2.moveToZeroNB();
```



(a) First lift

(b) Second lift

Figure 16: Lifting motion with two connected modules

```
mobot1.moveWait();
mobot2.moveWait();
```

### 10.3.2 lift.ch Source Code Explanation

This demo is designed to lift two connected modules up into a two-legged standing configuration. The mobots are connected such that the fourth joint of mobot one is connected to the first joint of mobot two. The two connect mobots act as one long mobot.

The first portion of the program initialize two variables called `mobot1` and `mobot2`, which will be used to control the two connected modules. Once the mobots are initialized, they are both moved into a flat zero position.

The next lines,

```
/* First lift */
mobot1.moveNB(0, -90, 0, 0);
mobot2.moveNB(0, 0, 90, 0);
mobot1.moveWait();
mobot2.moveWait();
```

rotate the joints on either end of the compound mobot to perform the first portion of the lift, as shown in Figure 16a. The motion is performed as two non-blocking calls to `moveNB()` and then waiting for both movements to finish.

The next lines rotate the inner joints of the compound mobot to lift the mobot one step higher, as shown in Figure 16b.

```
/* Second lift */
mobot1.moveToNB(0, 0, 90, 0);
mobot2.moveToNB(0, -90, 0, 0);
mobot1.moveWait();
mobot2.moveWait();
```

Again, these lines perform two calls to the non-blocking function `moveNB()` and then wait for the movements to finish.

Finally, we move both mobots back to their zero positions, which drops the compound mobot back onto the ground.

```
mobot1.moveToZeroNB();
mobot2.moveToZeroNB();
mobot1.moveWait();
mobot2.moveWait();
```

## 11 Commanding Multiple Mobots to Perform Identical Tasks

The class called `CMobotGroup` can be used to control multiple modules simultaneously. The `CMobotGroup` represents a group of mobots. Any command that is given to the group of modules is duplicated to each member of the group.

The majority of the movement functions available in the `CMobot` class are also available in the `CMobotGroup` class. The detailed information for each member function are presented in Appendix B. Following is a complete listing of the available member functions in the `CMobotGroup` class.

Function	Description
<code>CMobotGroup()</code>	The <code>CMobotGroup</code> constructor function. This function is called automatically and should not be called explicitly.
<code>~CMobotGroup()</code>	The <code>CMobotGroup</code> destructor function. This function is called automatically and should not be called explicitly.
<code>addRobot()</code>	Add a mobot to be a member of the mobot group.
<code>move()</code>	Move all four joints of the mobots by specified angles.
<code>moveNB()</code>	Identical to <code>move()</code> but non-blocking.
<code>moveContinuousNB()</code>	Move joints continuously. Joints will move until stopped.
<code>moveContinuousTime()</code>	Move joints continuously for a certain amount of time.
<code>moveJointContinuousNB()</code>	Move a single joint on all mobots continuously.
<code>moveJointContinuousTime()</code>	Move a single joint on all mobots continuously for a specific amount of time.
<code>moveJoint()</code>	Move a motor from its current position by an angle.
<code>moveJointNB()</code>	Identical to <code>moveJoint()</code> but non-blocking.
<code>moveJointTo()</code>	Set the desired motor position for a joint.
<code>moveJointToNB()</code>	Identical to <code>moveJointTo()</code> but non-blocking.
<code>moveJointWait()</code>	Wait until the specified motor has stopped moving.
<code>moveTo()</code>	Move all four joints of the mobots to specified absolute angles.
<code>moveToNB()</code>	Identical to <code>moveTo()</code> but non-blocking.
<code>moveToZero()</code>	Instructs all motors to go to their zero positions.
<code>moveToZeroNB()</code>	Identical to <code>moveToZero()</code> but non-blocking.
<code>moveWait()</code>	Wait until all motors have stopped moving.
<code>setJointSpeed()</code>	Set a motor's speed setting in radians per second.
<code>setJointSpeeds()</code>	Set all motor speeds in radians per second.
<code>setJointSpeedRatio()</code>	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
<code>setJointSpeedRatios()</code>	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
<code>setTwoWheelRobotSpeed()</code>	Move the mobots at a constant forward velocity.
<code>stop()</code>	Stop all currently executing motions of the mobot.

Compound Motions	These are convenience functions of commonly used compound motions.
<code>motionArch()</code>	Move each mobot in the group into an arched configuration.
<code>motionArchNB()</code>	Identical to <code>motionArch()</code> but non-blocking.
<code>motionInchwormLeft()</code>	Inchworm motion towards the left.
<code>motionInchwormLeftNB()</code>	Identical to <code>motionInchwormLeft()</code> but non-blocking.
<code>motionInchwormRight()</code>	Inchworm motion towards the right.
<code>motionInchwormRightNB()</code>	Identical to <code>motionInchwormRight()</code> but non-blocking.
<code>motionRollBackward()</code>	Roll on the faceplates toward the backward direction.
<code>motionRollBackwardNB()</code>	Identical to <code>motionRollBackward()</code> but non-blocking.
<code>motionRollForward()</code>	Roll on the faceplates forwards.
<code>motionRollForwardNB()</code>	Identical to <code>motionRollForward()</code> but non-blocking.
<code>motionSkinny()</code>	Move the mobots into a skinny configuration.
<code>motionSkinnyNB()</code>	Identical to <code>motionSkinnyNB()</code> but non-blocking.
<code>motionStand()</code>	Stand the mobots up on its end.
<code>motionStandNB()</code>	Identical to <code>motionStandNB()</code> but non-blocking.
<code>motionTumbleLeft()</code>	Tumble the mobots end over end.
<code>motionTumbleLeftNB()</code>	Identical to <code>motionTumbleLeftNB()</code> but non-blocking.
<code>motionTumbleRight()</code>	Tumble the mobots end over end.
<code>motionTumbleRightNB()</code>	Identical to <code>motionTumbleRightNB()</code> but non-blocking.
<code>motionTurnLeft()</code>	Rotate the mobots counterclockwise.
<code>motionTurnLeftNB()</code>	Identical to <code>motionTurnLeft()</code> but non-blocking.
<code>motionTurnRight()</code>	Rotate the mobots clockwise.
<code>motionTurnRightNB()</code>	Identical to <code>motionTurnRight()</code> but non-blocking.
<code>motionUnstand()</code>	Move each mobot in the group currently standing on its end down into zero position.
<code>motionUnstandNB()</code>	Identical to <code>motionUnstand()</code> but non-blocking.
<code>motionWait()</code>	Wait for preprogrammed mobotic motions to complete.

## 11.1 Demo program group.ch

### 11.1.1 Source Code

```
/* Filename: group.ch
 * Control multiple Mobot modules simultaneously using the CMobotGroup class */
#include <mobot.h>
CMobot mobot1;
CMobot mobot2;
CMobotGroup group;

/* Connect to the mobots listed in the configuration file. */
mobot1.connect();
mobot2.connect();

/* Add the two modules to be members of our group */
group.addRobot(mobot1);
group.addRobot(mobot2);

/* Now, any commands given to "group" will cause both mobot1 and mobot2 to
 * execute the command. */
group.motionInchwormLeft(4); /* Both mobots inchworm left 4 times */
group.motionStand(); /* Both mobots stand */
```

```

group.move(360, 0, 0, 360); /* Joints 1 and 4 rotate 360 degrees */
delay(3);                  /* Mobots stand still for 3 seconds */
group.motionUnstand();     /* Mobots get back down from standing */

```

### 11.1.2 Demo Explanation

The first lines of interest appear as such:

```

CMobot mobot1;
CMobot mobot2;
CMobotGroup group;

```

These lines declare two mobot variables, and one variable which will represent a group of mobots. Next, we connect the mobot variables to their physical counterparts.

```

mobot1.connect();
mobot2.connect();

```

Once they are connected, we wish to add both of these mobots to our mobot group, which we have named `group`.

```

group.addRobot(mobot1);
group.addRobot(mobot2);

```

Finally, we wish for all of the mobots in our mobot group, namely `mobot1` and `mobot2`, to perform an inchworm motion four times, followed by a standing motion. This is done with the following lines:

```

group.motionInchwormLeft(4); /* Both mobots inchworm left 4 times */
group.motionStand();        /* Both mobots stand */

```

After the mobots stand up, the line

```
group.move(360, 0, 0, 360); /* Joints 1 and 4 rotate 360 degrees */
```

makes the mobots perform a 360 degree rotation on their faceplates while standing.

The next line,

```
delay(3); /* Mobots stand still for 3 seconds */
```

makes the mobots stand still for three seconds. After standing still for three seconds, the line

```
group.motionUnstand(); /* Mobots get back down from standing */
```

makes both mobots move back down from a standing position into a prone position.

## 11.2 Demo Program groups.ch

### 11.2.1 groups.ch Source Code

```

/* Filename: groups.ch
 * Control multiple Mobot groups simultaneously using the CMobotGroup class */
#include <mobot.h>
CMobot mobot1;
CMobot mobot2;
CMobot mobot3;
CMobot mobot4;

```

```

CMobotGroup groupA;
CMobotGroup groupB;
CMobotGroup groupC;
CMobotGroup groupD;

/* Connect to the mobots listed in the configuration file. */
mobot1.connect();
mobot2.connect();
mobot3.connect();
mobot4.connect();

/* Add the mobots to our groups. The groups should be 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(mobot1);
groupA.addRobot(mobot2);
groupA.addRobot(mobot3);
groupA.addRobot(mobot4);

/* Group B */
groupB.addRobot(mobot1);
groupB.addRobot(mobot2);

/* Group C */
groupC.addRobot(mobot3);
groupC.addRobot(mobot4);

/* Group D */
groupD.addRobot(mobot1);
groupD.addRobot(mobot2);
groupD.addRobot(mobot3);

/* Make group B roll forward and group C roll backward at the same time */
groupB.motionRollForwardNB(360);
groupC.motionRollBackwardNB(360);
groupB.motionWait();
groupC.motionWait();

/* Make all the mobot stand up */
groupA.motionStand();

/* Make mobots 1 and 2 (Group B) rotate counter-clockwise and mobots 3 and 4
 * (Group C) rotate clockwise. */
groupB.moveNB(360, 0, 0, 360);
groupC.moveNB(-360, 0, 0, -360);
groupB.moveWait();
groupC.moveWait();

```

```

/* Make mobot 4 unstand and inchworm while the remaining mobots spin. */
groupD.moveContinuousNB(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_FORWARD);
mobot4.motionUnstand();
mobot4.motionInchwormLeft(2);
groupD.motionUnstand();

```

### 11.2.2 Demo Explanation

The program begins by including the `mobot.h` header file and declaring a number of mobot variables and mobot group variables.

```

#include <mobot.h>
CMobot mobot1;
CMobot mobot2;
CMobot mobot3;
CMobot mobot4;
CMobotGroup groupA;
CMobotGroup groupB;
CMobotGroup groupC;
CMobotGroup groupD;

```

Next, we connect our four robot variables to the first four robots configured in RoboMancer.

```

mobot1.connect();
mobot2.connect();
mobot3.connect();
mobot4.connect();

```

Now we begin adding our mobots to their groups. We will have four different groups, and each mobot will belong to more than one group. They will be organized as such:

- Group A: mobot1, mobot2, mobot3, mobot4
- Group B: mobot1, mobot2
- Group C: mobot3, mobot4
- Group D: mobot1, mobot2, mobot3

The following code divides the mobots up into our groups.

```

/* Group A */
groupA.addRobot(mobot1);
groupA.addRobot(mobot2);
groupA.addRobot(mobot3);
groupA.addRobot(mobot4);

/* Group B */
groupB.addRobot(mobot1);
groupB.addRobot(mobot2);

/* Group C */
groupC.addRobot(mobot3);
groupC.addRobot(mobot4);

```

```

/* Group D */
groupD.addRobot(mobot1);
groupD.addRobot(mobot2);
groupD.addRobot(mobot3);

```

Now we perform some group oriented motions. First, we will have group B roll forward and group C roll backward simultaneously. This is done with calls to non-blocking functions and using the function `motionWait()` to wait for the non-blocking functions to finish.

```

/* Make group B roll forward and group C roll backward at the same time */
groupB.motionRollForwardNB(360);
groupC.motionRollBackwardNB(360);
groupB.motionWait();
groupC.motionWait();

```

Next, we make all the mobots, which are members of the group `groupA`, stand up.

```

/* Make all the mobot stand up */
groupA.motionStand();

```

While the mobots are standing, we want group B to spin counter-clockwise, and group C to spin clockwise. To do this, we use the non-blocking function `moveNB()` and then wait for the movements to finish with `moveWait()`.

```

/* Make mobots 1 and 2 (Group B) rotate counter-clockwise and mobots 3 and 4
 * (Group C) rotate clockwise. */
groupB.moveNB(360, 0, 0, 360);
groupC.moveNB(-360, 0, 0, -360);
groupB.moveWait();
groupC.moveWait();

```

Finally, we want mobots 1, 2, and 3 to spin while mobot 4 unstands and inchworms twice. As soon as mobot 4 is done inchworming, we want mobots 1, 2, and 3 to unstand as well. We do this by using a non-blocking continuous move function, `moveContinuousNB()`. While the group is moving, we unstand mobot 4 and inchworm it twice. After mobot 4 is done inchworming, we unstand Group D.

```

/* Make mobot 4 unstand and inchworm while the remaining mobots spin. */
groupD.moveContinuousNB(MOBOT_FORWARD, MOBOT_HOLD, MOBOT_HOLD, MOBOT_FORWARD);
mobot4.motionUnstand();
mobot4.motionInchwormLeft(2);
groupD.motionUnstand();

```

## 12 Data Acquisition, Data Processing, and Application Examples for Learning Algebra

### 12.1 Example 1

#### 12.1.1 Problem Statement

Roll a mobot forward by rotating the wheels 720 degrees at a constant speed of 45 degrees per second. Record the motion of joint 1 during the motion and display a plot of the joint angle versus time. The motion should show that the joint angle  $\theta$  is a linear function of time in the form  $\theta = 45t$ .

### 12.1.2 dataAcquisition.ch Source Code

```
/* Filename: dataAcquisition.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

double speed = 45; /* Degrees/second */
double angle = 720; /* Degrees */
double timeInterval = 0.1; /* Seconds */

/* Figure out how many data points we will need. First, figure out the
 * approximate amount of time the movement should take. */
double movementTime = angle / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data for time and angle */
array double time[numDataPoints];
array double angles1[numDataPoints];

/* Declare plotting variables */
CPlot plot1, plot2, plot3;

/* Declare time shifted data */
double tolerance = 1.0; /* Degrees */

/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
mobot.setJointSpeed(MOBOT_JOINT1, speed);
mobot.setJointSpeed(MOBOT_JOINT4, speed);

/* Start capturing data */
mobot.recordAngle(MOBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

/* Move the joint 720 degrees */
mobot.move(angle, 0, 0, angle);

/* Wait for recording to finish */
mobot.recordWait();

/* Plot the data */
plot1.title("Original Data for Joint Angle 1 versus Time");
```

```

plot1.label(PLOT_AXIS_X, "Time (seconds)");
plot1.label(PLOT_AXIS_Y, "Angle (degrees)");
plot1.data2DCurve(time, angles1, numDataPoints);
plot1.grid(PLOT_ON);
plot1.plotting();

/* Plot the data */
plot2.title("Data for Joint Angle 1 versus Time");
plot2.label(PLOT_AXIS_X, "Time (seconds)");
plot2.label(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2DCurve(time, angles1, numDataPoints);
plot2.grid(PLOT_ON);
plot2.plotting();

/* Adjust the time delay */
/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time, angles1);

/* Plot the data */
plot3.title("Shifted Data for Joint Angle 1 versus Time");
plot3.label(PLOT_AXIS_X, "Time (seconds)");
plot3.label(PLOT_AXIS_Y, "Angle (degrees)");
plot3.data2DCurve(time, angles1, numDataPoints);
plot3.grid(PLOT_ON);
plot3.plotting();

```

### 12.1.3 dataAcquisition.ch Explained

The first block of code,

```

#include <mobot.h>
#include <chplot.h>
#include <numeric.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

```

includes header files and declares our mobot variable. The header file `chplot.h` is necessary for plotting figures, which will be done later in the program. The header file `numeric.h` is necessary for creating and manipulating computational arrays, which are also used in this program. Both plotting and computational arrays are covered in *Ch User's Guide*.

Next some variables are declared. First, we declare a variable to hold our desired speed:

```
double speed = 45; /* Degrees/second */
```

Now we declare a variable to hold the angle we want to rotate our faceplate joints:

```
double angle = 720; /* Degrees */
```

And another variable to hold our polling interval:

```
double timeInterval = 0.1; /* Seconds */
```

The variable `timeInterval` holds the time in seconds between data acquisition events. The value 0.1 indicates that new data should be acquired from the mobot every 0.1 seconds, or ten times a second, in other words. Lower values will result in faster rates of data acquisition and smoother plots.

Next, we need to determine how long to record the data. This requires us to estimate the amount of time our motion will take. We will perform a motion in which the wheels of the mobot turn 720 degrees at 45 degrees a second. To find the amount of time the movement will take, we can use the formula

$$t = \frac{\theta}{\omega}$$

where  $\theta$  is the angle to turn the joint and  $\omega$  is the speed at which the joint is turning. The line

```
double movementTime = angle / speed; /* Seconds */
```

performs this calculation and stores the result in a variable called `movementTime`.

Because physical systems are never mathematically precise, we must account for some potential fluctuations during the motion of the mobot. Because we want to be sure to record the motion in its entirety, it is beneficial to record for a timespan that is actually longer than the estimated movement time. The next line:

```
movementTime = movementTime + 1;
```

adds another second to the estimated movement time.

Finally, we calculate the number of data points which will be captured during our recording session. This information is necessary so that we know how big to make the computational arrays which will be used to store the data. The number of data points can be determined by the formula

$$N = \frac{t}{\Delta t}$$

where  $t$  is the total movement time and  $\Delta t$  is the time interval between readings. We declare a new variable based on the previous equation called `numDataPoints` with the following line of code:

```
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

Next, we declare some computational arrays to hold the data we will record.

```
array double time[numDataPoints];
array double angles1[numDataPoints];
```

Note that we have used the variable `numDataPoints` to specify the size of our arrays. The `recordAngle()` function, used later to record the data, requires two arrays to store data. The first array will store a series of timestamps for each data point. Timestamps are stored for greater accuracy of the data. Although the user is able to request a certain time interval, wireless and communication uncertainties may contribute to some fluctuation in communication times and speeds. In order to ensure accurate data, as the Mobot sends joint data to the computer, the Mobot will place an accurate timestamp on each piece of data as it was recorded on the Mobot itself. These timestamps are stored in the `time` array and the joint angles will be stored in the `angles1` array.

The last variables we will declare are related to generating a plot to display the captured data.

```
CPlot plot1, plot2, plot3;
array double angles1Unwrapped[numDataPoints];
```

The variables `plot1` and `plot2` will hold the two plots we will generate, and the computational array `angles1Unwrapped` will hold the “unwrapped” data recorded from the mobot. More discussion regarding unwrapping data will be presented towards the end of this demo.

Finally, the last variable declared is `tolerance`. This variable holds a value in degrees. This tolerance declared here is the tolerance used later to detect the time where motion first begins. Because the data

recording process typically starts a fraction of a second before the motion begins, the plotted motions may not begin at time zero. When detecting the time of the first motion, if any joint moves by an amount more than the tolerance value, that time is taken to be the beginning of the motion.

Before we begin the mobot's motion, we first move it to zero position and set the joints to the correct speeds with the following lines of code:

```
/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
mobot.setJointSpeed(MOBOT_JOINT1, speed);
mobot.setJointSpeed(MOBOT_JOINT4, speed);
```

Next, we start capturing the data by using the `recordAngle()` function:

```
mobot.recordAngle(MOBOT_JOINT1, time, angles1, numDataPoints, timeInterval);
```

The recording process begins immediately and continues in the background. Now, we may perform the motion we wish to record, which is to simply rotate each of the faceplates by the amount desired:

```
mobot.move(angle, 0, 0, angle);
```

After we have performed all of our motions, we need to ensure that the recording process which had been running in the background is completed. We used the function `recordWait()` to make the program wait for any currently recording tasks to finish:

```
mobot.recordWait();
```

After data has been recorded, we use our previously declared plotting variables to generate plots of the data. We will generate two separate plots. First, we simply plot the original raw data directly from the Mobot.

```
plot1.title("Original Data for Joint Angle 1 versus Time");
plot1.labelX(PLOT_AXIS_X, "Time (seconds)");
plot1.labelY(PLOT_AXIS_Y, "Angle (degrees)");
plot1.data2DPlot(time, angles1, numDataPoints);
plot1.grid(PLOT_ON);
plot1.plotting();
```

The previous lines of code set the plot title, set the X-axis label, set the Y-axis label, insert the plot data, turn the grid on, and plot the data, respectively. The plot that is generated is shown in Figure 17.

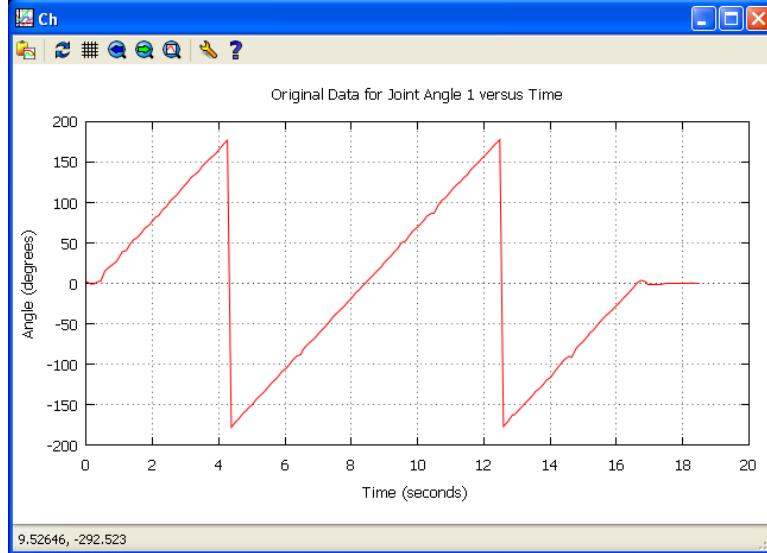


Figure 17: The original data recorded by `dataAcquisition.ch`.

Figure 17 appears to show a zig-zag pattern for the angle of the joint. This is because the raw angle data obtained from a Mobot is only in the range -180 degrees to 180 degrees. When a joint rotates past 180 degrees, the angle reading switches over -180 degrees. In order to get rid of the zig-zagging of the data, the data must be “unwrapped”. This is done with the `unwrapdeg()` function, as done in the following line of code:

```
unwrapdeg(angles1_unwrapped, angles1);
```

The function places the unwrapped version of the `angles1` array into our array `angles1_unwrapped`. Next, we plot the unwrapped data in a similar method to plotting our wrapped data:

```
plot2.title("Unwrapped Data for Joint Angle 1 versus Time");
plot2.labelX(PLOT_AXIS_X, "Time (seconds)");
plot2.labelY(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2DCurve(time, angles1_unwrapped, numDataPoints);
plot2.grid(PLOT_ON);
plot2.plotting();
```

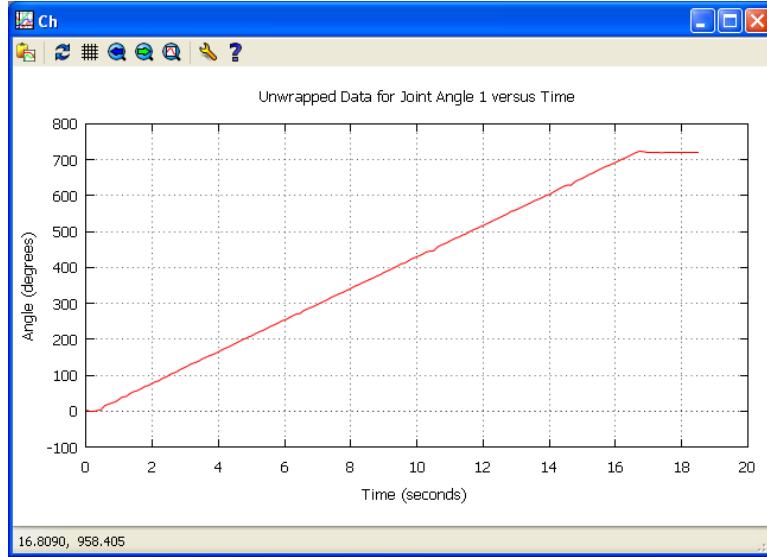


Figure 18: The unwrapped data recorded by `dataAcquisition.ch`.

Figure 18 displays the unwrapped data. It shows the motion as one fluid motion starting at 0 degrees and ending at 720 degrees.

Finally, we want to generate a plot where the beginning of the motion correctly starts at time 0. Looking at Figure 18, you will notice that there is about a fraction of a second delay before the joints actually start moving. For our final plot, we want to display a plot of the same data, but shifted to the left so that the motion starts at time zero. This will make it easier to verify the speed of the joint and the time it takes to get to 720 degrees.

To do this, we use the `shiftTime()` function. The `shiftTime()` function is used to shift graphs of data to the left by detecting the first moment a joint begins moving. The following code shifts the data to the left so that it appears that the motion begins right at time 0.

```
shiftTime(tolerance, numDataPoints, time, angles1Unwrapped);
```

Finally, we plot the shifted data.

```
plot3.title("Unwrapped and shifted Data for Joint Angle 1 versus Time");
plot3.labelX(PLOT_AXIS_X, "Time (seconds)");
plot3.labelY(PLOT_AXIS_Y, "Angle (degrees)");
plot3.data2DCurve(time, angles1_unwrapped, numDataPoints);
plot3.grid(PLOT_ON);
plot3.plotting();
```

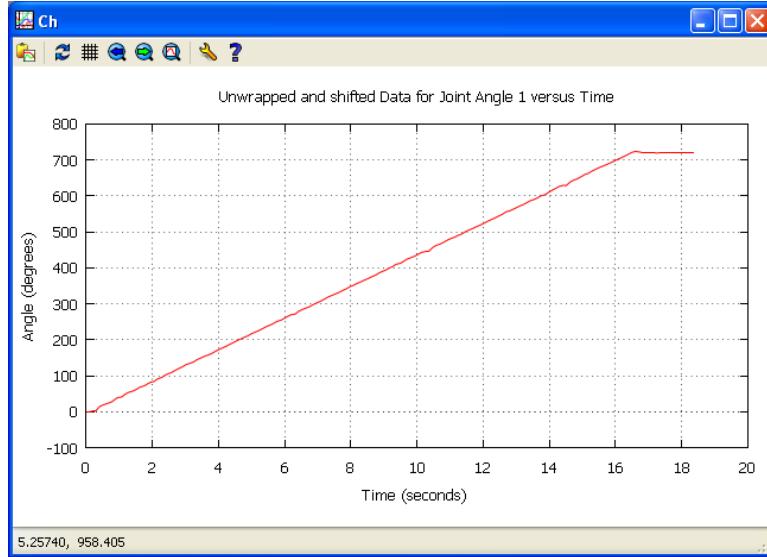


Figure 19: The unwrapped and shifted data recorded by `dataAcquisition.ch`.

Figure 19 displays our unwrapped data that has been time shifted. It can be seen that the beginning of the motion now happens at time 0, and it may be verified that the line generated is very close to the expected function,  $\theta = 45t$ .

In many applications, the user may only be interested in the unwrapped, time shifted data. In such a case, a simplified version of the program may be used. The program `dataAcquisition1.ch` below removes the intermediate plots and can be modified to solve other data acquisition problems.

#### 12.1.4 `dataAcquisition1.ch` Source Code

```
/* Filename: dataAcquisition1.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

double speed = 45; /* Degrees/second */
double angle = 720; /* Degrees */
double timeInterval = 0.1; /* Seconds */

/* Figure out how many data points we will need. First, figure out the
 * approximate amount of time the movement should take. */
double movementTime = angle / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

```

/* Initialize the arrays to be used to store data for time and angle */
array double time[numDataPoints];
array double angles1[numDataPoints];

/* Declare plotting variables */
CPlot plot;
array double angles1Unwrapped[numDataPoints];

/* Declare time shifted data */
double tolerance = 1.0; /* Degrees */

/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
mobot.setJointSpeed(MOBOT_JOINT1, speed);
mobot.setJointSpeed(MOBOT_JOINT4, speed);

/* Start capturing data */
mobot.recordAngle(MOBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

/* Move the joint 720 degrees */
mobot.move(angle, 0, 0, angle);

/* Wait for recording to finish */
mobot.recordWait();

/* Adjust the time delay */
/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time, angles1);

/* Plot the data */
plot.title("Unwrapped and shifted Data for Joint Angle 1 versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2DCurve(time, angles1, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();

```

## 12.2 Example 2

It is also possible to record the joint angles for all four joints simultaneously with the `recordAngles()` function. The function is conceptually similar to `recordAngle()`, except that it obtains 4 joint angles for each timestamp.

### 12.2.1 Problem Statement

Record and plot the motion of all 4 joints of the Mobot as it performs the following two motions:

1. Turn right by rotating joints 360 degrees.
2. Inchworm to the left twice.

### 12.2.2 dataAcquisition2.ch Source Code

```
/* Filename: dataAcquisition2.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

double timeInterval = 0.1; /* Seconds */

/* Record for 20 seconds */
double movementTime = 20;
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double angles2[numDataPoints];
array double angles3[numDataPoints];
array double angles4[numDataPoints];

/* Declare plotting variables */
CPlot plot;
double tolerance = 1.0; /* 1 degree for time shifting */

/* Set all joint speeds to 45 degrees/second */
mobot.setJointSpeeds(45, 45, 45, 45);

/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();

/* Start capturing data */
mobot.recordAngles(time, angles1, angles2, angles3, angles4, numDataPoints, timeInterval);

/* Perform the standing and unstanding motions */
mobot.motionTurnRight(360);
mobot.motionInchwormLeft(2);

/* Wait for recording to finish */
mobot.recordWait();

/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time,
          angles1,
          angles2,
          angles3,
          angles4);
```

```

plot.title("Unwrapped Data for Joint Angles versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2DCurve(time, angles1, numDataPoints);
plot.data2DCurve(time, angles2, numDataPoints);
plot.data2DCurve(time, angles3, numDataPoints);
plot.data2DCurve(time, angles4, numDataPoints);
plot.legend("Joint 1", 0);
plot.legend("Joint 2", 1);
plot.legend("Joint 3", 2);
plot.legend("Joint 4", 3);
plot.grid(PLOT_ON);
plot.plotting();

```

### 12.2.3 dataAcquisition2.ch Explained

Similar to the previous data acquisition demo, we begin by including header files, declaring our mobot variable, and connecting to the mobot:

```

#include <mobot.h>
#include <chplot.h>
#include <numeric.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

```

We also set our data acquisition interval, movement time, and calculate the number of data points:

```

double timeInterval = 0.1; /* Seconds */

/* Record for 20 seconds */
double movementTime = 20;
int numDataPoints = movementTime / timeInterval; /* Unitless */

```

For this motion, the value of 20 seconds was decided through trial-and-error. For complex motions such as inchworming, it is difficult to accurately estimate the amount of time the motion will take to complete. Through experimentation and experience, it was determined that 20 seconds allowed for a sufficient amount of time to record the turning motion and the inchworming motion.

Next, we declare our computational arrays for storing data and plotting, similar to the previous demo. One main difference is we declare a separate computational array to store data for each joint angle.

```

/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double angles2[numDataPoints];
array double angles3[numDataPoints];
array double angles4[numDataPoints];

/* Declare plotting variables */
CPlot plot1, plot2;
array double angles1_unwrapped[numDataPoints];
array double angles2_unwrapped[numDataPoints];

```

```

array double angles3_unwrapped[numDataPoints];
array double angles4_unwrapped[numDataPoints];
double tolerance = 1.0; /* 1 degree for time shifting */

```

Before we begin the motion, we set all the joint speeds to 45 degrees/second and move the mobot into its zero position.

```

/* Set all joint speeds to 45 degrees/second */
mobot.setJointSpeeds(45, 45, 45, 45);

/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();

```

Next, we begin recording the data.

```
mobot.recordAngles(time, angles1, angles2, angles3, angles4, numDataPoints, timeInterval);
```

Similar to `recordAngle()`, the recording process occurs in the background as the main program continues. We next execute our desired motions, which are to turn the mobot to the right by rotating each wheel 360 degrees, and then inchworming to the left twice:

```

mobot.motionTurnRight(360);
mobot.motionInchwormLeft(2);

```

Finally, we wait for the recording to finish.

```
mobot.recordWait();
```

Now that all of the data has been acquired, we unwrap, shift, and plot the data.

```

unwrapdeg(angles1_unwrapped, angles1);
unwrapdeg(angles2_unwrapped, angles2);
unwrapdeg(angles3_unwrapped, angles3);
unwrapdeg(angles4_unwrapped, angles4);
/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time,
          angles1_unwrapped,
          angles2_unwrapped,
          angles3_unwrapped,
          angles4_unwrapped);
plot2.title("Unwrapped Data for Joint Angles versus Time");
plot2.labelX(PLOT_AXIS_X, "Time (seconds)");
plot2.labelY(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2DCurve(time, angles1_unwrapped, numDataPoints);
plot2.data2DCurve(time, angles2_unwrapped, numDataPoints);
plot2.data2DCurve(time, angles3_unwrapped, numDataPoints);
plot2.data2DCurve(time, angles4_unwrapped, numDataPoints);
plot2.legend("Joint 1", 0);
plot2.legend("Joint 2", 1);
plot2.legend("Joint 3", 2);
plot2.legend("Joint 4", 3);
plot2.grid(PLOT_ON);
plot2.plotting();

```

Note that the previous call to `shiftTime()` appears different than the previous example. The `shiftTime()` function is able to accept any number of data arrays per time array to shift. The function tests the tolerance angle on all of the input data arrays, seeking the first occurrence of a movement greater than the tolerance for all of the data arrays. This makes the plotted motion begin right at time 0 on the plot.

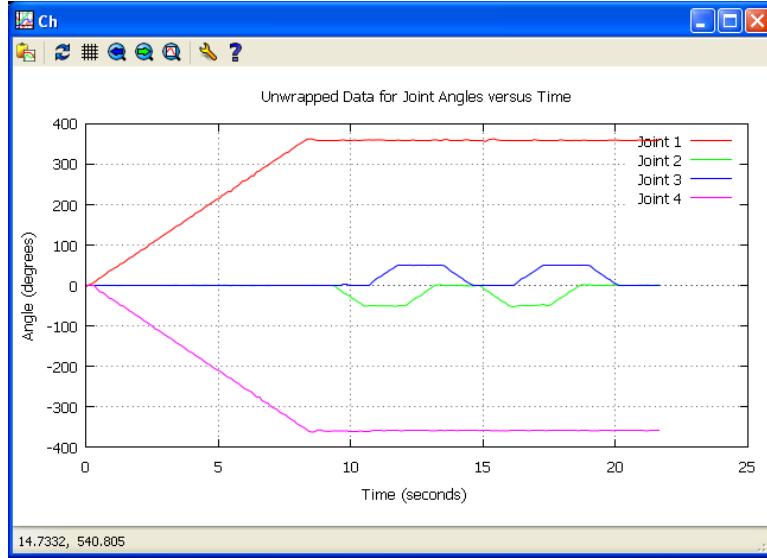


Figure 20: The unwrapped and shifted data recorded by `dataAcquisition2.ch`.

Figure 20 shows the captured data from the Mobot. During the first part of the motions, joints 1 and 4 move in opposite directions to rotate the mobot to the right. After rotating 360 degrees, joints 2 and 3 move to inchworm the mobot to the left twice.

## 12.3 Example 3

### 12.3.1 Problem Statement

A mobot is equipped with 3.5 inch diameter wheels. Write a piece of code which rolls the mobot forward at constant speed of 2.5 inches per second for a distance of 12 inches. Verify that the data is linear and follows the relation  $d = 2.5t$

### 12.3.2 `dataAcquisition3.ch` Source Code

```
/* Filename: dataAcquisition3.ch
 * Make a graph of the mobot's joint angle versus time */

#include <mobot.h>
#include <chplot.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();

double speed = 2.5; /* inches / second */
double distance = 12; /* inches */
```

```

double radius = 3.5/2.0; /* inches */
double angle = distance2angle(radius, distance); /* degrees */

/* Figure out how many data points we will need. First, figure out the
 * approximate amount of time the movement should take. */
double movementTime = distance / speed; /* Seconds */
/* Add an extra second of recording time to make sure the entire movement is
 * recorded */
movementTime = movementTime + 1;
double timeInterval = 0.1; /* seconds */
int numDataPoints = movementTime / timeInterval; /* Unitless */

/* Initialize the arrays to be used to store data */
array double time[numDataPoints];
array double angles1[numDataPoints];
array double distances[numDataPoints];

/* Declare plotting variables */
CPlot plot;
double tolerance = 1.0; /* Degrees */

/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();
/* Set mobot wheel speed */
mobot.setTwoWheelRobotSpeed(speed, radius);

/* Start capturing data */
mobot.recordAngle(MOBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

/* Roll the mobot the calculated distance */
mobot.motionRollForward(angle);

/* Wait for recording to finish */
mobot.recordWait();

/* Shift the data */
shiftTime(tolerance, numDataPoints, time, angles1);
/* Convert angles to displacement */
distances = angle2distance(radius, angles1);

/* Plot the data */
plot.title("Displacement versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Displacement (inches)");
plot.data2DCurve(time, distances, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();

```

### 12.3.3 dataAcquisition3.ch Explained

The first lines include header files and set up the mobot similar to the previous demos:

```
#include <mobot.h>
#include <chplot.h>
#include <numeric.h>
CMobot mobot;

/* Connect to the mobot */
mobot.connect();
```

Next, we initialize some variables. First, we make a variable to store the speed we want to move the mobot.

```
double speed = 2.5; /* inches / second */
```

Next, a variable for the distance.

```
double distance = 12; /* inches */
```

And a variable for the wheel radius...

```
double radius = 3.5/2.0; /* inches */
```

Now we wish to calculate the angle that the wheels need to turn in order to travel a distance of 12 inches. To do this, we use a function called `distance2angle()`. The `distance2angle()` function takes a wheel radius and the distance as arguments and returns the angle the wheel must turn to travel that distance.

```
double angle = distance2angle(radius, distance); /* degrees */
```

Internally, the angle in degrees is calculated by the formula

$$\theta = \left( \frac{d}{r} \right) * 180/\pi$$

where  $\theta$  is the angle in degrees,  $d$  is the distance travelled, and  $r$  is the radius of the wheel.

For the next step, we must calculate our estimated movement time, similar to the `dataAcquisition.ch` demo presented earlier.

```
double movementTime = distance / speed; /* Seconds */
```

In previous demos, we include an extra second in our movement time. We also calculate the number of data points based on the total movement time and the time interval.

```
movementTime = movementTime + 1;
double timeInterval = 0.1; /* seconds */
int numDataPoints = movementTime / timeInterval; /* Unitless */
```

Now, we allocate our computational arrays for storing data. We have arrays for the time, angles, and also distances.

```
array double time[numDataPoints];
array double angles1[numDataPoints];
array double distances[numDataPoints];

/* Declare plotting variables */
CPlot plot;
array double angles1Unwrapped[numDataPoints];
double tolerance = 1.0; /* Degrees */
```

Before beginning the motion, we move the mobot to zero position and set the wheel speeds.

```
/* Start the motion. First, move mobot to zero position */
mobot.moveToZero();
/* Set mobot wheel speed */
mobot.setTwoWheelRobotSpeed(speed, radius);
```

Start recording the angle of the first joint.

```
mobot.recordAngle(MOBOT_JOINT1, time, angles1, numDataPoints, timeInterval);
```

Move the mobot forward by the angle calculated earlier with `distance2angle()`.

```
mobot.motionRollForward(angle);
```

Wait for the recording to finish.

```
mobot.recordWait();
```

Finally, we want to unwrap and shift the data and convert the angle readings to a distance. The unwrap and shifting process is identical to Example 1 of this section. We convert the angles back to a distance using the `angle2distance()` function, which takes the radius of a wheel and the angle moved in degrees as arguments and return the distance traveled. The arguments can be single variables or computational arrays. If the angle argument is a computational array, the value returned is also a computational array. The `angle2distance()` function converts the angle to a distance using the following formula.

$$d = \left( \theta \frac{\pi}{180} \right) r$$

where  $d$  is the distance travelled,  $\theta$  is the angle rotated in degrees, and  $r$  is the radius of the wheel. The units of distance for  $d$  will be the same units as those chosen for  $r$ . For instance, if  $r$  is expressed in inches, the result

```
/* Unwrap the data */
unwrapdeg(angles1_unwrapped, angles1);
/* Shift the data */
shiftTime(tolerance, numDataPoints, time, angles1Unwrapped);
/* Convert angles to displacement */
distances = angle2distance(radius, angles1Unwrapped);
```

Finally, we create a plot of the data.

```
plot.title("Displacement versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Displacement (inches)");
plot.data2DCurve(time, distances, numDataPoints);
plot.grid(PLOT_ON);
plot.plotting();
```

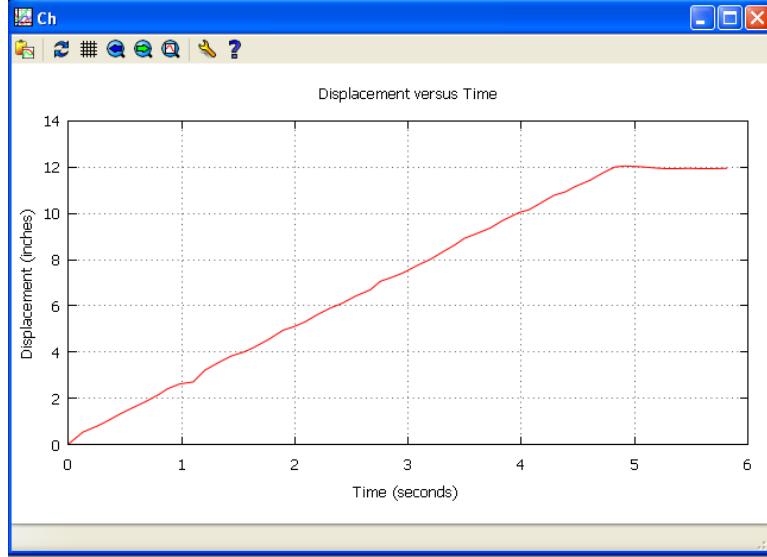


Figure 21: The unwrapped data recorded by `dataAcquisition2.ch`.

Figure 21 displays the distance data acquired during the motion. It can be seen that the mobot traveled to a distance of 12 inches, and then stopped. The expected relation between time and distance is the linear function

$$d = 2.5t$$

which may be verified on the graph. Furthermore, the time that it should take to reach 12 inches is

$$t = d/2.5 = 12/2.5 = 4.8 \text{ seconds} \quad (1)$$

which may also be verified from the graph.

## 13 Using the On-Board Buttons

### 13.1 Running the On-Board Joint Test program

The Mobot comes with a demo test program hard-coded into the on-board computer. The test program is designed to test the Mobot's joints range of motion and make sure everything is working properly. The test program will move the mobot continuously until the Mobot is shut off.

To run the test program, simply hold the "A" button for 3 seconds. After 3 seconds have passed, the blue LED should blink 3 times quickly, and the demo motion will begin.

### 13.2 Recalibrating the Mobot's Zero Position

A user may recalibrate the Mobot's zero positions if necessary. To recalibrate the Mobot, perform the following steps:

- Position the Mobot into its zero position. If the Mobot is currently actuating any of its joints, or if any joints feel "stiff", you may power the Mobot off and on again.
- While the Mobot is powered on, press both the "A" and "B" buttons simultaneously. The blue LED should flash quickly 3 times. The Mobot has now been recalibrated.

### **13.3 Testing the Mobot's Zero Position**

Button “B” may be used to toggle the mobot between holding its zero position or relaxing all of its joints. Press and hold the “B” button until the blue LED flashes. It should take no longer than 1 second. If the Mobot is currently relaxed, the Mobot will move to zero position and hold its position. If the “B” button is pressed again, the Mobot will relax its motors, allowing the user to reposition the Mobot.

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

## A CMobot Class

### A.1 Data Types

The data types defined in the header file `mobot.h` are described in this appendix. These data types are used by the Mbot library to represent certain values, such as joint id's and motor directions.

Data Type	Description
<code>mobotJointId_t</code>	An enumerated value that indicates a Mbot joint.
<code>mobotJointState_t</code>	The current state of a Mbot joint.

### A.2 `mobotJointId_t`

This datatype is an enumerated type used to identify a joint on the Mbot. Valid values for this type are:

```
typedef enum mobot_joints_e {
    MOBOT_JOINT1 = 1,
    MOBOT_JOINT2 = 2,
    MOBOT_JOINT3 = 3,
    MOBOT_JOINT4 = 4
} mobotJointId_t;
```

Value	Description
MOBOT_JOINT1	Joint number 1 on the Mbot, which is a faceplate joint.
MOBOT_JOINT2	Joint number 2 on the Mbot, which is a body joint.
MOBOT_JOINT3	Joint number 3 on the Mbot, which is a body joint.
MOBOT_JOINT4	Joint number 4 on the Mbot, which is a faceplate joint.

### A.3 `mobotJointState_t`

This datatype is an enumerated type used to designate the current movement state of a joint. The values may be retrieved from the mobot with the `getJointState()` function and may be set with the `moveContinuous()` family of functions. Valid values are:

```
typedef enum mobot_joint_state_e {
    MOBOT_NEUTRAL    = 0,
    MOBOT_FORWARD    = 1,
    MOBOT_BACKWARD   = 2,
    MOBOT_HOLD       = 3
} mobotJointState_t;
```

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

### A.4 CMobot API Functions

The header file `mobot.h` defines all the data types, macros and function prototypes for the mobot API library. The header file declares a class called `CMobot` which contains member functions which may be used to control the mobot.

Table 1: CMobot Member Functions.

Function	Description
<code>CMobot()</code>	The CMobot constructor function. This function is called automatically and should not be called explicitly.
<code>~CMobot()</code>	The CMobot destructor function. This function is called automatically and should not be called explicitly.
<code>blinkLED()</code>	Blink the LED on a Mobot module.
<code>connect()</code>	Connect to a remote mobot module. This function connects to the first mobot listed in the Barobo configuration file. To edit the configuration file, use the mobot control graphical user interface, and select the menu item “Mobot → Configure Mobot Bluetooth”.
<code>connectWithAddress()</code>	Connect to a mobot module by specifying its Bluetooth address.
<code>disconnect()</code>	Disconnect from a mobot module.
<code>disableButtonCallback()</code>	Reverts a Mobot’s buttons to their factory-default functions.
<code>driveToDirect()</code>	Move all four joints of the mobot to specified absolute angles.
<code>driveToDirectNB()</code>	Move all four joints of the mobot to specified absolute angles.
<code>enableButtonCallback()</code>	Enables a user-driven callback for handling button-press events on a module.
<code>getJointAngle()</code>	Get a joint’s angle.
<code>getJointAngles()</code>	Get joint angles for all joints.
<code>getJointMaxSpeed()</code>	Get a joint’s maximum speed in radians per second.
<code>getJointSafetyAngle()</code>	Get the mobot’s safety angle limit.
<code>getJointSafetyAngleTimeout()</code>	Get the mobot’s safety angle limit timeout.
<code>getJointSpeed()</code>	Get a motor’s current speed setting in radians per second.
<code>getJointSpeeds()</code>	Get all motor’s current speed settings in radians per second.
<code>getJointSpeedRatio()</code>	Get a motor’s speed as a ratio of the motor’s maximum speed.
<code>getJointSpeedRatios()</code>	Get all motor speeds as ratios of the motor’s maximum speed.
<code>getJointState()</code>	Get a motor’s current status.
<code>isConnected()</code>	This function is used to check the connection to a mobot.
<code>isMoving()</code>	This function is used to check if any joints are currently in motion.
<code>move()</code>	Move all four joints of the mobot by specified angles.
<code>moveNB()</code>	Identical to <code>move()</code> but non-blocking.
<code>moveContinuousNB()</code>	Move joints continuously. Joints will move until stopped.
<code>moveContinuousTime()</code>	Move joints continuously for a certain amount of time.
<code>moveJoint()</code>	Move a joint.
<code>moveJointContinuousNB()</code>	Move a joint continuously.
<code>moveJointNB()</code>	Move a joint.
<code>moveJointTo()</code>	Set the desired motor position for a joint.
<code>moveJointToNB()</code>	Identical to <code>moveJointTo()</code> but non-blocking.
<code>moveJointWait()</code>	Wait until the specified motor has stopped moving.
<code>moveTo()</code>	Move all four joints of the mobot to specified absolute angles.
<code>moveToNB()</code>	Identical to <code>moveTo()</code> but non-blocking.
<code>moveToZero()</code>	Instruct all motors to go to their zero positions.
<code>moveToZeroNB()</code>	Identical to <code>moveToZero()</code> but non-blocking.
<code>moveWait()</code>	Wait until all motors have stopped moving.
<code>recordAngle()</code>	Begin recording the angle of a joint.
<code>recordAngleBegin()</code>	Begin recording the angle of a joint.
<code>recordAngleEnd()</code>	End recording the angle of a joint.
<code>recordAngles()</code>	Begin recording the angle of all joints.
<code>recordAnglesBegin()</code>	Begin recording the angle of all joints.
<code>recordAnglesEnd()</code>	End recording the angle of all joints.
<code>recordWait()</code>	Wait for recording operations to finish.

Table 1: CMobot Member Functions (Continued)

Function	Description
<code>setJointSafetyAngle()</code>	Set the mobot's safety angle limit. The default value is 10 degrees.
<code>setJointSafetyAngleTimeout()</code>	Set the mobot's safety angle limit timeout. The default value is 0.5 seconds.
<code>setJointSpeed()</code>	Set a motor's speed setting in radians per second.
<code>setJointSpeeds()</code>	Set all motor speeds in radians per second.
<code>setJointSpeedRatio()</code>	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
<code>setJointSpeedRatios()</code>	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
<code>stopAllJoints()</code>	Stop all currently executing motions of the mobot.
<code>stopOneJoint()</code>	Stop a joint on the mobot.
<code>stopTwoJoints()</code>	Stop two joints on the mobot.
<code>stopThreeJoints()</code>	Stop three joints on the mobot.

Table 2: CMobot Member Functions for Compound Motions.

Compound Motions	These are convenience functions of commonly used compound motions.
<code>motionArch()</code> .....	Arch the mobot for better ground clearance.
<code>motionArchNB()</code> .....	Identical to <code>motionArch</code> but non-blocking.
<code>motionInchwormLeft()</code>	Inchworm motion towards the left.
<code>motionInchwormLeftNB()</code>	Identical to <code>motionInchwormLeft</code> but non-blocking.
 <code>motionInchwormRight()</code>	Inchworm motion towards the right.
 <code>motionInchwormRightNB()</code>	Identical to <code>motionInchwormRight</code> but non-blocking.
 <code>motionRollBackward()</code>	Roll on the faceplates toward the backward direction.
<code>motionRollBackwardNB()</code>	Identical to <code>motionRollBackward()</code> but non-blocking.
 <code>motionRollForward()</code> .	Roll on the faceplates forwards.
<code>motionRollForwardNB()</code>	Identical to <code>motionRollForward()</code> but non-blocking.
 <code>motionSkinny()</code> .....	Move the mobot into a skinny profile.
<code>motionSkinnyNB()</code> ....	Identical to <code>motionSkinny()</code> but non-blocking.
<code>motionStand()</code> .....	Stand the mobot up on its end.
<code>motionStandNB()</code> .....	Identical to <code>motionStand()</code> but non-blocking.
<code>motionTumbleLeft()</code> ..	Perform the tumbling motion.
<code>motionTumbleLeftNB()</code>	Identical to <code>motionTumbleLeft()</code> but non-blocking.
<code>motionTumbleRight()</code> .	Perform the tumbling motion in the opposite direction of “ <code>motionTumbleLeft</code> ”.
<code>motionTumbleRightNB()</code>	Identical to <code>motionTumbleRight()</code> but non-blocking.
 <code>motionTurnLeft()</code> ....	Rotate the mobot counterclockwise.
<code>motionTurnLeftNB()</code> ..	Identical to <code>motionTurnLeft()</code> but non-blocking.
<code>motionTurnRight()</code> ...	Rotate the mobot clockwise.
<code>motionTurnRightNB()</code> .	Identical to <code>motionTurnRight()</code> but non-blocking.
<code>motionWait()</code> .....	Wait for a motion to finish.

---

## CMobot::blinkLED()

**Synopsis**

```
#include <mobot.h>
int CMobot::blinkLED(double delay, int numBlinks);
```

**Purpose**

Blink the on-board LED on a Mobot module.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**delay** The amount of time between blinks.  
**numBlinks** The number of times to blink the LED.

**Description**

This function is used to blink or flash the LED on a Mobot module. The first argument, **delay**, is used to control the speed of the blinking LED, and the second argument, **numBlinks**, controls the number of times to blink. For instance, the line

```
mobot.blinkLED(0.1, 10);
```

would cause a mobot to blink 10 times fast, and the line

```
mobot.blinkLED(1, 2);
```

would cause the mobot to do two slow blinks.

**Example****See Also**

---

## CMobot::connect()

**Synopsis**

```
#include <mobot.h>
int CMobot::connect();
```

**Purpose**

Connect to a remote mobot via Bluetooth.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

None.

**Description**

This function is used to connect to a mobot. The function looks inside of a Barobo configuration file and connects to the first mobot listed in the file. The configuration file may be created and/or modified using the Mobot Controller Interface, and selecting the “Mobot → Configure Mobot Bluetooth” menu item.

**Example**

Please see the example in Section 5.1.2 on page 22.

**See Also**

`connectWithAddress()`, `disconnect()`

---

## CMobot::connectWithAddress()

**Synopsis**

```
#include <mobot.h>
int CMobot::connectWithAddress(char address[], int channel);
```

**Purpose**

Connect to a remote mobot via Bluetooth by specifying the specific Bluetooth address of the device.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

`address` The Bluetooth address of the mobot.

`channel` The Bluetooth channel that the listening program is listening on. The default channel is channel 1.

**Description**

This function is used to connect to a mobot.

**Example****See Also**

`connect()`, `disconnect()`

---

## CMobot::disconnect()

**Synopsis**

```
#include <mobot.h>
int CMobot::disconnect();
```

**Purpose**

Disconnect from a remote mobot.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

None.

**Description**

This function is used to disconnect from a mobot. A call to this function is not necessary before the termination of a program. It is only necessary if another connection will be established within the same program at a later time.

## Example

### See Also

`connect()`, `connectWithAddress()`

---

## `CMobot::driveJointToDirect()` `CMobot::driveJointToDirectNB()`

### Synopsis

```
#include <mobot.h>
int CMobot::driveJointToDirect(moboJointId_t id, double angle);
int CMobot::driveJointToDirectNB(moboJointId_t id, double angle);
```

### Purpose

Move all of the joints of a mobot to the specified positions.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<code>id</code>	The id of the joint to move.
<code>angle</code>	The absolute position to move the joint, expressed in degrees.

### Description

Note that this function is similar to the `moveJointTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

#### `CMobot::driveJointToDirect()`

This function moves all of the joints of a mobot to the specified absolute positions.

#### `CMobot::driveJointToDirectNB()`

This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveJointToDirectNB()` is the non-blocking version of the `driveJointToDirect()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

## Example

### See Also

---

## `CMobot::driveToDirect()` `CMobot::driveToDirectNB()`

### Synopsis

```
#include <mobot.h>
int CMobot::driveToDirect(double angle1, double angle2, double angle3, double angle4);
int CMobot::driveToDirectNB(double angle1, double angle2, double angle3, double angle4);
```

**Purpose**

Move all of the joints of a mobot to the specified positions.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

<code>angle1</code>	The absolute position to move joint 1, expressed in degrees.
<code>angle2</code>	The absolute position to move joint 2, expressed in degrees.
<code>angle3</code>	The absolute position to move joint 3, expressed in degrees.
<code>angle4</code>	The absolute position to move joint 4, expressed in degrees.

**Description**

Note that this function is similar to the `moveTo()` family of functions, except that this variant ignores the speed settings of the robot joints. This function causes the robot to move its joints towards its goal using a Proportional-Integral-Derivative (PID) controller.

**CMobot::driveToDirect()**

This function moves all of the joints of a mobot to the specified absolute positions.

**CMobot::driveToDirectNB()**

This function moves all of the joints of a mobot to the specified absolute positions.

The function `driveToDirectNB()` is the non-blocking version of the `driveToDirect()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

**Example****See Also**

---

**CMobot::getJointAngle()**  
**CMobot::getJointAngleAbs()**

**Synopsis**

```
#include <mobot.h>
int CMobot::getJointAngle(mobotJointId_t id, double &angle);
int CMobot::getJointAngleAbs(mobotJointId_t id, double &angle);
```

**Purpose**

Retrieve a mobot joint's current angle.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

<code>id</code>	The joint number. This is an enumerated type discussed in Section A.2 on page 70.
<code>angle</code>	A variable to store the current angle of the mobot motor. The contents of this variable will be overwritten with a value that represents the motor's angle in degrees.

### Description

This function gets the current motor angle of a mobot's motor. The angle returned is in units of degrees and is accurate to roughly  $\pm 0.17$  degrees.

The function `getJointAngle()` always returns an angle between -180 and +180 degrees. The `getJointAngleAbs()` function, however, gets the total angle the joint has turned since the mobot has been powered on. For instance, if the faceplate joint 1 has been rotated two full rotations after initial powerup, the function `getJointAngle()` will report that the joint is at angle 0, whereas the function `getJointAngleAbs()` will report that the joint angle is 720 degrees.

### Example

### See Also

---

## CMobot::getJointMaxSpeed()

### Synopsis

```
#include <mobot.h>
int CMobot::getJointMaxSpeed(mobotJointId_t id, double &speed);
```

### Purpose

Get the maximum speed of a joint on the mobot.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<code>id</code>	The joint number. This is an enumerated type discussed in Section A.2 on page 70.
<code>speed</code>	A variable of type <code>double</code> . The value of this variable will be overwritten with the maximum speed setting of the joint, which is in units of degrees per second.

### Description

This function is used to find the maximum speed setting of a joint. This is the maximum speed at which the joint will accept speed setting from the function `setJointSpeed()`. The values are in units of degrees per second.

### Example

### See Also

`getJointSpeed()`, `getJointMaxSpeedRatio()`, `setJointSpeed()`, `setJointSpeedRatio()`

---

## CMobot::getJointSafetyAngle()

### Synopsis

```
#include <mobot.h>
int CMobot::getJointSafetyAngle(double &degrees);
```

### Purpose

Get the current angle safety limit of the Mobot.

### Return Value

The function returns 0 on success and -1 on failure.

### Parameters

A variable which will be overwritten with the safety angle limit in degrees.

### Description

The Mobot 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 Mobot 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 retrieved using this function. The default setting is 10 degrees.

### Example

#### See Also

`getJointSafetyAngleTimeout()`, `setJointSafetyAngle()`, `setJointSafetyAngleTimeout()`

---

## CMobot::getJointSafetyAngleTimeout()

### Synopsis

```
#include <mobot.h>
int CMobot::getJointSafetyAngleTimeout(double &seconds);
```

### Purpose

Get the current angle safety limit timeout of the Mobot.

### Return Value

The function returns 0 on success and -1 on failure.

### Parameters

A variable which will be overwritten with the safety angle limit timeout in seconds.

### Description

The Mobot 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 Mobot will shut off all power to the motor after a certain period of time, in case it has hit an obstacle, or for any other reason. The period of time that the mobot waits before shutting the motor off is the joint safety angle timeout, which can be retrieved with this function. The default value for the timeout is 500 milliseconds, or 0.5 seconds.

### Example

#### See Also

`getJointSafetyAngle()`, `setJointSafetyAngle()`, `setJointSafetyAngleTimeout()`

---

## CMobot::getJointSpeed()

### Synopsis

```
#include <mobot.h>
int CMobot::getJointSpeed(mobotJointId_t id, double &speed);
```

### Purpose

Get the speed of a joint on the mobot.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

- id** The joint number to pose. This is an enumerated type discussed in Section A.2 on page 70.
- speed** A variable of type **double**. The value of this variable will be overwritten with the current speed setting of the joint, which is in units of degrees per second.

### **Description**

This function is used to find the current speed setting of a joint. This is the speed at which the joint will move when given motion commands. The values are in units of degrees per second.

### **Example**

#### **See Also**

`getJointMaxSpeed()`, `getJointSpeedRatio()`, `setJointSpeed()`, `setJointSpeedRatio()`

---

## **CMobot::getJointSpeedRatio()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::getJointSpeedRatio(mobotJointId_t id, double &ratio);
```

### **Purpose**

Get the speed ratio settings of a joint on the mobot.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

- id** Retrieve the speed ratio setting of this joint. This is an enumerated type discussed in Section A.2 on page 70.
- ratio** A variable of type **double**. The value of this variable will be overwritten with the current speed ratio setting of the joint.

### **Description**

This function is used to find the speed ratio setting of a joint. The speed ratio setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 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.

### **Example**

#### **See Also**

`setJointSpeeds()`, `getJointSpeedRatio()`, `getJointSpeed()`

---

## **CMobot::getJointSpeedRatios()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::getJointSpeedRatios(double &ratio1, double &ratio2, double &ratio3, double &ratio4);
```

**Purpose**

Get the speed ratio settings of all joints on the mobot.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

- ratio1** A variable to store the speed ratio of joint 1.
- ratio2** A variable to store the speed ratio of joint 2.
- ratio3** A variable to store the speed ratio of joint 3.
- ratio4** A variable to store the speed ratio of joint 4.

**Description**

This function is used to retrieve all four joint speed ratio settings of a mobot simultaneously. The speed ratios are as a value from 0 to 1.

**Example****See Also**

`setJointSpeeds()`, `getJointSpeedRatios()`, `getJointSpeed()`

---

## CMobot::getJointSpeeds()

**Synopsis**

```
#include <mobot.h>
int CMobot::getJointSpeeds(double &speed1, double &speed2, double &speed3, double &speed4);
```

**Purpose**

Get the speed settings of all joints on the mobot.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

- speed1** The joint speed setting for joint 1.
- speed2** The joint speed setting for joint 2.
- speed3** The joint speed setting for joint 3.
- speed4** The joint speed setting for joint 4.

**Description**

This function is used to retrieve all four joint speed settings of a mobot simultaneously. The speeds are in degrees per second.

**Example****See Also**

`setJointSpeeds()`, `getJointSpeedRatios()`, `getJointSpeed()`

---

## CMobot::getJointState()

**Synopsis**

```
#include <mobot.h>
int CMobot::getJointState(mobotJointId_t id, mobotJointState_t &state);
```

**Purpose**

Determine whether a motor is moving or not.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

- |                    |  |
|--------------------|--|
| <code>id</code>    | The joint number. This is an enumerated type discussed in Section A.2 on page 70.  |
| <code>state</code> | An integer variable which will be overwritten with the current state of the motor. This is an enumerated type discussed in Section A.3 on page 70. |

**Description**

This function is used to determine the current state of a motor. Valid states are listed below.

Value	Description
<code>MOBOT_NEUTRAL</code>	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
<code>MOBOT_FORWARD</code>	This value indicates that the joint is currently moving forward.
<code>MOBOT_BACKWARD</code>	This value indicates that the joint is currently moving backward.
<code>MOBOT_HOLD</code>	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

**Example****See Also**

`isMoving()`

---

## CMobot::isConnected()

**Synopsis**

```
#include <mobot.h>
int CMobot::isConnected();
```

**Purpose**

Check to see if currently connected to a remote mobot via Bluetooth.

**Return Value**

The function returns zero if it is not currently connected to a mobot or if an error has occurred, or 1 if the mobot is connected.

**Parameters**

None.

**Description**

This function is used to check if the software is currently connected to a mobot.

**Example****See Also**

`connect()`, `disconnect()`

---

## CMobot::isMoving()

**Synopsis**

```
#include <mobot.h>
int CMobot::isMoving();
```

**Purpose**

Check to see if a mobot is currently moving any of its joints.

**Return Value**

This function returns 0 if none of the joints are being driven or if an error has occurred, or 1 if any joint is being driven. A value of 1 is equivalent to either a joint state of MOBOT\_FORWARD or MOBOT\_BACKWARD from `getJointState()`

**Parameters**

None.

**Description**

This function is used to determine if a mobot is currently moving any of its joints.

**Example****See Also**

`getJointState()`

---

**CMobot::motionArch()**

**CMobot::motionArchNB()**

**Synopsis**

```
#include <mobot.h>
int CMobot::motionArch(double angle);
int CMobot::motionArchNB(double angle);
```

**Purpose**

Arch the mobot for more ground clearance.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**angle** The angle in degrees to arch. This number can range from 0 degrees, which is no arch, to 180 degrees, which is a fully curled up position.

**Description**

**CMobot::motionArch()**

This function causes the mobot to Arch up for better ground clearance while rolling.

**CMobot::motionArchNB()**

This function causes the mobot to Arch up for better ground clearance while rolling.

The non-blocking function, `motionArchNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobot::motionInchwormLeft()**  
**CMobot::motionInchwormLeftNB()**

**Synopsis**

```
#include <mobot.h>
int CMobot::motionInchwormLeft(int num);
int CMobot::motionInchwormLeftNB(int num);
```

**Purpose**

Perform the inch-worm gait to the left.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**num**        The number of times to perform the inchworm gait.

**Description**

**CMobot::motionInchwormLeft()**

This function causes the mobot to perform a single cycle of the inchworm gait to the left.

**CMobot::motionInchwormLeftNB()**

This function causes the mobot to perform a single cycle of the inchworm gait to the left.

This is the non-blocking version of the function **CMobot::motionInchwormLeft()**, meaning that the function will return immediately while the motion is performed asynchronously.

**See Also**

[motionInchwormRight\(\)](#)

---

**CMobot::motionInchwormRight()**  
**CMobot::motionInchwormRightNB()**

**Synopsis**

```
#include <mobot.h>
int CMobot::motionInchwormRight(int num);
int CMobot::motionInchwormRightNB(int num);
```

**Purpose**

Perform the inch-worm gait to the right.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**num**        The number of times to perform the inchworm gait.

## Description

### **CMobot::motionInchwormRight()**

This function causes the mobot to perform a single cycle of the inchworm gait to the right.

### **CMobot::motionInchwormRightNB()**

This function causes the mobot to perform a single cycle of the inchworm gait to the right.

This function has both a blocking and non-blocking version. The blocking version, `motionInchwormRight()`, will block until the mobot motion has completed. The non-blocking version, `motionInchwormRightNB()`, will return immediately, and the motion will be performed asynchronously.

## See Also

`motionInchwormLeft()`

---

### **CMobot::motionRollBackward()**

### **CMobot::motionRollBackwardNB()**

## Synopsis

```
#include <mobot.h>
int CMobot::motionRollBackward(double angle);
int CMobot::motionRollBackwardNB(double angle);
```

## Purpose

Use the faceplates as wheels to roll backward.

## Return Value

The function returns 0 on success and non-zero otherwise.

## Parameters

`angle`      The angle to turn the wheels, specified in degrees.

## Description

### **CMobot::motionRollBackward()**

This function causes each of the faceplates to rotate to roll the mobot backward. The amount to roll the wheels is specified by the argument, `angle`.

### **CMobot::motionRollBackwardNB()**

This function causes each of the faceplates to rotate to roll the mobot backward. The amount to roll the wheels is specified by the argument, `angle`.

This function has both a blocking and non-blocking version. The blocking version, `motionRollBackward()`, will block until the mobot motion has completed. The non-blocking version, `motionRollBackwardNB()`, will return immediately, and the motion will be performed asynchronously.

## See Also

`motionRollForward()`

---

### **CMobot::motionRollForward()**

### **CMobot::motionRollForwardNB()**

## Synopsis

```
#include <mobot.h>
int CMobot::motionRollForward(double angle);
int CMobot::motionRollForwardNB(double angle);
```

### Purpose

Use the faceplates as wheels to roll forward.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

**angle** The angle to turn the wheels, specified in degrees.

### Description

#### **CMobot::motionRollForward()**

This function causes each of the faceplates to rotate to roll the mobot forward. The amount to roll the wheels is specified by the argument, **angle**.

#### **CMobot::motionRollForwardNB()**

This function causes each of the faceplates to rotate to roll the mobot forward. The amount to roll the wheels is specified by the argument, **angle**.

This function has both a blocking and non-blocking version. The blocking version, **motionRollForward()**, will block until the mobot motion has completed. The non-blocking version, **motionRollForwardNB()**, will return immediately, and the motion will be performed asynchronously.

### See Also

**motionRollBackward()**

---

## **CMobot::motionSkinny()**

### **CMobot::motionSkinnyNB()**

### Synopsis

```
#include <mobot.h>
int CMobot::motionSkinny(double angle);
int CMobot::motionSkinnyNB(double angle);
```

### Purpose

Move the mobot into a skinny profile.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

**angle** The angle in degrees to move the joints. A value of zero means a completely flat profile, while a value of 90 degrees means a fully skinny profile.

### Description

**CMobot::motionSkinny()**

This function makes the mobot assume a skinny rolling profile.

**CMobot::motionSkinnyNB()**

This function makes the mobot assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the mobot motion has completed. The non-blocking version, `motionSkinnyNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobot::motionStand()****CMobot::motionStandNB()****Synopsis**

```
#include <mobot.h>
int CMobot::motionStand();
int CMobot::motionStandNB();
```

**Purpose**

Stand the mobot up on a faceplate.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

None.

**Description****CMobot::motionStand()**

This function causes the mobot to motionStand up into the camera platform.

**CMobot::motionStandNB()**

This function causes the mobot to motionStand up into the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionStand()`, will block until the mobot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobot::motionTumbleLeft()****CMobot::motionTumbleLeftNB()****Synopsis**

```
#include <mobot.h>
int CMobot::motionTumbleLeft(int num);
int CMobot::motionTumbleLeftNB(int num);
```

**Purpose**

Make the mobot tumble end over end.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

`num` The number of times to tumble.

**Description****CMobot::motionTumbleLeft()**

This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

**CMobot::motionTumbleLeftNB()**

This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleLeft()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleLeftNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobot::motionTumbleRight()****CMobot::motionTumbleRightNB()****Synopsis**

```
#include <mobot.h>
int CMobot::motionTumbleRight(int num);
int CMobot::motionTumbleRightNB(int num);
```

**Purpose**

Make the mobot tumble end over end.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

`num` The number of times to tumble.

**Description****CMobot::motionTumbleRight()**

This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

**CMobot::motionTumbleRightNB()**

This causes the mobot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, `motionTumbleRight()`, will block until the mobot motion has completed. The non-blocking version, `motionTumbleRightNB()`, will return immediately, and the motion will be performed asynchronously.

#### See Also

---

### `CMobot::motionTurnLeft()` `CMobot::motionTurnLeftNB()`

#### Synopsis

```
#include <mobot.h>
int CMobot::motionTurnLeft(double angle);
int CMobot::motionTurnLeftNB(double angle);
```

#### Purpose

Rotate the mobot using the faceplates as wheels.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

`angle` The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the left.

#### Description

##### `CMobot::motionTurnLeft()`

This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

##### `CMobot::motionTurnLeftNB()`

This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnLeft()`, will block until the mobot motion has completed. The non-blocking version, `motionTurnLeftNB()`, will return immediately, and the motion will be performed asynchronously.

#### See Also

### `motionTurnRight()`

---

### `CMobot::motionTurnRight()` `CMobot::motionTurnRightNB()`

#### Synopsis

```
#include <mobot.h>
int CMobot::motionTurnRight(double angle);
int CMobot::motionTurnRightNB(double angle);
```

**Purpose**

Rotate the mobot using the faceplates as wheels.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**angle** The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the right.

**Description****CMobot::motionTurnRight()**

This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

**CMobot::motionTurnRightNB()**

This function causes the mobot to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnRight()`, will block until the mobot motion has completed. The non-blocking version, `motionTurnRightNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

`motionTurnRight()`

---

**CMobot::motionUnstand()****CMobot::motionUnstandNB()****Synopsis**

```
#include <mobot.h>
int CMobot::motionUnstand();
int CMobot::motionUnstandNB();
```

**Purpose**

Move a mobot currently standing on a faceplate back down into a prone position.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

None.

**Description****CMobot::motionUnstand()**

This function causes the mobot to move down from the camera platform.

**CMobot::motionUnstandNB()**

This function causes the mobot to move down from the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionUnstand()`, will block until the mobot motion has completed. The non-blocking version, `motionUnstandNB()`,

will return immediately, and the motion will be performed asynchronously.

#### See Also

---

## CMobot::motionWait()

#### Synopsis

```
#include <mobot.h>
int CMobot::motionWait();
```

#### Purpose

Wait for a motion to complete execution.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Description

This function is used to wait for a motion function to fully complete its cycle. The CMobot motion functions are those member functions which begin with “motion” as part of their name, such as `motionInchwormLeft()`.

#### Example

#### See Also

---

## CMobot::move() CMobot::moveNB()

#### Synopsis

```
#include <mobot.h>
int CMobot::move(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveNB(double angle1, double angle2, double angle3, double angle4);
```

#### Purpose

Move all of the joints of a mobot by specified angles.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

<code>angle1</code>	The amount to move joint 1, expressed in degrees, relative to the current position.
<code>angle2</code>	The amount to move joint 2, expressed in degrees, relative to the current position.
<code>angle3</code>	The amount to move joint 3, expressed in degrees, relative to the current position.
<code>angle4</code>	The amount to move joint 4, expressed in degrees, relative to the current position.

#### Description

##### CMobot::move()

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

## **CMobot::moveNB()**

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

The function `moveNB()` is the non-blocking version of the `move()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 9 on page 36.

### **Example**

Please see the demo at Section 5.1.2 on page 22.

### **See Also**

---

## **CMobot::moveContinuousNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveContinuousNB(
    mobotJointState_t dir1,
    mobotJointState_t dir2,
    mobotJointState_t dir3,
    mobotJointState_t dir4);
```

### **Purpose**

Move the joints of a mobot continuously in the specified directions.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

Each parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

### **Description**

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

### **Example**

### **See Also**

---

## CMobot::moveContinuousTime()

### Synopsis

```
#include <mobot.h>
int CMobot::moveContinuousTime( mobotJointState_t dir1,
                                mobotJointState_t dir2,
                                mobotJointState_t dir3,
                                mobotJointState_t dir4,
                                double seconds);
```

### Purpose

Move the joints of a mobot continuously in the specified directions for some amount of time.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

Each direction parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

The `seconds` parameter is the time to perform the movement, in seconds.

### Description

This function causes joints of a mobot to begin moving. The joints will continue moving until the joint hits a joint limit, or the time specified in the `seconds` parameter is reached. This function will block until the motion is completed.

### Example

### See Also

---

## CMobot::moveJoint() CMobot::moveJointNB()

### Synopsis

```
#include <mobot.h>
int CMobot::moveJoint(mobotJointId_t id, double angle);
int CMobot::moveJointNB(mobotJointId_t id, double angle);
```

### Purpose

Move a joint on the mobot by a specified angle with respect to the current position.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id** The joint number to move.

**angle** The angle in degrees to move the motor relative to its current position.

### **Description**

#### **CMobot::moveJoint()**

This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJoint()` function.

#### **CMobot::moveJointNB()**

This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointNB()` function.

The function `moveJointNB()` is the non-blocking version of the `moveJoint()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

### **Example**

Please see the example in Section 5.1.2 on page 22.

### **See Also**

`connectWithAddress()`

---

## **CMobot::moveJointContinuousNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveJointContinuousNB(mobotJointId_t id, mobotJointState_t dir1);
```

### **Purpose**

Move a joint of a mobot continuously in the specified directions.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id** The joint number to move.

**dir** This parameter specifies the direction the joint should move.

### **Description**

The `dir` parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

### Example

### See Also

---

## CMobot::moveJointContinuousTime()

### Synopsis

```
#include <mobot.h>
int CMobot::moveJointContinuousTime(mobotJointId_t id, mobotJointState_t dir1, double seconds);
```

### Purpose

Move a joint of a mobot continuously in the specified directions for a certain amount of time.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

- id** The joint number to move.
- dir** This parameter specifies the direction the joint should move.
- seconds** The number of seconds to rotate the joint.

### Description

The **dir** parameter specifies the direction the joint should move. The types are enumerated in **mobot.h** and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, the joint is stopped by setting the speed to zero, or the

specified amount of time has expired.

### Example

### See Also

---

## CMobot::moveJointTo() CMobot::moveJointToNB()

### Synopsis

```
#include <mobot.h>
int CMobot::moveJointTo(mobotJointId_t id, double angle);
int CMobot::moveJointToNB(mobotJointId_t id, double angle);
```

### Purpose

Move a joint on the mobot to an absolute position.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

**id**      The joint number to wait for.  
**angle**    The absolute angle in degrees to move the motor to.

### Description

#### CMobot::moveJointTo()

This function commands the motor to move to a position specified in radians at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointTo()` function.

#### CMobot::moveJointToNB()

This function commands the motor to move to a position specified in radians at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToNB()` function.

The function `moveJointToNB()` is the non-blocking version of the `moveJointTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

### Example

Please see the example in Section 5.1.2 on page 22.

### See Also

`connectWithAddress()`

---

## CMobot::moveJointToAbs()

## **CMobot::moveJointToAbsNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveJointToAbs(mobotJointId_t id, double angle);
int CMobot::moveJointToAbsNB(mobotJointId_t id, double angle);
```

### **Purpose**

Move a joint on the mobot to an absolute position.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id**      The joint number to wait for.  
**angle**    The absolute angle in degrees to move the motor to.

### **Description**

#### **CMobot::moveJointToAbs()**

This function commands the motor to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToAbs()` function.

#### **CMobot::moveJointToAbsNB()**

This function commands the motor to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToAbsNB()` function.

The function `moveJointToAbsNB()` is the non-blocking version of the `moveJointToAbs()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

The difference between this function and its sister function, `moveJointTo()`, is that this function takes into consideration any full rotations a joint has experienced. For instance, consider the following code segment:

```
mobot.moveJointTo(MOBOT_JOINT1, 375);
mobot.moveJointTo(MOBOT_JOINT1, 0);
```

The first line in the above code segment will cause joint 1 to rotate one full rotation in the positive direction, plus an additional 15 degrees. The second line will cause the same joint to rotate backwards 15 degrees to reach its "0" goal.

This following code segment, however, differs.

```
mobot.moveJointToAbs(MOBOT_JOINT1, 375);
mobot.moveJointToAbs(MOBOT_JOINT1, 0);
```

The first line will have the same effect as the first code segment, rotating joint 1 375 degrees, assuming it is starting at 0. The second line, however, will cause joint 1 to rotate the full 375 degrees backwards back to its original location.

### **Example**

Please see the example in Section 5.1.2 on page 22.

### **See Also**

`connectWithAddress()`

---

**CMobot::moveJointToDirect()**  
**CMobot::moveJointToDirectNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveJointToDirect(mobotJointId_t id, double angle);
int CMobot::moveJointToDirectNB(mobotJointId_t id, double angle);
```

### **Purpose**

Move a joint on the mobot to an absolute position taking the shortest direction possible.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id**      The joint number to wait for.  
**angle**    The absolute angle in degrees to move the motor to.

### **Description**

#### **CMobot::moveJointToDirect()**

This function commands the motor to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToDirect()` function.

#### **CMobot::moveJointToDirectNB()**

This function commands the motor to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointToDirectNB()` function.

The function `moveJointToDirectNB()` is the non-blocking version of the `moveJointToDirect()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

Note that the main difference between this function and its sister function, `moveJointTo()`, is that this function explicitly moves the motor in whichever direction is the shortest to achieve its goal.

### **Example**

Please see the example in Section 5.1.2 on page 22.

### **See Also**

`connectWithAddress()`

---

## **CMobot::moveJointWait()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveJointWait(mobotJointId_t id);
```

### **Purpose**

Wait for a joint to stop moving.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id**      The joint number to wait for.

### **Description**

This function is used to wait for a joint motion to finish. Functions such as `moveNB()` and `moveJointNB()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveJointWait()` function is used to wait for a mobotic joint motion to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

### **Example**

Please see the example in Section 5.1.2 on page 22.

### **See Also**

`moveWait()`

---

## **CMobot::moveTo() CMobot::moveToNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobot::moveTo(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveToNB(double angle1, double angle2, double angle3, double angle4);
```

### **Purpose**

Move all of the joints of a mobot to the specified positions.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**angle1**      The absolute position to move joint 1, expressed in degrees.  
**angle2**      The absolute position to move joint 2, expressed in degrees.  
**angle3**      The absolute position to move joint 3, expressed in degrees.  
**angle4**      The absolute position to move joint 4, expressed in degrees.

### **Description**

### **CMobot::moveTo()**

This function moves all of the joints of a mobot to the specified absolute positions.

### **CMobot::moveToNB()**

This function moves all of the joints of a mobot to the specified absolute positions.

The function `moveToNB()` is the non-blocking version of the `moveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

#### **Example**

Please see the demo at Section 5.1.2 on page 22.

#### **See Also**

---

### **CMobot::moveToAbs()**

### **CMobot::moveToAbsNB()**

#### **Synopsis**

```
#include <mobot.h>
int CMobot::moveToAbs(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveToAbsNB(double angle1, double angle2, double angle3, double angle4);
```

#### **Purpose**

Move all of the joints of a mobot to the specified positions.

#### **Return Value**

The function returns 0 on success and non-zero otherwise.

#### **Parameters**

- |                     |  |
|---------------------|--|
| <code>angle1</code> | The absolute position to move joint 1, expressed in degrees. |
| <code>angle2</code> | The absolute position to move joint 2, expressed in degrees. |
| <code>angle3</code> | The absolute position to move joint 3, expressed in degrees. |
| <code>angle4</code> | The absolute position to move joint 4, expressed in degrees. |

#### **Description**

### **CMobot::moveToAbs()**

This function moves all of the joints of a mobot to the specified absolute positions.

### **CMobot::moveToAbsNB()**

This function moves all of the joints of a mobot to the specified absolute positions.

The function `moveToAbsNB()` is the non-blocking version of the `moveToAbs()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

The difference between this function and its sister function, `moveTo()`, is that this function takes into consideration any full rotations a joint has experienced. For instance, consider the following code segment:

```
mobot.moveTo(375, 0, 0, 0);
mobot.moveTo(0, 0, 0, 0);
```

The first line in the above code segment will cause joint 1 to rotate one full rotation in the positive direction, plus an additional 15 degrees. The second line will cause the same joint to rotate backwards 15 degrees to reach its "0" goal.

This following code segment, however, differs.

```
mobot.moveToAbs(375, 0, 0, 0);
mobot.moveToAbs(0, 0, 0, 0);
```

The first line will have the same effect as the first code segment, rotating joint 1 375 degrees, assuming it is starting at 0. The second line, however, will cause joint 1 to rotate the full 375 degrees backwards back to its original location.

### Example

Please see the demo at Section 5.1.2 on page 22.

### See Also

---

**CMobot::moveToDirect()**  
**CMobot::moveToDirectNB()**

### Synopsis

```
#include <mobot.h>
int CMobot::moveToDirect(double angle1, double angle2, double angle3, double angle4);
int CMobot::moveToDirectNB(double angle1, double angle2, double angle3, double angle4);
```

### Purpose

Move all of the joints of a mobot to the specified positions.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<b>angle1</b>	The absolute position to move joint 1, expressed in degrees.
<b>angle2</b>	The absolute position to move joint 2, expressed in degrees.
<b>angle3</b>	The absolute position to move joint 3, expressed in degrees.
<b>angle4</b>	The absolute position to move joint 4, expressed in degrees.

### Description

#### **CMobot::moveToDirect()**

This function moves all of the joints of a mobot to the specified absolute positions.

#### **CMobot::moveToDirectNB()**

This function moves all of the joints of a mobot to the specified absolute positions.

The function `moveToDirectNB()` is the non-blocking version of the `moveToDirect()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

Note that the main difference between this function and its sister function, `moveJointTo()`, is that this function explicitly moves the motor in whichever direction is the shortest to achieve its goal.

#### Example

The `moveToDirect()` function moves the joint to the requested position choosing the shortest distance. For instance, consider the following function calls:

```
mobot.moveTo(350, 0, 0, 0);
```

When the preceding line of code is executed, joint 1 on the mobot will rotate 350 degrees in the forward direction. However, in the following line,

```
mobot.moveToDirect(350, 0, 0, 0);
```

the first joint of the mobot will actually rotate -10 degrees in the negative direction to achieve the desired final position. Instead of rotating the entire 350 degrees, the mobot calculates that a movement of -10 degrees will require less traveled distance to achieve the same final position.

#### See Also

---

**CMobot::moveToZero()**  
**CMobot::moveToZeroNB()**

#### Synopsis

```
#include <mobot.h>
int CMobot::moveToZero();
int CMobot::moveToZeroNB();
```

#### Purpose

Move all of the joints of a mobot to their zero position.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

None.

#### Description

##### **CMobot::moveToZero()**

This function moves all of the joints of a mobot to their zero position.

##### **CMobot::moveToZeroNB()**

This function moves all of the joints of a mobot to their zero position.

The function `moveToZeroNB()` is the non-blocking version of the `moveToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

#### Example

Please see the demo at Section 5.1.2 on page 22.

## See Also

---

## CMobot::moveWait()

### Synopsis

```
#include <mobot.h>
int CMobot::moveWait();
```

### Purpose

Wait for all joints to stop moving.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Description

This function is used to wait for all joint motions to finish. Functions such as `move()` and `moveTo()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` function is used to wait for mobotic motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

### Example

See the sample program in Section 5.1.2 on page 22.

### See Also

`moveWait()`, `moveJointWait()`

---

## CMobot::recordAngle()

### Synopsis

```
#include <mobot.h>
int CMobot::recordAngle(mobotJointId_t id, double time[], double angle[], int num, double seconds);
```

### Purpose

Record joint angle data for a joint for a set amount of time at a specified time interval.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<code>id</code>	The joint number. This is an enumerated type discussed in Section A.2 on page 70.
<code>time</code>	An array which will store time values for each of the angle readings.
<code>angle</code>	An array which will store the angle values for each time.
<code>num</code>	The size of the arrays.
<code>seconds</code>	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.

### Description

This function is used to accurately record the motion of a Mobot joint at a relatively fast rate. The function will fill the `time` and `angle` arrays with data at the rate specified by `seconds`. If the communication speed cannot maintain the requested rate, (if `msecs` is too low, in other words), the function will collect data as

fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

The length of time to collect the data can be calculated by the formula

$$\text{Total Time} = (\text{num} \times \text{seconds})$$

This function is a non-blocking function. After calling this function, a call to `recordWait()` should be performed to ensure that the data has been fully collected.

#### Example

#### See Also

`recordAngles()`, `recordWait()`

---

## CMobot::recordAngleBegin()

#### Synopsis

```
#include <mobot.h>
int CMobot::recordAngleBegin(mobotJointId_t id, mobotRecordData_t &time, mobotRecordData_t &angle, double seconds);
```

#### Purpose

Record joint angle data for a joint for a set amount of time at a specified time interval.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

<code>id</code>	The joint number. This is an enumerated type discussed in Section A.2 on page 70.
<code>time</code>	A variable which will store time values for each of the angle readings.
<code>angle</code>	A variable which will store the angle values for each time.
<code>seconds</code>	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.

#### Description

This function is used in conjunction with the `recordAngleEnd()` function to accurately record the motion of a Mobot joint at a relatively fast rate.

The function will allocate memory and fill the `time` and `angle` variables with data at the rate specified by `seconds`.

If the communication speed cannot maintain the requested rate, (if `seconds` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

When the `recordAngleBegin()` function is called, the Mobot library will immediately begin recording angle data. The `recordAngleEnd()` function must be called to halt the recording process.

#### Example

#### See Also

`recordAngleEnd()`

---

## CMobot::recordAngleEnd()

#### Synopsis

```
#include <mobot.h>
int CMobot::recordAngleEnd(mobotJointId_t id, int &num);
```

### Purpose

End a joint recording process that is currently running.

### Return Value

This function returns zero on success and -1 on failure.

### Parameters

<b>id</b>	The joint number. This is an enumerated type discussed in Section A.2 on page 70.
<b>num</b>	An integer variable that will be set to the number of recorded elements.

### Description

This function is used in conjunction with the `recordAngleBegin()` function. This function stops the recording process and returns the number of valid data points allocated for the arrays.

### Example

#### See Also

`recordAngleBegin()`

---

## CMobot::recordAngles()

### Synopsis

```
#include <mobot.h>
int CMobot::recordAngles(double time[], double angle1[], double angle2[],
                         double angle3[], double angle4[], int num,
                         double seconds);
```

### Purpose

Record joint angle data for all joint for a set amount of time at a specified time interval.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<b>time</b>	An array which will store time values for each of the angle readings.
<b>angle1</b>	An array which will store the angle values for joint 1.
<b>angle2</b>	An array which will store the angle values for joint 2.
<b>angle3</b>	An array which will store the angle values for joint 3.
<b>angle4</b>	An array which will store the angle values for joint 4.
<b>num</b>	The number of elements of the arrays.
<b>seconds</b>	The number of seconds between angle readings.

### Description

This function is used to accurately record the motion of a Mobot joint at a relatively fast rate. The function will fill the `time`, `angle1`, `angle2`, `angle3`, and `angle4` arrays with data at the rate specified by `seconds`. A typical value for `seconds` is 0.1, or polling 10 times a second. If the communication speed cannot maintain the requested rate, (if `msecs` is too low, in other words), the function will collect data as fast as possible.

The lowest allowable rate is 0.05, however there is no guarantee that data will actually be collected at that rate, due to communication noise, distance to the module, etc.

The length of time to collect the data can be calculated by the formula

$$\text{Total Time} = (\text{num} \times \text{seconds})$$

This function is a non-blocking function. After calling this function, a call to `recordWait()` should be performed to ensure that the data has been fully collected.

#### Example

#### See Also

`recordAngle()`, `recordWait()`

---

## CMobot::recordAnglesBegin()

#### Synopsis

```
#include <mobot.h>
int CMobot::recordAnglesBegin(double* &time,
                               double* &angle1,
                               double* &angle2,
                               double* &angle3,
                               double* &angle4,
                               double seconds);
```

#### Purpose

Record joint angle data for all joints for a set amount of time at a specified time interval.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

<code>time</code>	A pointer to an array which will store time values for each of the angle readings.
<code>angle1</code>	A pointer to an array which will store the angle values for each time.
<code>angle2</code>	...
<code>angle3</code>	...
<code>angle4</code>	...
<code>seconds</code>	The number of seconds between angle readings. The minimum value allowed for this variable is 0.05.

#### Description

This function is used in conjunction with the `recordAnglesEnd()` function to accurately record the motion of a Mbot joint at a relatively fast rate.

The function will allocate memory and fill the `time` and `angle` arrays with data at the rate specified by `seconds`. The `time` and `angle` arrays should be freed by the user to prevent memory leaks.

If the communication speed cannot maintain the requested rate, (if `seconds` is too low, in other words), the function will collect data as fast as possible. The minimum value for `seconds` is 0.05, but the actual minimum time will depend on other factors, such as communication noise and distance to the mobot.

When the `recordAnglesBegin()` function is called, the Mbot library will immediately begin recording angle data. The `recordAngleEnd()` function must be called to halt the recording process.

#### Example

**See Also**

`recordAngleEnd()`

---

## CMobot::recordAnglesEnd()

**Synopsis**

```
#include <mobot.h>
int CMobot::recordAnglesEnd(int &num);
```

**Purpose**

End a joint recording process that is currently running.

**Return Value**

This function returns 0 on success or -1 on failure.

**Parameters**

`num` An integer variable that will be set to the number of recorded elements.

**Description**

This function is used in conjunction with the `recordAngleBegin()` function. This function stops the recording process and returns the number of valid data points allocated for the arrays.

**Example**

**See Also**

`recordAngleBegin()`

---

## CMobot::recordWait()

**Synopsis**

```
#include <mobot.h>
int CMobot::recordWait();
```

**Purpose**

Wait for a joint recording operation to finish.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

None

**Description**

This function is used in conjunction with the `recordAngle()` function and/or the `recorgAngles()` function. The `recordAngle()` and `recordAngles()` functions both initiate a recording operation that runs in the background for a certain amount of time. The `recordWait()` function is used to pause the main program until the recording operation has finished.

**Example**

**See Also**

`recordAngle()`, `recordAngles()`

---

## CMobot::setJointSafetyAngle()

**Synopsis**

```
#include <mobot.h>
int CMobot::setJointSafetyAngle(double degrees);
```

**Purpose**

Set the current angle safety limit of the Mobot.

**Return Value**

The function returns 0 on success and -1 on failure.

**Parameters**

The requested joint safety angle limit for the Mobot.

**Description**

The Mobot 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 Mobot 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 this function. The default setting is 10 degrees. Higher values indicate “less safe” behavior of the Mobot because the Mobot will not engage safety protocols until the joint has deviated by a greater amount. Values greater than 90 degrees effectively disengage the Mobot’s safety protocols altogether, and this function should be used with care.

**Example****See Also**

`getJointSafetyAngle()`, `getJointSafetyAngleTimeout()`, `setJointSafetyAngleTimeout()`

---

## CMobot::setJointSafetyAngleTimeout()

**Synopsis**

```
#include <mobot.h>
int CMobot::setJointSafetyAngleTimeout(double seconds);
```

**Purpose**

Set the current angle safety limit timeout of the Mobot.

**Return Value**

The function returns 0 on success and -1 on failure.

**Parameters**

A variable which will be overwritten with the safety angle limit timeout in seconds.

**Description**

The Mobot 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 Mobot will shut off all power to the motor after a certain period of time, in case it has hit an obstacle, or for any other reason. The period of time that

the mobot waits before shutting the motor off is the joint safety angle timeout, which can be set with this function. The default value for the timeout is 500 milliseconds, or 0.5 seconds.

#### Example

#### See Also

`getJointSafetyAngle()`, `setJointSafetyAngle()`, `getJointSafetyAngleTimeout()`

---

## CMobot::setJointSpeed()

#### Synopsis

```
#include <mobot.h>
int CMobot::setJointSpeed(mobotJointId_t id, double speed);
```

#### Purpose

Set the speed of a joint on the mobot.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

**id** The joint number to pose.  
**speed** A variable of type `double` for the requested average angular speed in degrees per second.

#### Description

This function is used to set the angular speed of a joint of a mobot. The maximum possible angular speed for a particular joint may be obtained by using the function `getJointMaxSpeed()`.

#### Example

#### See Also

---

## CMobot::setJointSpeedRatio()

#### Synopsis

```
#include <mobot.h>
int CMobot::setJointSpeedRatio(mobotJointId_t id, double ratio);
```

#### Purpose

Set the speed ratio settings of a joint on the mobot.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

**id** Set the speed ratio setting of this joint. This is an enumerated type discussed in Section A.2 on page 70.  
**ratio** A variable of type `double` with a value from 0 to 1.

#### Description

This function is used to set the speed ratio setting of a joint. The speed ratio setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 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.

### Example

#### See Also

`setJointSpeeds()`, `setJointSpeedRatio()`, `getJointSpeed()`

---

## CMobot::setJointSpeedRatios()

#### Synopsis

```
#include <mobot.h>
int CMobot::setJointSpeedRatios(double ratio1, double ratio2, double ratio3, double ratio4);
```

#### Purpose

Set the speed ratio settings of all joints on the mobot.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

- `ratio1` The speed ratio setting for the first joint.
- `ratio2` The speed ratio setting for the second joint.
- `ratio3` The speed ratio setting for the third joint.
- `ratio4` The speed ratio setting for the fourth joint.

#### Description

This function is used to simultaneously set the angular speed ratio settings of all four joints of a mobot. The speed ratio is a percentage of the maximum speed of a joint, expressed in a value from 0 to 1.

### Example

#### See Also

`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

---

## CMobot::setJointSpeeds()

#### Synopsis

```
#include <mobot.h>
int CMobot::setJointSpeeds(double speed1, double speed2, double speed3, double speed4);
```

#### Purpose

Set the speed settings of all joints on the mobot.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

- `speed1` The speed for the first joint, in degrees per second.
- `speed2` The speed for the second joint, in degrees per second.
- `speed3` The speed for the third joint, in degrees per second.
- `speed4` The speed for the fourth joint, in degrees per second.

### Description

This function is used to simultaneously set the angular speed settings of all four joints of a mobot.

### Example

#### See Also

`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

---

## CMobot::setTwoWheelRobotSpeed()

### Synopsis

```
#include <mobot.h>
int CMobot::setTwoWheelRobotSpeed(double speed, double radius);
```

### Purpose

Roll the mobot at a certain speed in a straight line.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

<b>speed</b>	The speed at which to roll the mobot. The units used will be the units specified in the <b>unit</b> parameter.
<b>radius</b>	The radius of the wheels attached to the mobot. The units of the parameter should match the units provided in the <b>unit</b> parameter.
<b>speed</b>	<b>radius</b>

cm/s	cm
m/s	m
inch/s	inch
foot/s	foot

---

### Description

This function is used to make a two wheeled mobot roll at a certain speed. The desired speed and radius of the wheels is provided and the function will rotate the wheels at the appropriate rate in order to achieve the desired speed.

### Example

#### See Also

---

## CMobot::stopAllJoints() CMobot::stopOneJoint() CMobot::stopTwoJoints() CMobot::stopThreeJoints()

### Synopsis

```
#include <mobot.h>
int CMobot::stopAllJoints();
int CMobot::stopOneJoint(mobotJointId_t id);
```

```
int CMobot::stopTwoJoints(mobotJointId_t id1, mobotJointId_t id2);
int CMobot::stopThreeJoints(
    mobotJointId_t id1,
    mobotJointId_t id2,
    mobotJointId_t id3);
```

#### Purpose

These functions stop joints on a mobot.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Description

These functions are used to stop joints on a mobot. A stopped joint will immediately cease any and all actuation and go limp.

#### Example

#### See Also

`setJointSpeed()`, `setJointSpeeds()`

## B CMobotGroup API

The `CMobotGroup` class is used to control multiple modules simultaneously. The member functions of the `CMobotGroup` class closely mimic those of the `CMobot` group. The main difference is that the member functions of the `CMobot` class affect a single mobot, whereas the member functions of the `CMobotGroup` class move and affect a group of many mobots.

Table 3: CMobotGroup Member Functions.

Function	Description
<code>CMobotGroup()</code>	The CMobotGroup constructor function. This function is called automatically and should not be called explicitly.
<code>~CMobotGroup()</code>	The CMobotGroup destructor function. This function is called automatically and should not be called explicitly.
<code>addRobot()</code>	Add a mobot to be a member of the mobot group.
<code>move()</code>	Move all four joints of the mobots by specified angles.
<code>moveNB()</code>	Identical to <code>move()</code> but non-blocking.
<code>moveContinuousNB()</code>	Move joints continuously. Joints will move until stopped.
<code>moveContinuousTime()</code>	Move joints continuously for a certain amount of time.
<code>moveJointContinuousNB()</code>	Move a single joint on all mobots continuously.
<code>moveJointContinuousTime()</code>	Move a single joint on all mobots continuously for a specific amount of time.
<code>moveJoint()</code>	Move a motor from its current position by an angle.
<code>moveJointNB()</code>	Identical to <code>moveJoint()</code> but non-blocking.
<code>moveJointTo()</code>	Set the desired motor position for a joint.
<code>moveJointToNB()</code>	Identical to <code>moveJointTo()</code> but non-blocking.
<code>moveJointWait()</code>	Wait until the specified motor has stopped moving.
<code>moveTo()</code>	Move all four joints of the mobots to specified absolute angles.
<code>moveToNB()</code>	Identical to <code>moveTo()</code> but non-blocking.
<code>moveToZero()</code>	Instructs all motors to go to their zero positions.
<code>moveToZeroNB()</code>	Identical to <code>moveToZero()</code> but non-blocking.
<code>moveWait()</code>	Wait until all motors have stopped moving.
<code>setJointSpeed()</code>	Set a motor's speed setting in radians per second.
<code>setJointSpeeds()</code>	Set all motor speeds in radians per second.
<code>setJointSpeedRatio()</code>	Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1.
<code>setJointSpeedRatios()</code>	Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1.
<code>setTwoWheelRobotSpeed()</code>	Move the mobots at a constant forward velocity.
<code>stop()</code>	Stop all currently executing motions of the mobot.

Table 4: CMobotGroup Member Functions for Compound Motions.

Compound Motions	These are convenience functions of commonly used compound motions.
<code>motionArch()</code>	Move each mobot in the group into an arched configuration.
<code>motionArchNB()</code>	Identical to <code>motionArch()</code> but non-blocking.
<code>motionInchwormLeft()</code>	Inchworm motion towards the left.
<code>motionInchwormLeftNB()</code>	Identical to <code>motionInchwormLeft()</code> but non-blocking.
<code>motionInchwormRight()</code>	Inchworm motion towards the right.
<code>motionInchwormRightNB()</code>	Identical to <code>motionInchwormRight()</code> but non-blocking.
<code>motionRollBackward()</code>	Roll on the faceplates toward the backward direction.
<code>motionRollBackwardNB()</code>	Identical to <code>motionRollBackward()</code> but non-blocking.
<code>motionRollForward()</code>	Roll on the faceplates forwards.
<code>motionRollForwardNB()</code>	Identical to <code>motionRollForward()</code> but non-blocking.
<code>motionSkinny()</code>	Move the mobots into a skinny configuration.
<code>motionSkinnyNB()</code>	Identical to <code>motionSkinnyNB()</code> but non-blocking.
<code>motionStand()</code>	Stand the mobots up on its end.
<code>motionStandNB()</code>	Identical to <code>motionStandNB()</code> but non-blocking.
<code>motionTumbleLeft()</code>	Tumble the mobots end over end.
<code>motionTumbleLeftNB()</code>	Identical to <code>motionTumbleLeftNB()</code> but non-blocking.
<code>motionTumbleRight()</code>	Tumble the mobots end over end.
<code>motionTumbleRightNB()</code>	Identical to <code>motionTumbleRightNB()</code> but non-blocking.
<code>motionTurnLeft()</code>	Rotate the mobots counterclockwise.
<code>motionTurnLeftNB()</code>	Identical to <code>motionTurnLeft()</code> but non-blocking.
<code>motionTurnRight()</code>	Rotate the mobots clockwise.
<code>motionTurnRightNB()</code>	Identical to <code>motionTurnRight()</code> but non-blocking.
<code>motionUnstand()</code>	Move each mobot in the group currently standing on its end down into zero position.
<code>motionUnstandNB()</code>	Identical to <code>motionUnstand()</code> but non-blocking.
<code>motionWait()</code>	Wait for preprogrammed mobotic motions to complete.

---

## **CMobotGroup::addRobot()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::addRobot(CMobot &mobot);
```

**Purpose**

Add a mobot to a mobot group.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

A mobot handle attached to the mobot to add to the group.

**Description**

This function is used to add a mobot to a mobot group.

**Example****See Also**

---

## **CMobotGroup::motionArch()**

## **CMobotGroup::motionArchNB()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionArch(double angle);
int CMobotGroup::motionArchNB(double angle);
```

**Purpose**

Arch the mobots in the group for more ground clearance.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**angle** The angle which to arch. This number can range from 0 degrees, which is no arch, to 180 degrees, which is a fully curled up position.

**Description**

### **CMobot::motionArch()**

This function causes the mobots to Arch up for better ground clearance while rolling.

### **CMobot::motionArch()**

This function causes the mobots to Arch up for better ground clearance while rolling.

This function has both a blocking and non-blocking version. The blocking version, `motionArch()`, will block until the mobot motion has completed. The non-blocking version, `motionArchNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobotGroup::motionInchwormLeft()**  
**CMobotGroup::motionInchwormLeftNB()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionInchwormLeft(int num);
int CMobotGroup::motionInchwormLeftNB(int num);
```

**Purpose**

Make all mobots in the group perform the inch-worm gait to the left.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**num**        The number of times to perform the inchworm gait.

**Description**

**CMobot::motionInchwormLeft()**

This function causes the mobots to perform a single cycle of the inchworm gait to the left.

**CMobot::motionInchwormLeftNB()**

This function causes the mobots to perform a single cycle of the inchworm gait to the left.

The function `motionInchwormLeft()` is blocking, and the function will hang until the motion has finished. The alternative function, `motionInchwormLeftNB()` will return immediately, and the motion will execute asynchronously.

**See Also**

`motionInchwormRight()`

---

**CMobotGroup::motionInchwormRight()**  
**CMobotGroup::motionInchwormRightNB()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionInchwormRight(int num);
int CMobotGroup::motionInchwormRightNB(int num);
```

**Purpose**

Make all the mobots in the group perform the inch-worm gait to the right.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**num** The number of times to perform the inchworm gait.

### Description

#### **CMobot::motionInchwormRight()**

This function causes the mobots to perform a single cycle of the inchworm gait to the right.

#### **CMobot::motionInchwormRightNB()**

This function causes the mobots to perform a single cycle of the inchworm gait to the right.

This function has both a blocking and non-blocking version. The blocking version, `motionInchwormRight()`, will block until the mobot motion has completed. The non-blocking version, `motionInchwormRightNB()`, will return immediately, and the motion will be performed asynchronously.

### See Also

`motionInchwormLeft()`

---

## **CMobotGroup::motionRollBackward()**

## **CMobotGroup::motionRollBackwardNB()**

### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionRollBackward(double angle);
int CMobotGroup::motionRollBackwardNB(double angle);
```

### Purpose

Use the faceplates as wheels to roll all the mobots in a group backward.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

**angle** The angle to turn the wheels, specified in degrees.

### Description

#### **CMobot::motionRollBackward()**

This function causes each of the faceplates to rotate 90 degrees to roll the mobots backward.

#### **CMobot::motionRollBackwardNB()**

This function causes each of the faceplates to rotate 90 degrees to roll the mobots backward.

This function has both a blocking and non-blocking version. The blocking version, `motionRollBackward()`, will block until the mobot motion has completed. The non-blocking version, `motionRollBackwardNB()`, will return immediately, and the motion will be performed asynchronously.

### See Also

`motionRollForward()`

---

## **CMobotGroup::motionRollForward()**

## **CMobotGroup::motionRollForwardNB()**

### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionRollForward(double angle);
int CMobotGroup::motionRollForwardNB(double angle);
```

#### Purpose

Use the faceplates as wheels to roll mobots forward.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

**angle** The angle to turn the wheels, specified in degrees.

#### Description

##### **CMobot::motionRollForward()**

This function causes each of the faceplates to rotate 90 degrees to roll the mobots forward.

##### **CMobot::motionRollForwardNB()**

This function causes each of the faceplates to rotate 90 degrees to roll the mobots forward.

This function has both a blocking and non-blocking version. The blocking version, `motionRollForward()`, will block until the mobot motion has completed. The non-blocking version, `motionRollForwardNB()`, will return immediately, and the motion will be performed asynchronously.

#### See Also

`motionRollBackward()`

---

## **CMobotGroup::motionSkinny()**

### **CMobotGroup::motionSkinnyNB()**

#### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionSkinny(double angle);
int CMobotGroup::motionSkinnyNB(double angle);
```

#### Purpose

Move the mobots in the group into a skinny profile.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

**angle** The angle in degrees to move the joints. A value of zero means a completely flat profile, while a value of 90 degrees means a fully skinny profile.

#### Description

##### **CMobot::motionSkinny()**

This function makes the mobots assume a skinny rolling profile.

##### **CMobot::motionSkinnyNB()**

This function makes the mobots assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the mobot motion has completed. The non-blocking version, `motionSkinnyNB()`, will return immediately, and the motion will be performed asynchronously.

#### See Also

---

`CMobotGroup::motionStand()`  
`CMobotGroup::motionStandNB()`

#### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionStand();
int CMobotGroup::motionStandNB();
```

#### Purpose

Stand mobots up on a faceplate.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

None.

#### Description

##### `CMobot::motionStand()`

This function causes the mobots to stand up into the camera platform.

##### `CMobot::motionStandNB()`

This function causes the mobots to stand up into the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionStand()`, will block until the mobot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

#### See Also

---

`CMobotGroup::motionTumbleLeft()`  
`CMobotGroup::motionTumbleLeftNB()`

#### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionTumbleLeft(int num);
int CMobotGroup::motionTumbleLeftNB(int num);
```

#### Purpose

Make the mobots in the group tumble end over end.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**num** The number of times to tumble.

### **Description**

#### **CMobot::motionTumbleLeft()**

This causes the mobot to tumble end over end. The argument, **num**, indicates the number of times to tumble.

#### **CMobot::motionTumbleLeftNB()**

This causes the mobot to tumble end over end. The argument, **num**, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, **motionTumbleLeft()**, will block until the mobot motion has completed. The non-blocking version, **motionTumbleLeftNB()**, will return immediately, and the motion will be performed asynchronously.

### **See Also**

---

#### **CMobotGroup::motionTumbleRight()**

#### **CMobotGroup::motionTumbleRightNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionTumbleRight(int num);
int CMobotGroup::motionTumbleRightNB(int num);
```

### **Purpose**

Make the mobots in the group tumble end over end.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**num** The number of times to tumble.

### **Description**

#### **CMobot::motionTumbleRight()**

This causes the mobot to tumble end over end. The argument, **num**, indicates the number of times to tumble.

#### **CMobot::motionTumbleRightNB()**

This causes the mobot to tumble end over end. The argument, **num**, indicates the number of times to tumble.

This function has both a blocking and non-blocking version. The blocking version, **motionTumbleRight()**, will block until the mobot motion has completed. The non-blocking version, **motionTumbleRightNB()**, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

**CMobotGroup::motionTurnLeft()**  
**CMobotGroup::motionTurnLeftNB()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionTurnLeft(double angle);
int CMobotGroup::motionTurnLeftNB(double angle);
```

**Purpose**

Rotate the mobots using the faceplates as wheels.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**angle** The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the left.

**Description**

This function causes the mobots to rotate the faceplates in opposite directions to cause the mobot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnLeft()`, will block until the mobot motion has completed. The non-blocking version, `motionTurnLeftNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

`motionTurnRight()`

---

**CMobotGroup::motionTurnRight()**  
**CMobotGroup::motionTurnRightNB()**

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionTurnRight(double angle);
int CMobotGroup::motionTurnRightNB(double angle);
```

**Purpose**

Rotate the mobots using the faceplates as wheels.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

**angle** The angle in degrees to turn the wheels. The wheels will turn in opposite directions by the amount specified by this argument in order to turn the mobot to the right.

### Description

This function causes the mobots to rotate the faceplates in opposite directions to cause the mobot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnRight()`, will block until the mobot motion has completed. The non-blocking version, `motionTurnRightNB()`, will return immediately, and the motion will be performed asynchronously.

### See Also

`motionTurnLeft()`

---

## **CMobotGroup::motionUnstand()**

### **CMobotGroup::motionUnstandNB()**

### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionUnstand();
int CMobotGroup::motionUnstandNB();
```

### Purpose

Move mobots currently standing on a faceplate back down into a prone position.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

None.

### Description

#### **CMobot::motionUnstand()**

This function causes the mobot to move down from the camera platform.

#### **CMobot::motionUnstandNB()**

This function causes the mobot to move down from the camera platform.

This function has both a blocking and non-blocking version. The blocking version, `motionUnstand()`, will block until the mobot motion has completed. The non-blocking version, `motionUnstandNB()`, will return immediately, and the motion will be performed asynchronously.

### See Also

---

## **CMobotGroup::motionWait()**

### Synopsis

```
#include <mobot.h>
int CMobotGroup::motionWait();
```

### Purpose

Wait for a preprogrammed mobotic motion to finish.

## **Return Value**

The function returns 0 on success and non-zero otherwise.

## **Description**

This function is used to wait for a preprogrammed motion to finish. Functions such as `motionInchwormLeftNB()` and `motionRollForwardNB()` do not wait for the motion to finish moving before continuing. The `motionWait()` function is used to wait for preprogrammed motions to complete. See Section 7 for a list of all preprogrammed mobotic motions.

## **Example**

See the sample program in Section 5.1.2 on page 22.

## **See Also**

`motionWait()`, `moveJointWait()`

---

## **CMobotGroup::move()**

## **CMobotGroup::moveNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::move(double angle1, double angle2, double angle3, double angle4);
int CMobotGroup::moveNB(double angle1, double angle2, double angle3, double angle4);
```

### **Purpose**

Move all of the joints of mobots in a group by specified angles.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

- |                     |  |
|---------------------|--|
| <code>angle1</code> | The amount to move joint 1, expressed in degrees relative to the current position. |
| <code>angle2</code> | The amount to move joint 2, expressed in degrees relative to the current position. |
| <code>angle3</code> | The amount to move joint 3, expressed in degrees relative to the current position. |
| <code>angle4</code> | The amount to move joint 4, expressed in degrees relative to the current position. |

### **Description**

#### **CMobot::move()**

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

#### **CMobot::moveNB()**

This function moves all of the joints of a mobot by the specified number of degrees from their current positions.

The function `moveNB()` is the non-blocking version of the `move()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 9 on page 36.

## Example

## See Also

---

## CMobotGroup::moveContinuousNB()

### Synopsis

```
#include <mobot.h>
int CMobotGroup::moveContinuousNB(
    mobotJointState_t dir1,
    mobotJointState_t dir2,
    mobotJointState_t dir3,
    mobotJointState_t dir4);
```

### Purpose

Move the joints of grouped mobots continuously in the specified directions.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

Each integer parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

### Description

This function causes joints of mobots to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

## Example

## See Also

---

## CMobotGroup::moveContinuousTime()

### Synopsis

```
#include <mobot.h>
int CMobotGroup::moveContinuousTime(mobotJointState_t dir1,
                                      mobotJointState_t dir2,
                                      mobotJointState_t dir3,
```

```
    mobotJointState_t dir4,  
    double seconds);
```

#### Purpose

Move the joints of mobots continuously in the specified directions.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

Each of the direction parameters, `dir1`, `dir2`, `dir3`, and `dir4`, specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
<code>MOBOT_NEUTRAL</code>	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
<code>MOBOT_FORWARD</code>	This value indicates that the joint is currently moving forward.
<code>MOBOT_BACKWARD</code>	This value indicates that the joint is currently moving backward.
<code>MOBOT_HOLD</code>	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

The `seconds` parameter is the time to perform the movement, in seconds.

#### Description

This function causes joints of mobots to begin moving. The joints will continue moving until the joint hits a joint limit, or the time specified in the `seconds` parameter is reached. This function will block until the motion is completed.

#### Example

#### See Also

---

## CMobotGroup::moveJoint() CMobotGroup::moveJointNB()

#### Synopsis

```
#include <mobot.h>  
int CMobotGroup::moveJoint(mobotJointId_t id, double angle);  
int CMobotGroup::moveJointNB(mobotJointId_t id, double angle);
```

#### Purpose

Move a joint on the mobots in the group by a specified angle with respect to the current position.

#### Return Value

The function returns 0 on success and non-zero otherwise.

#### Parameters

`id` The joint number to wait for.  
`angle` The angle in degrees to move the motor, relative to the current position.

## Description

### **CMobot::moveJoint()**

This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJoint()` function.

### **CMobot::moveJointNB()**

This function commands the motor to move by an angle relative to the joint's current position at the joints current speed setting. The current motor speed may be set with the `setJointSpeed()` member function. Please note that if the motor speed is set to zero, the motor will not move after calling the `moveJointNB()` function.

The function `moveJoint()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveJointNB()` is the non-blocking version of the `moveJoint()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

## Example

Please see the example in Section 5.1.2 on page 22.

## See Also

`connectWithAddress()`

---

## **CMobotGroup::moveJointContinuousNB()**

### Synopsis

```
#include <mobot.h>
int CMobotGroup::moveJointContinuousNB(mobotJointId_t id, mobotJointState_t dir1);
```

### Purpose

Move a joint of a mobot continuously in the specified directions.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

- |                  |   |
|------------------|---|
| <code>id</code>  | The joint number to move.                                     |
| <code>dir</code> | This parameter specifies the direction the joint should move. |

### Description

The `dir` parameter specifies the direction the joint should move. The types are enumerated in `mobot.h` and have the following values:

Value	Description
<code>MOBOT_NEUTRAL</code>	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
<code>MOBOT_FORWARD</code>	This value indicates that the joint is currently moving forward.
<code>MOBOT_BACKWARD</code>	This value indicates that the joint is currently moving backward.
<code>MOBOT_HOLD</code>	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, or the joint is stopped by setting the speed to zero. This function is a non-blocking function.

### Example

### See Also

---

## CMobotGroup::moveJointContinuousTime()

### Synopsis

```
#include <mobot.h>
int CMobotGroup::moveJointContinuousTime(mobotJointId_t id, mobotJointState_t dir1, double seconds);
```

### Purpose

Move a joint of a mobot continuously in the specified directions for a certain amount of time.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Parameters

- id** The joint number to move.
- dir** This parameter specifies the direction the joint should move.
- seconds** The number of seconds to rotate the joint.

### Description

The **dir** parameter specifies the direction the joint should move. The types are enumerated in **mobot.h** and have the following values:

Value	Description
MOBOT_NEUTRAL	This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable.
MOBOT_FORWARD	This value indicates that the joint is currently moving forward.
MOBOT_BACKWARD	This value indicates that the joint is currently moving backward.
MOBOT_HOLD	This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable.

More documentation about these types may be found at Section A.3 on page 70.

This function causes joints of a mobot to begin moving at the previously set speed. The joints will continue moving until the joint hits a joint limit, the joint is stopped by setting the speed to zero, or the specified amount of time has expired.

### Example

### See Also

---

## CMobotGroup::moveJointTo()

## **CMobotGroup::moveJointToNB()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveJointTo(mobotJointId_t id, double angle);
int CMobotGroup::moveJointToNB(mobotJointId_t id, double angle);
```

### **Purpose**

Move a joint on mobots to an absolute position.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

**id**      The joint number to wait for.  
**angle**    The absolute angle in degrees to move the motor to.

### **Description**

#### **CMobot::moveJointTo()**

This function commands the motor on mobots in a group to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the **setJointSpeed()** member function. Please note that if the motor speed is set to zero, the motor will not move after calling the **moveJointTo()** function.

#### **CMobot::moveJointToNB()**

This function commands the motor on mobots in a group to move to a position specified in degrees at the current motor's speed. The current motor speed may be set with the **setJointSpeed()** member function. Please note that if the motor speed is set to zero, the motor will not move after calling the **moveJointToNB()** function.

The function **moveJointTo()** is a blocking function, which means that the function will not return until the commanded motion is completed. The function **moveJointToNB()** is the non-blocking version of the **moveJointTo()** function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

### **Example**

Please see the example in Section 5.1.2 on page 22.

### **See Also**

---

## **CMobotGroup::moveJointWait()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveJointWait(mobotJointId_t id);
```

### **Purpose**

Wait for a joint to stop moving on all mobots in a group.

## **Return Value**

The function returns 0 on success and non-zero otherwise.

## **Parameters**

**id**      The joint number to wait for.

## **Description**

This function is used to wait for a joint motion to finish. Functions such as `moveJointToNB()` and `moveJointNB()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` or `moveJointWait()` functions are used to wait for mobotic joint motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

## **Example**

Please see the example in Section 5.1.2 on page 22.

## **See Also**

`moveWait()`

---

**CMobotGroup::moveTo()**

**CMobotGroup::moveToNB()**

## **Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveTo(double angle1, double angle2, double angle3, double angle4);
int CMobotGroup::moveToNB(double angle1, double angle2, double angle3, double angle4);
```

## **Purpose**

Move all of the joints of mobots in the group to the specified positions.

## **Return Value**

The function returns 0 on success and non-zero otherwise.

## **Parameters**

**angle1**      The absolute position to move joint 1, expressed in degrees.  
**angle2**      The absolute position to move joint 2, expressed in degrees.  
**angle3**      The absolute position to move joint 3, expressed in degrees.  
**angle4**      The absolute position to move joint 4, expressed in degrees.

## **Description**

**CMobot::moveTo()**

This function moves all of the joints of mobots in the group to the specified absolute positions.

**CMobot::moveToNB()**

This function moves all of the joints of mobots in the group to the specified absolute positions.

The function `moveTo()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveToNB()` is the non-blocking version of the `moveTo()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

### **Example**

Please see the demo at Section 5.1.2 on page 22.

### **See Also**

---

## **CMobotGroup::moveToZero()**

### **CMobotGroup::moveToZeroNB()**

#### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveToZero();
int CMobotGroup::moveToZeroNB();
```

#### **Purpose**

Move all of the joints of mobots in the group to their zero position.

#### **Return Value**

The function returns 0 on success and non-zero otherwise.

#### **Parameters**

None.

#### **Description**

##### **CMobot::moveToZero()**

This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

##### **CMobot::moveToZeroNB()**

This function moves all of the joints of mobots in the group to their zero position. Please note that this function is non-blocking and will return immediately. Use this function in conjunction with the `moveWait()` function to block until the movement completes.

The function `moveToZero()` is a blocking function, which means that the function will not return until the commanded motion is completed. The function `moveToZeroNB()` is the non-blocking version of the `moveToZero()` function, which means that the function will return immediately and the physical mobot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 9 on page 36.

### **Example**

Please see the demo at Section 5.1.2 on page 22.

### **See Also**

---

## **CMobotGroup::moveWait()**

#### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveWait();
```

**Purpose**

Wait for all joints of all mobots in the group to stop moving.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Description**

This function is used to wait for all joint motions to finish. Functions such as `moveJointToNB()` and `moveJointNB()` do not wait for a joint to finish moving before continuing to allow multiple joints to move at the same time. The `moveWait()` or `moveJointWait()` functions are used to wait for mobotic motions to complete.

Please note that if this function is called after a motor has been commanded to turn indefinitely, this function may never return and your program may hang.

**Example**

See the sample program in Section 5.1.2 on page 22.

**See Also**

`moveWait()`, `moveJointWait()`

---

## CMobotGroup::setJointSpeed()

**Synopsis**

```
\vspace{-8pt}
#include <mobot.h>
int CMobotGroup::setJointSpeed(mobotJointId_t id, double speed);
```

**Purpose**

Set the speed of a joint on all mobots in the group.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

`id` The joint number to pose.  
`speed` An variable of type `double` for the requested average angular speed in degrees per second.

**Description**

This function is used to set the angular speed of a joint of all mobots in the group.

**Example****See Also**

---

## CMobotGroup::setJointSpeedRatio()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::setJointSpeedRatio(mobotJointId_t id, double ratio);
```

**Purpose**

Set the speed ratio settings of a joint on all mobots in the group.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

- id** Set the speed ratio setting of this joint. This is an enumerated type discussed in Section A.2 on page 70.
- ratio** A variable of type double with a value from 0 to 1.

### **Description**

This function is used to set the speed ratio setting of a joint for all mobots in the group. The speed ratio setting of a joint is the percentage of the maximum joint speed, and the value ranges from 0 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.

### **Example**

#### **See Also**

`setJointSpeeds()`, `setJointSpeedRatio()`

---

## **CMobotGroup::setJointSpeedRatios()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::setJointSpeedRatios(double ratio1, double ratio2, double ratio3, double ratio4);
```

### **Purpose**

Set the speed ratio settings of all joints on the mobots in the group.

### **Return Value**

The function returns 0 on success and non-zero otherwise.

### **Parameters**

- ratio1** The speed ratio setting for the first joint.
- ratio2** The speed ratio setting for the second joint.
- ratio3** The speed ratio setting for the third joint.
- ratio4** The speed ratio setting for the fourth joint.

### **Description**

This function is used to simultaneously set the angular speed ratio settings of all four joints of a mobot for all mobots in the group. The speed ratio is a percentage of the maximum speed of a joint, expressed in a value from 0 to 1.

### **Example**

#### **See Also**

`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

---

## **CMobotGroup::setJointSpeeds()**

### **Synopsis**

```
#include <mobot.h>
int CMobotGroup::setJointSpeeds(double speed1, double speed2, double speed3, double speed4);
```

**Purpose**

Set the speed settings of all joints on all mobot in the group.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

- speed1** The speed setting for the first joint, in units of degrees per second.
- speed2** The speed setting for the second joint, in units of degrees per second.
- speed3** The speed setting for the third joint, in units of degrees per second.
- speed4** The speed setting for the fourth joint, in units of degrees per second.

**Description**

This function is used to simultaneously set the angular speed settings of all four joints of all mobots in the group. The joint speeds are expressed in degrees per second.

**Example****See Also**

`setJointSpeed()`

---

## CMobotGroup::setTwoWheelRobotSpeed()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::setTwoWheelRobotSpeed(double speed, double radius);
```

**Purpose**

Roll the mobots in the group at a certain speed in a straight line.

**Return Value**

The function returns 0 on success and non-zero otherwise.

**Parameters**

- speed** The speed at which to roll the mobot. The units used will be the units specified in the **unit** parameter.
- radius** The radius of the wheels attached to the mobot. The units of the parameter should match the units provided in the **unit** parameter.
- speed      radius**

---

cm/s	cm
m/s	m
inch/s	inch
foot/s	foot

**Description**

This function is used to make a two wheeled mobot roll at a certain speed. The desired speed and radius of the wheels is provided and the function will rotate the wheels at the appropriate rate in order to achieve the desired speed.

**Example****See Also**

---

## CMobotGroup::stop()

### Synopsis

```
#include <mobot.h>
int CMobotGroup::stop();
```

### Purpose

Stop all current motions on all mobot in the group.

### Return Value

The function returns 0 on success and non-zero otherwise.

### Description

This function stops all currently occurring movements on the mobot. Internally, this function simply sets all motor speeds to zero. If it is only required to stop a single motor, use the `setJointSpeed()` function to set the motor's speed to zero.

### Example

#### See Also

`setJointSpeed()`, `setJointSpeeds()`

## C Miscellaneous Utility Functions

There are several utility functions which are useful when programming for the Mobot.

Table 5: Mobot Utility Functions.

Function	Description
<code>angle2distance()</code>	Calculates the angle a wheel has turned from the radius and distance traveled.
<code>deg2rad()</code>	Converts degrees to radians.
<code>delay()</code>	Puts a pause into a program.
<code>distance2angle()</code>	Calculates the distance traveled by a wheel from the wheel's radius and angle turned.
<code>rad2deg()</code>	Converts radians to degrees.
<code>shiftTime()</code>	Shift the data of a plot.

---

## angle2distance()

### Synopsis

```
#include <mobot.h>
double angle2distance(double radius, double angle);
array double angle2distance(double radius, array double angle[:])[:];
```

### Purpose

Calculate the distance a wheel has traveled from the radius of the wheel and the angle the wheel has turned.

### Return Value

The value returned is the distance traveled by the wheel. If the angle argument is an array of angles, then the value returned is an array of distances. Each element of the distance array returned is the distance calculated from the respective element in the angle array.

### Parameters

radius	The radius of the wheel.
angle	This value is the angle the wheel has turned. This parameter may be of <code>double</code> type, or a Ch computational array.

### Description

This function calculates the angle a wheel has turned given the wheel radius and distance traveled. The equation used is

$$d = r\theta$$

where  $d$  is the distance traveled,  $r$  is the radius of the wheel, and  $\theta$  is the angle the wheel has turned in radians.

### Example

#### See Also

`distance2angle()`

---

## deg2rad()

### Synopsis

```
#include <mobot.h>
double deg2rad(double degrees);
array double deg2rad(double degrees[:])[:];
```

### Purpose

Convert degrees to radians.

### Return Value

The angle parameter converted to radians.

### Parameters

degrees	The angle to convert, in degrees.
---------	-----------------------------------

### Description

This function converts an angle expressed in degrees into radians. Degrees and radians are two popular ways

to express an angle, though they are not interchangable. The following equation is used to convert degrees to radians:

$$\theta = \delta * \frac{\pi}{180}$$

where  $\theta$  is the angle in radians and  $\delta$  is the angle in degrees.

#### Example

#### See Also

`rad2deg()`

---

## delay()

#### Synopsis

```
#include <mobot.h>
void delay(double seconds);
```

#### Purpose

Pause a program for a set amount of time.

#### Return Value

None.

#### Parameters

`seconds` The number of seconds to delay.

#### Description

This function delays or pauses a program for a number of seconds. For instance, the code

```
delay(0.5);
printf("Hello.\n");
delay(2);
printf("Goodbye.\n");
```

will pause for half a second, print the text Hello., delay for 2 seconds, and then print the text Goodbye..

#### Example

#### See Also

---

## distance2angle()

#### Synopsis

```
#include <mobot.h>
double distance2angle(double radius, double distance);
array double distance2angle(double radius, array double distance[:])[:];
```

#### Purpose

Calculate the angle a wheel has turned from the radius of the wheel and the distance the wheel has traveled.

#### Return Value

The value returned is the angle turned by the wheel in degrees. If the distance argument is an array of distances, then the value returned is an array of angles. Each element of the angle array returned is the angle calculated from the respective element in the distance array.

#### Parameters

<b>radius</b>	The radius of the wheel.
<b>distance</b>	This value is the distance the wheel has traveled. This parameter may be of <b>double</b> type, or a Ch computational array.

### Description

This function calculates the distance a wheel has turned given the wheel radius and angle turned. The equation used is

$$\theta = \frac{d}{r}$$

where  $d$  is the distance traveled,  $r$  is the radius of the wheel, and  $\theta$  is the angle the wheel has turned in radians. A further conversion is done in the code to convert the angle from radians into degrees before returning the value.

### Example

#### See Also

`angle2distance()`

---

## processRecordedAngles()

### Synopsis

```
#include <mobot.h>
int processRecordedAngles(double time[], double anglen[], int numElements);
```

### Syntax

```
#include <mobot.h>
int processRecordedAngles(double time[], double anglen[], int numElements);
```

### Purpose

This function is used to condition the data obtained by the `recordAngle()` family of functions.

### Return Value

The return value is the number of elements which have been shifted of the plots.

### Parameters

<b>time</b>	The array holding time or "x-axis" values.
<b>data1</b>	An array holding data.
<b>numElements</b>	The number of elements in the arrays.

### Description

This function conditions recorded data by shifting the graphs to the left such that the first motion occurs at time 0.

### Example

#### See Also

---

## rad2deg()

### Synopsis

```
#include <mobot.h>
double rad2deg(double radians);
array double rad2deg(double radians[:])[:];
```

### Purpose

Convert radians to degrees.

### Return Value

The angle parameter converted to degrees.

### Parameters

**radians** The angle to convert, in radians.

### Description

This function converts an angle expressed in radians into degrees. Degrees and radians are two popular ways to express an angle, though they are not interchangable. The following equation is used to convert radians to degrees:

$$\delta = \theta * \frac{180}{\pi}$$

where  $\theta$  is the angle in radians and  $\delta$  is the angle in degrees.

### Example

#### See Also

`deg2rad()`

## shiftTime()

### Synopsis

```
#include <mobot.h>
int shiftTime(double tolerance, int numDataPoints, double time[], double data1[], ...);
```

### Syntax

```
#include <mobot.h>
shiftTime(tolerance, numDataPoints, time, angles1);
shiftTime(tolerance, numDataPoints, time, angles1, angles2);
shiftTime(tolerance, numDataPoints, time, angles1, angles2, angles3);
shiftTime(tolerance, numDataPoints, time, angles1, angles2, angles3, angles4);
etc...
```

### Purpose

This function is used to shift the data in one or more plots to the left. It is commonly used to line up the beginning of mobot motions with the y-axis on plots.

### Return Value

The return value is the number of elements which have been shifted of the plots.

### Parameters

**tolerance**

The angle tolerance to detect the beginning of the motion. A lower tolerance is more sensitive to small motions, but also more sensitive to noise. A higher tolerance will reject noise, but may yield an inaccurate shift in time such that the motion does not appear to begin at time 0.

**numDataPoints**

The number of elements in the arrays.

**time**

The array holding time or "x-axis" values.

**data1**

An array holding data.

**...**

Additional arrays holding data.

### **Description**

This function is used to shift data to the left to align motion start times with the y-axis. This is done by detecting a change in value in any of the data arrays provided to the function. If there is a change greater than the value provided as the tolerance, that time is labeled as the beginning of the motion. All data points prior to the beginning of the motion are deleted, and the beginning of the motion is aligned with time 0.

### **Example**

### **See Also**

# Index

angle2distance(), 135  
CMobot::blinkLED(), 74  
CMobot::connect(), 74  
CMobot::connectWithAddress(), 75  
CMobot::disconnect(), 75  
CMobot::driveJointToDirect(), 76  
CMobot::driveJointToDirectNB(), 76  
CMobot::driveToDirect(), 76  
CMobot::driveToDirectNB(), 76  
CMobot::getJointAngle(), 77  
CMobot::getJointAngleAbs(), 77  
CMobot::getJointMaxSpeed(), 78  
CMobot::getJointSafetyAngle(), 78  
CMobot::getJointSafetyAngleTimeout(), 79  
CMobot::getJointSpeed(), 79  
CMobot::getJointSpeedRatio(), 80  
CMobot::getJointSpeedRatios(), 80  
CMobot::getJointSpeeds(), 81  
CMobot::getJointState(), 81  
CMobot::isConnected(), 82  
CMobot::isMoving(), 82  
CMobot::motionArch(), 83  
CMobot::motionArchNB(), 83  
CMobot::motionInchwormLeft(), 84  
CMobot::motionInchwormRight(), 84  
CMobot::motionInchwormRightNB(), 84  
CMobot::motionRollBackward(), 85  
CMobot::motionRollBackwardNB(), 85  
CMobot::motionRollForward(), 85  
CMobot::motionRollForwardNB(), 85  
CMobot::motionSkinny(), 86  
CMobot::motionSkinnyNB(), 86  
CMobot::motionStand(), 87  
CMobot::motionStandNB(), 87  
CMobot::motionTumbleLeft(), 87  
CMobot::motionTumbleLeftNB(), 87  
CMobot::motionTumbleRight(), 88  
CMobot::motionTumbleRightNB(), 88  
CMobot::motionTurnLeft(), 89  
CMobot::motionTurnLeftNB(), 89  
CMobot::motionTurnRight(), 89  
CMobot::motionTurnRightNB(), 89  
CMobot::motionUnstand(), 90  
CMobot::motionUnstandNB(), 90  
CMobot::motionWait(), 91  
CMobot::move(), 91  
CMobot::moveContinuousNB(), 92  
CMobot::moveContinuousTime(), 93  
CMobot::moveJoint(), 93  
CMobot::moveJointContinuousNB(), 94  
CMobot::moveJointContinuousTime(), 95  
CMobot::moveJointNB(), 93  
CMobot::moveJointTo(), 96  
CMobot::moveJointToAbs(), 96  
CMobot::moveJointToAbsNB(), 97  
CMobot::moveJointToDirect(), 98  
CMobot::moveJointToDirectNB(), 98  
CMobot::moveJointToNB(), 96  
CMobot::moveJointWait(), 99  
CMobot::moveNB(), 91  
CMobot::moveTo(), 99  
CMobot::moveToAbs(), 100  
CMobot::moveToAbsNB(), 100  
CMobot::moveToDirect(), 101  
CMobot::moveToDirectNB(), 101  
CMobot::moveToNB(), 99  
CMobot::moveToZero(), 102  
CMobot::moveToZeroNB(), 102  
CMobot::moveWait(), 103  
CMobot::recordAngle(), 103  
CMobot::recordAngleBegin(), 104  
CMobot::recordAngleEnd(), 104  
CMobot::recordAngles(), 105  
CMobot::recordAnglesBegin(), 106  
CMobot::recordAnglesEnd(), 107  
CMobot::recordWait(), 107  
CMobot::setJointSafetyAngle(), 108  
CMobot::setJointSafetyAngleTimeout(), 108  
CMobot::setJointSpeed(), 109  
CMobot::setJointSpeedRatio(), 109  
CMobot::setJointSpeedRatios(), 110  
CMobot::setJointSpeeds(), 110  
CMobot::setTwoWheelRobotSpeed(), 111  
CMobot::stopAllJoints(), 111  
CMobot::stopOneJoint(), 111  
CMobot::stopThreeJoints(), 111  
CMobot::stopTwoJoints(), 111  
CMobotGroup::addRobot(), 115  
CMobotGroup::motionArch(), 115  
CMobotGroup::motionArchNB(), 115  
CMobotGroup::motionInchwormLeft(), 116  
CMobotGroup::motionInchwormLeftNB(), 116  
CMobotGroup::motionInchwormRight(), 116  
CMobotGroup::motionInchwormRightNB(), 116  
CMobotGroup::motionRollBackward(), 117

CMobotGroup::motionRollBackwardNB(), 117  
CMobotGroup::motionRollForward(), 117  
CMobotGroup::motionRollForwardNB(), 117  
CMobotGroup::motionSkinny(), 118  
CMobotGroup::motionSkinnyNB(), 118  
CMobotGroup::motionStand(), 119  
CMobotGroup::motionStandNB(), 119  
CMobotGroup::motionTumbleLeft(), 119  
CMobotGroup::motionTumbleLeftNB(), 119  
CMobotGroup::motionTumbleRight(), 120  
CMobotGroup::motionTumbleRightNB(), 120  
CMobotGroup::motionTurnLeft(), 121  
CMobotGroup::motionTurnLeftNB(), 121  
CMobotGroup::motionTurnRight(), 121  
CMobotGroup::motionTurnRightNB(), 121  
CMobotGroup::motionUnstand(), 122  
CMobotGroup::motionUnstandNB(), 122  
CMobotGroup::motionWait(), 122  
CMobotGroup::move(), 123  
CMobotGroup::moveContinuousNB(), 124  
CMobotGroup::moveContinuousTime(), 124  
CMobotGroup::moveJoint(), 125  
CMobotGroup::moveJointContinuousNB(), 126  
CMobotGroup::moveJointContinuousTime(), 127  
CMobotGroup::moveJointNB(), 125  
CMobotGroup::moveJointTo(), 127  
CMobotGroup::moveJointToNB(), 128  
CMobotGroup::moveJointWait(), 128  
CMobotGroup::moveNB(), 123  
CMobotGroup::moveTo(), 129  
CMobotGroup::moveToNB(), 129  
CMobotGroup::moveToZero(), 130  
CMobotGroup::moveToZeroNB(), 130  
CMobotGroup::moveWait(), 130  
CMobotGroup::setJointSpeed(), 131  
CMobotGroup::setJointSpeedRatio(), 131  
CMobotGroup::setJointSpeedRatios(), 132  
CMobotGroup::setJointSpeeds(), 132  
CMobotGroup::setTwoWheelRobotSpeed(), 133  
CMobotGroup::stop(), 134  
copyright, 2  
  
deg2rad(), 135  
delay(), 136  
distance2angle(), 136  
  
MOBOT\_JOINT1, 70  
MOBOT\_JOINT2, 70  
MOBOT\_JOINT3, 70  
MOBOT\_JOINT4, 70  
mobot\_joints\_t, 70