Barobo™

# Mobot User's Guide

**Version 1.3.6**

## How to Contact Barobo


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



Revision 1.3.6, February 2012

# Contents

# 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 als obe 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 Mobots.

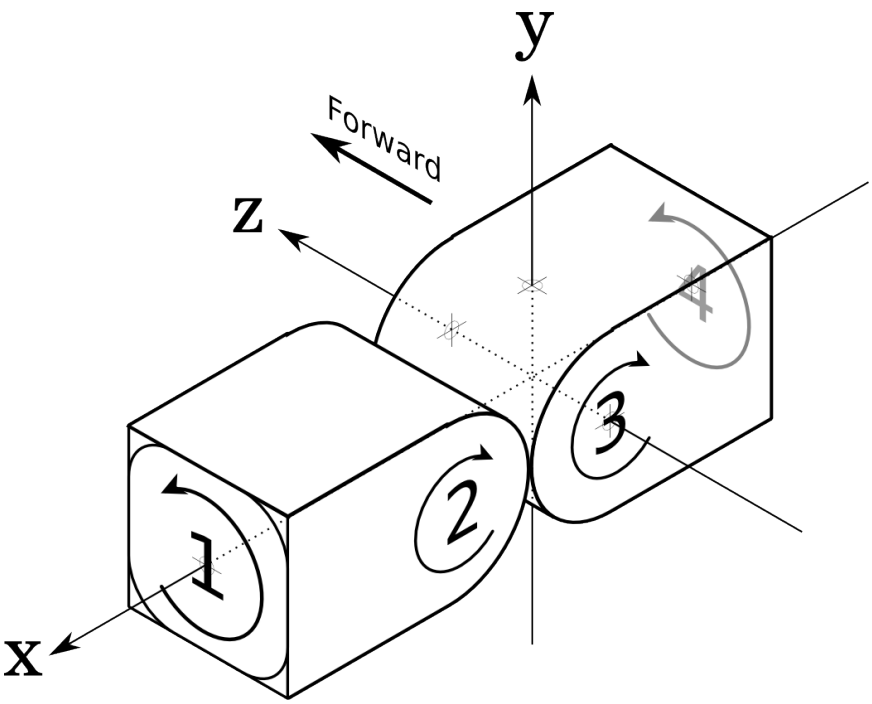



Figure 1: A schematic diagram of a Mobot module.

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 Mobots for Remote Control

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

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

To start the provided Barobo Robot Control Program click on the icon labeled "RobotController" on your desktop, as shown in 2. The control dialog as shown in Figure 3 should pop up.

Figure 2: The icon for the Barobo RobotController.

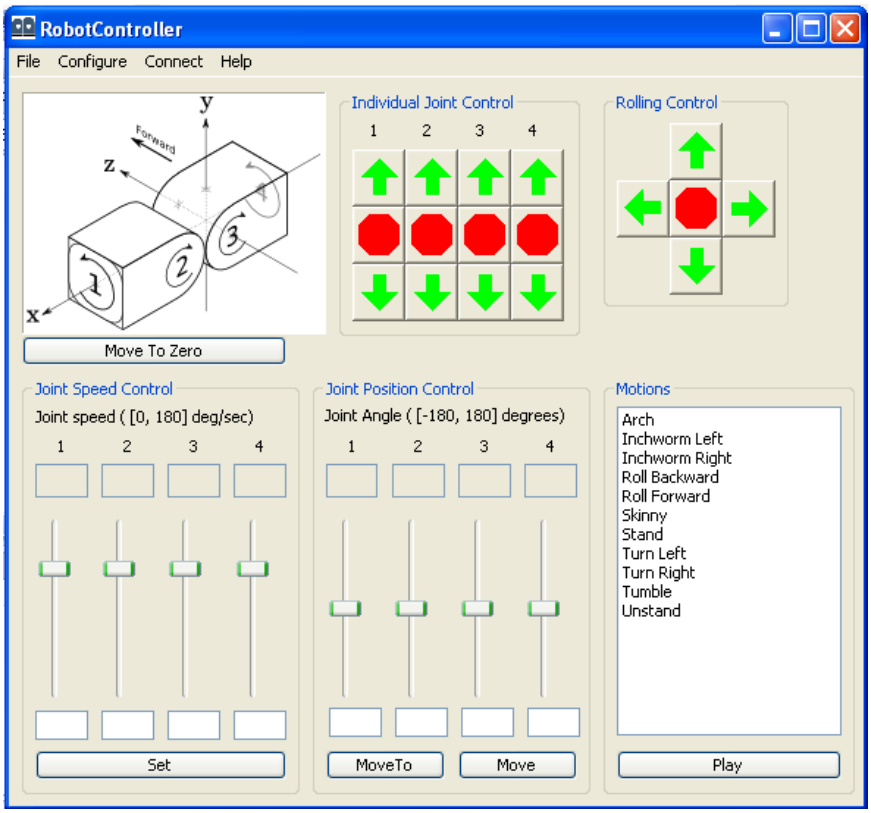



Figure 3: The graphical user interface of the RobotController.

## 2.1 Adding Bluetooth Addresses of Robots in RobotController.

Click on the menu item "Configure → Configure Robot Bluetooth", as shown in Figure 4.

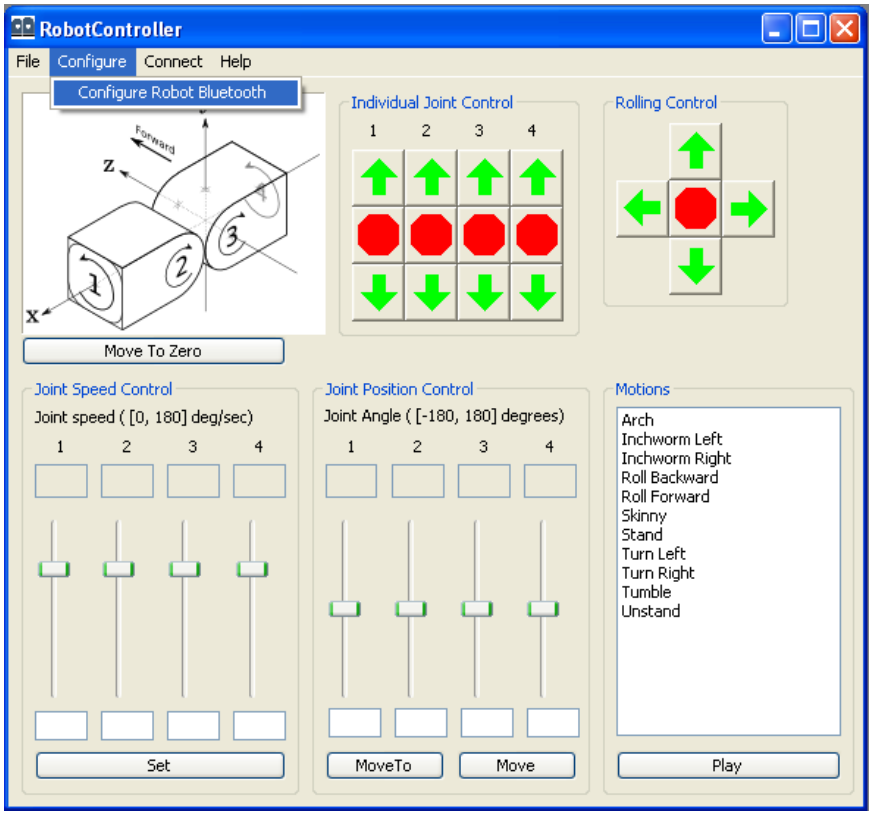



Figure 4: Configuring robot bluetooth connection.

This should bring up a second dialog, titled “Configure Robot Bluetooth”, as shown in Figure 5.

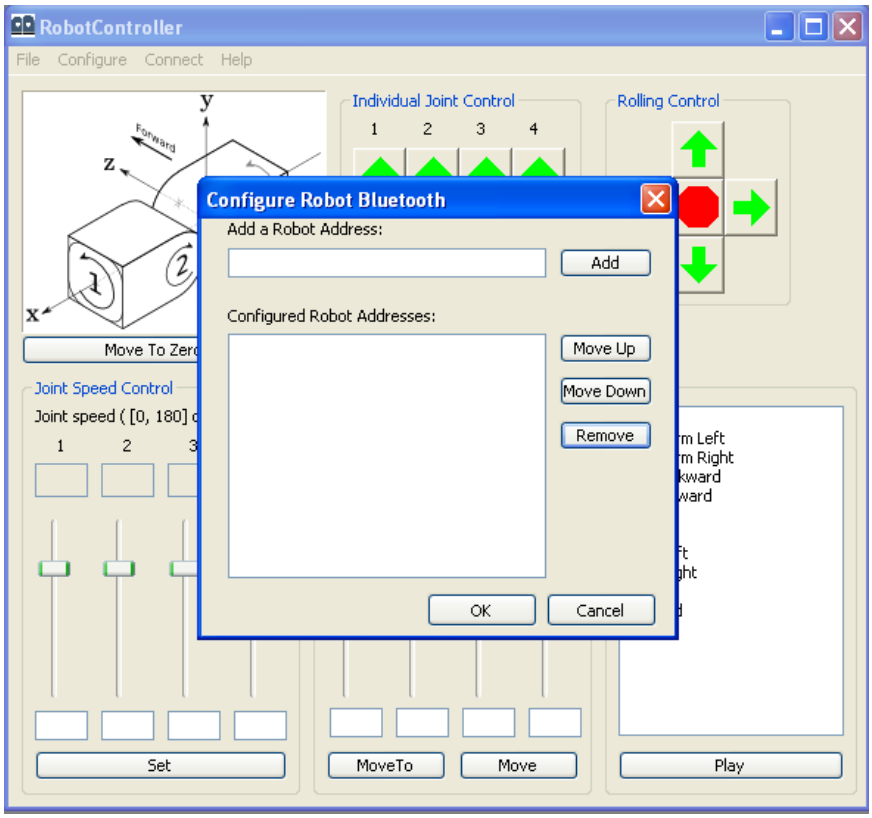



Figure 5: The dialog window for bluetooth connection.

This dialog allows us to add robot bluetooth addresses to the list of currently known robot bluetooth addresses. To add an address, first type in the address in the text box on the top of the dialog, as shown in Figure 6. You can find the bluetooth address of each robot inside the battery compartment of the robot on the same side as the power switch.

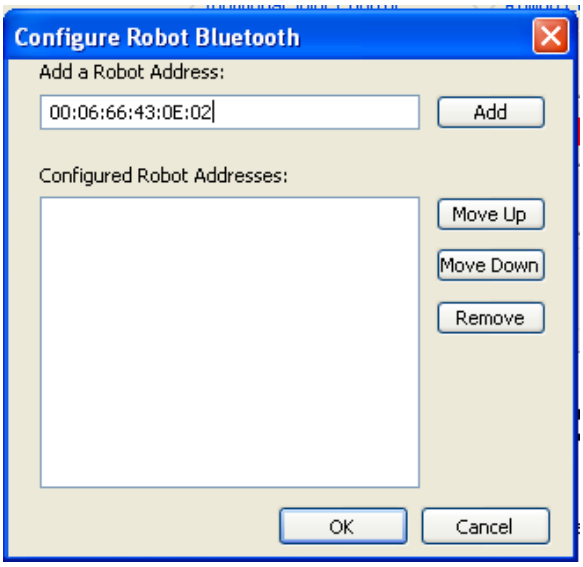



Figure 6: Adding the robot bluetooth address in the dialog window.

Next, click the "Add" button. The newly added address should appear in the list of known addresses, as shown in Figure 7. In our case, we have added the address of one of our robots, which is `"00:06:66:43:0E:02"`.

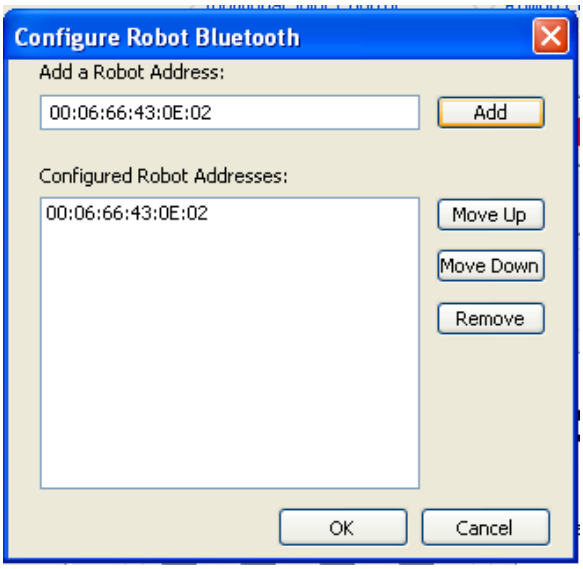



Figure 7: Displaying the added bluetooth address.

We use the same process to add our remaining two robots to the list, with addresses `"00:06:66:43:0D:F2"` and `"00:06:66:47:23:9C"`. The dialog now appears as shown in Figure 8.

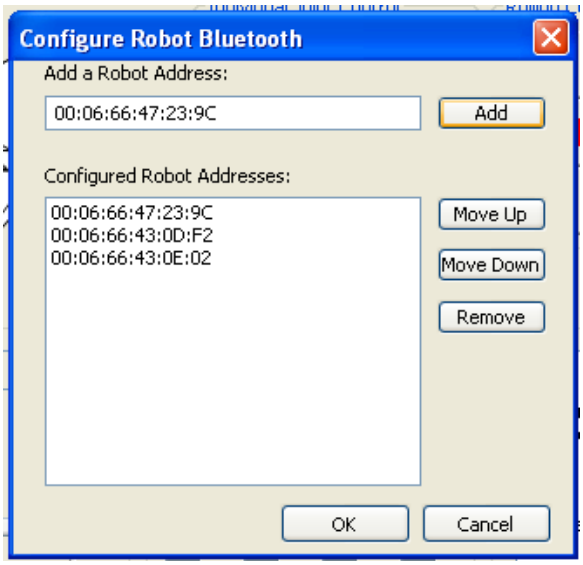



Figure 8: Displaying bluetooth addresses for three robots.

The dialog also allows users to reorder the addresses listed. The order the addresses are listed in affects the order in which the robots are connected to using the `connect()` member function. The remote control dialog connects to the primary address located at the top of the list by default. To reorder the list of addresses, simply select the address to move and click on the "Move Up" or "Move Down" button to either move the address higher in the list or lower. For instance, the result of clicking on the address `"00:06:66:43:0D:F2"` and clicking the "Move Up" button is shown in Figure 9.

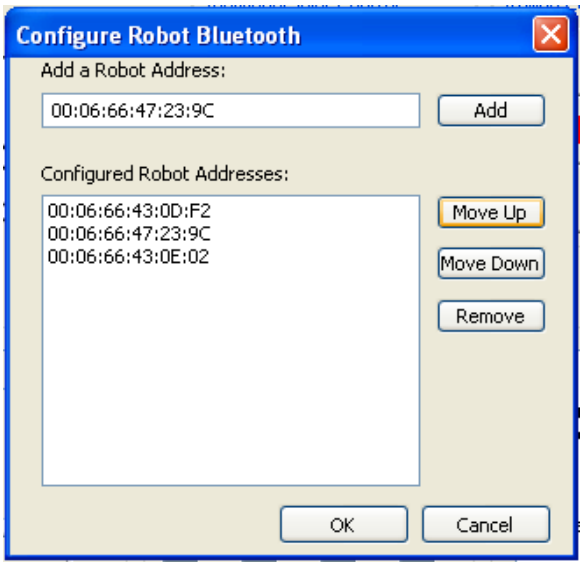



Figure 9: Modifying the order of bluetooth addresses with the "Configure Robot Bluetooth" dialog.

## 2.2 Connecting and Disconnecting to Robots from the RobotController

Once bluetooth addresses are added to the RobotController, you may connect to the first address by clicking on the "Connect → Connect to Robot" menu item. The connected robot may be disconnected by clicking on the "Connect → Disconnect from Robot" menu item. Any connected robots are automatically disconnected upon exiting the program. Note that in order to run a Ch program that controls a robot, the robot should not currently be connected to any other application, including the RobotController, other Ch programs, and other programs on other computers.

# 3 The Robot Remote Control Program

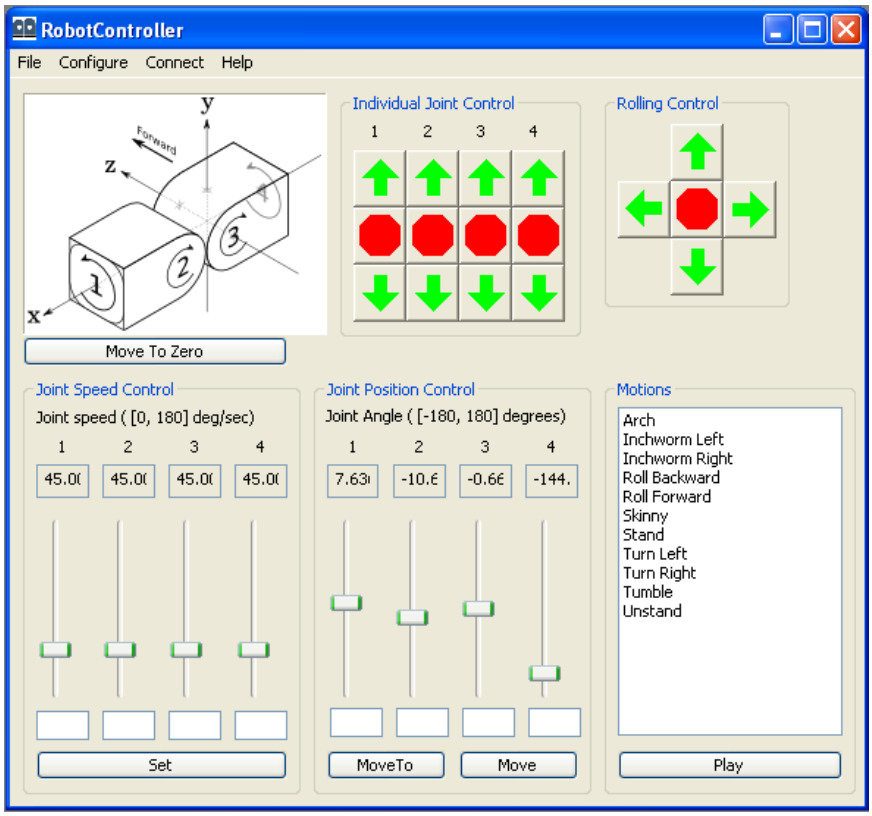



Figure 10: The graphical display of the RobotController while connected to a robot.

Once a robot is connected to the RobotController, the joint angles and speeds of the robot are displayed as shown in Figure 10. The RobotController can then 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.

## 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 robot. 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".

## 3.7 Application Example for Learning Algebra

### 3.7.1 Problem Statement

A robot has two 3.5 inch diameter wheels. It is moving at a constant velocity by turning its wheels 45 degrees per second.

1. How long will it take for the robot to rotate its wheels 2 full rotations? (720 degrees)
2. During that time, how far should the robot travel? Verify your results by measuring the distance the real robot travels during the calculated time.

### 3.7.2 Problem Solution

1. We are given that a robot is turning its wheels at a constant speed of 45 degrees per second, and needs to rotate the wheels 720 degrees. The amount of time this will take is

$$\frac{720 \text{ degrees}}{45 \text{ degrees/second}} = 16 \text{ seconds}$$

2. The circumference of a circle can be determined with the equation

$$c = 2\pi r$$

where $r$ is the radius of the circle. Since our robot will turn its wheels two full rotations during the motion, it will travel 2 circumferences in distance. Thus, the distance the robot should travel is

$$d = 2 \times c = 2 \times (2\pi r) = 4\pi r$$

The radius of a circle is half of its diameter, which is $3.5/2 = 1.75$ inches, in our case. Plugging in our numbers, we have

$$d = 4 \times \pi \times 1.75 = 21.99$$

Our robot should travel approximately 22 inches during the 16 second motion.

To verify these results with a real robot the RobotController program may be used.

1. Affix two 3.5 inch diameter wheels to the faceplates.
2. Set the joint speeds for joints 1 and 4 to 45 degrees per second. This may be done by typing "45" into the appropriate text boxes for joint 1 and 4 speeds at the bottom left of the RobotController and then clicking the "Set" button.
3. Type "720" into the text edit boxes below joints 1 and 4 in the joint position section of the GUI.
4. Click the "Move" button. Joints 1 and 4 will turn 720 degrees, rolling the robot forward. The motion may be timed with a stopwatch.

# 4 Getting Started with Programming the Mobot

The RobotController 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 the RobotController as described in Section 2. If a robot is already connected to the RobotController, disconnect from the robot before running the Ch program, or close the RobotController application.

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.

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

## 4.1 `start.ch`, A Basic Ch Mobot Program

### 4.1.1 `start.ch` Source Code

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

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

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

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

### 4.1.2 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 robotic control functions.

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

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

```
CMobot robot;
```

This line initializes a new variable named `robot` 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,

```
robot.connect();
```

will connect our new variable, `robot`, 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:

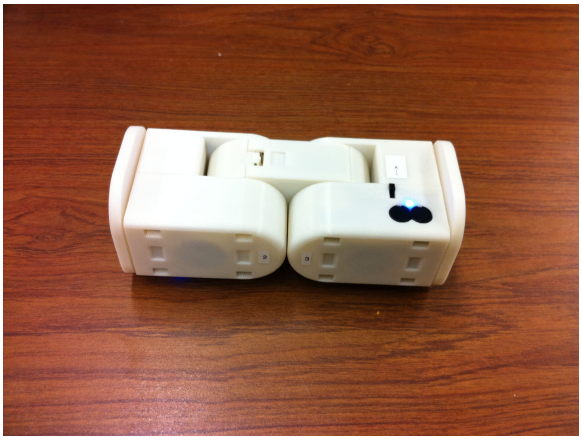


Figure 11: The robot in zero position

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

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 robot. Detailed documentation for each of the Mobot functions, such as `connect()` and `connectAddress()`, are presented in Appendix B.

The next line,

```
robot.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 11.

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

```
robot.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".

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

### 4.2.1 Source Code

```
/* Filename: returnval.ch
```

```
 * Rotate the faceplates by 90 degrees */
#include <mobot.h>
CMobot robot;
double angle1, angle4;

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

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

### 4.2.2 `returnval.ch` Explained

The first portion of the code, the lines

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

set up our program for controlling robots 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 robot, appear as such:

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

This section connects to the remote robot 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 robot to "home" position, where all joint angles are 0 degrees. */
robot.moveToZero();

/* Rotate each of the faceplates by 360 degrees */
```

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

move the robot 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.

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

### 4.3.1 Source Code

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

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

/* Get the joint angle of the first joint */
double angle;
robot.getJointAngle(ROBOT_JOINT1, angle);

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

### 4.3.2 `getJointAngle.ch` Explained

The first portion of the program,

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

/* Connect to a robot */
robot.connect();
```

initialize the robot variable and connect to the remote robot, 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 robotic joints. The next line,

```
robot.getJointAngle(ROBOT_JOINT1, angle);
```

retrieves the current angle of joint 1, which is one of the faceplates of the robot. `ROBOT_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.

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.

# 5 Controlling the Speed of Mobot Joints

## 5.1 setspeed.ch Source Code

```
/* Filename: setspeed.ch
  Move the two wheeled robot with different speed. */
#include <mobot.h>
#include <math.h>
CMobot robot;

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

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

double speed, radius;
robot.getJointMaxSpeed(ROBOT_JOINT1, speed);
printf("The maximum speed is %lf degrees/s\n", speed);

robot.setJointSpeed(ROBOT_JOINT1, 90);
robot.setJointSpeed(ROBOT_JOINT4, 90);

//robot.setJointSpeedRatio(ROBOT_JOINT1, 0.5);
//robot.setJointSpeedRatio(ROBOT_JOINT4, 0.5);

//robot.setJointSpeeds(90, 0, 0, 90);

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

speed = (3.5/2) * M_PI / 2; // 2.75 inch/s
radius = 3.5/2;             // radius is 1.75
robot.setTwoWheelRobotSpeed(speed, radius);

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

/* move at 2.75inch/sec with the radius 3.5 inches for 3 seconds */
printf("Move continuously for 3 seconds.\n");
robot.moveContinuousTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD, 3);
```

## 5.2 setspeed.ch Source Code Explanation

The first several lines,

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

```
CMobot robot;

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

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

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

```
double speed, radius;
robot.getJointMaxSpeed(ROBOT_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,

```
robot.setJointSpeed(ROBOT_JOINT1, 90);
robot.setJointSpeed(ROBOT_JOINT4, 90);
```

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

The next lines,

```
//robot.setJointSpeedRatio(ROBOT_JOINT1, 0.5);
//robot.setJointSpeedRatio(ROBOT_JOINT4, 0.5);

//robot.setJointSpeeds(90, 0, 0, 90);
```

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 explicitely sets the joint speeds of joints 2 and 3 to zero.

The next two lines,

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

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

The next lines,

```
speed = (3.5/2) * M_PI / 2; // 2.75 inch/s
radius = 3.5/2;             // radius is 1.75
robot.setTwoWheelRobotSpeed(speed, radius);
```

use the `setTwoWheelRobotSpeed()` function to set the faceplate joint speeds for a robot 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.

The following two lines,

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

rotate the faceplates forward at the necessary rate to achieve a forward speed of 2.75 inches per second.

Finally, the last two lines,

```
printf("Move continuously for 3 seconds.\n");
robot.moveContinuousTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD, 3);
```

roll the robot forward for three seconds. The enumerated values `ROBOT_FORWARD` and `ROBOT_BACKWARD` indicate the forward and backward directions for each motor to turn, respectively. `ROBOT_NEUTRAL` indicates that the motor should not turn, but should remain flexible and backdrivable. `ROBOT_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.2. 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.

# 6 Preprogrammed Motions

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

- `motionArch()`: This function causes the robot to arch up for better clearance.
- `motionInchwormLeft()`: This function causes the robot to perform the inchworm gait once, moving the robot towards its left.
- `motionInchwormRight()`: This function causes the robot to perform the inchworm gait once, moving the robot towards its right.
- `motionRollBackward()`: This function causes the robot to rotate its faceplates, using them as wheels to roll backward.
- `motionRollForward()`: This function causes the robot to rotate its faceplates, using them as wheels to roll forward.
- `motionSkinny()`: This function makes the robot assume a skinnier rolling profile.
- `motionStand()`: This function causes the robot to stand up onto a faceplate, assuming the camera platform position.
- `motionTumble()`: This function causes the robot to perform the tumbling motion, flipping end over end.
- `motionTurnLeft()`: This function uses the robot's faceplates as wheels, turning them in opposite directions in order to rotate the robot towards its left.
- `motionTurnRight()`: This function uses the robot's faceplates as wheels, turning them in opposite directions in order to rotate the robot towards its right.
- `motionUnstand()`: This function causes the robot 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 8.

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

### 6.1.1 inchworm.ch Source Code

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

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

/* Set robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);

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

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

### 6.1.2 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 reperesent our robot and connect to the robot with the following lines.

```
CMobot robot;

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

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

```
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);
```

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

```
robot.moveToZero();
```

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

```
robot.motionInchwormLeft(4);
```

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

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

### 6.2.1 `stand.ch` Source Code

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

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

### 6.2.2 `stand.ch` Explained

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

```
robot.motionStand();
```

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

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

pauses the program for three seconds, causing the robot to remain still for three seconds. If you would like to pause a program for less than one second, use the function `msleep(msecs)` to pause for "msecs" milliseconds.

After the pause is over, the line

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

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

```
robot.motionUnstand();
```

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

## 6.3 `tumble.ch`: A Demo Using the `motionTumble()` Preprogrammed Motion

### 6.3.1 `tumble.ch` Source Code

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

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

/* Tumble two times */
robot.motionTumble(2);
```

### 6.3.2 `tumble.ch` Explained

The first portion of the program,

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

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

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

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

```
robot.motionTumble(2);
```

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

## 6.4 `motion.ch`: A Demo Using Multiple Preprogrammed Motions

### 6.4.1 `motion.ch` Source Code

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

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

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

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

### 6.4.2 `motion.ch` Explained

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

The next portion of code executes a number of different pregrammed motions to demonstrate the motion capabilities of the robot. First, the "Arch" motion is demonstrated by the following line of code:

```
robot.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,

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

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

Next, the lines

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

make the robot 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 robot backward using its faceplates as wheels by rotating the faceplates 360 degrees.

Next, the robot stands up by executing the line

```
robot.motionStand();
```

While the robot is standing, the line

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

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

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

```
robot.motionUnstand();
```

Finally, we perform the tumbling motion.

```
robot.motionTumble(2);
```

The tumbling motion is a movement in which the robot 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.

# 7 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 ”motionTumble()” and ”motionUnstand()” will be discussed in Section 8.

## 7.1 Inchworm Gait Demo

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

### 7.1.1 `inchworm2.ch` Source Code

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

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

/* Set robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);

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

/* Do the inchworm motion four times */
int i, num = 4;
double angle2 = -45;
double angle3 = 45;
for(i = 0; i < num; i++) {
    robot.moveJointTo(ROBOT_JOINT2, angle2); /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, angle3); /* Move joint 3 */
    robot.moveJointTo(ROBOT_JOINT2, 0);      /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, 0);      /* Move joint 3 back to zero position */
}
```

### 7.1.2 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 robot;

/* Connect to the paired Mobot */
robot.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 robot motors to speed of 0.50 */
robot.setJointSpeedRatio(ROBOT_JOINT2, 0.50);
robot.setJointSpeedRatio(ROBOT_JOINT3, 0.50);
```

Next, we move the robot into a flat "zero" position, as shown in Figure 12a.

```
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.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:

(a) Zero Position (b) Move joint 2 (c) Move joint 3 (d) Move joint 2 (e) Move joint 3 back to zero position

Figure 12: The inchworm motion

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 12b)
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 12c)
3. Joint A moves back to a flat position. (Figure 12d)
4. Joint B moves back to a flat position. (Figure 12e)
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++) {
    robot.moveJointTo(ROBOT_JOINT2, angle2); /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, angle3); /* Move joint 3 */
    robot.moveJointTo(ROBOT_JOINT2, 0);      /* Move joint 2 */
    robot.moveJointTo(ROBOT_JOINT3, 0);      /* Move joint 3 bace 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.

## 7.2 Standing Demo

### 7.2.1 `stand2.ch` Source Code

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

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

/* Set robot motors to speed of 90 degrees per second */
```

```
robot.setJointSpeed(ROBOT_JOINT2, 90);
robot.setJointSpeed(ROBOT_JOINT3, 90);
/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.moveToZero();

/* Move the robot into a fetal position */
robot.moveJointTo(ROBOT_JOINT2, -85);
robot.moveJointTo(ROBOT_JOINT3, 70);

/* Wait a second for the robot to settle down */
sleep(1);

/* Rotate the bottom faceplate by 45 degrees */
robot.moveJointTo(ROBOT_JOINT1, 45);

/* Lift the body up */
robot.moveJointTo(ROBOT_JOINT2, 20);

/* Pan the robot around for 3 seconds at 45 degrees per second*/
robot.setJointSpeed(ROBOT_JOINT1, 45);
robot.moveContinuousTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_HOLD, 3);
```

### 7.2.2 `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 robot into a flat position. Next, the following lines are executed:

```
robot.moveJointTo(ROBOT_JOINT2, -85);
robot.moveJointTo(ROBOT_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,

```
sleep(1);
```

causes the program to pause for one second before continuing. This allows the robot 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.

```
robot.moveJointTo(ROBOT_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.

```
robot.moveJointTo(ROBOT_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.

```
robot.setJointSpeed(ROBOT_JOINT1, 45);
robot.moveContinuousTime(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_HOLD, 3);
```

# 8 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 robot 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.

## 8.1 List of Blocking Movement Functions

- `move()`
- `moveContinuousTime()`
- `moveJoint()`
- `moveJointTo()`
- `moveJointWait()`
- `moveTo()`
- `moveToZero()`
- `moveWait()`
- `motionArch()`
- `motionInchwormLeft()`
- `motionInchwormRight()`
- `motionRollBackward()`
- `motionRollForward()`
- `motionSkinny()`
- `motionStand()`
- `motionTumble()`
- `motionTurnLeft()`
- `motionTurnRight()`
- `motionUnstand()`

## 8.2 List of Non-Blocking Movement Functions

- moveNB()
- moveContinuousNB()
- moveJointNB()
- moveJointToNB()
- moveToNB()
- moveToZeroNB()
- motionArchNB()
- motionInchwormLeftNB()
- motionInchwormRightNB()
- motionRollBackwardNB()
- motionRollForwardNB()
- motionSkinnyNB()
- motionStandNB()
- motionTumbleNB()
- motionTurnLeftNB()
- motionTurnRightNB()
- motionUnstandNB()

## 8.3 Blocking and Non-Blocking Demo Programs

### 8.3.1 nonblock.ch Source Code

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

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

robot.moveToZero();

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

### 8.3.2 `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:

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

The previous lines of code basically loops as long as the `robot.isMoving()` function is returning true. In other words, in plain english, as long as the robot is moving, the program will print the message "robot is moving...". As soon as the robot 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

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

If the `moveNB()` function is replaced with the `move()` function in this program, the message "robot 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 robot has already stopped moving.

### 8.3.3 `nonblock2.ch` Source Code

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

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

robot.moveToZero();

/* Rotate each of the faceplates by 360 degrees */
//robot.moveJoint(ROBOT_JOINT1, 360); // Blocking version
robot.moveJointNB(ROBOT_JOINT1, 360); // Non-Blocking version
robot.moveJoint(ROBOT_JOINT4, 360);
```

### 8.3.4 `nonblock2.ch` Source Code Explanation

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

```
robot.moveJointNB(ROBOT_JOINT1, 360); // Non-Blocking version
robot.moveJoint(ROBOT_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 robots, 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.

### 8.3.5 `nonblock3.ch` Source Code

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

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

robot.moveToZero();

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

### 8.3.6 `nonblock3.ch` Source Code Explanation

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

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

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

```
robot.motionArchNB(15);
robot.motionRollForwardNB(360);
```

The first call is to the function `motionArchNB()`, which arches the robot up by moving joints 2 and 3. Since it is a non-blocking function, the program immediately continues on even before the arching motion as finished. The next call rolls the robot forward by rotating joints 1 and 4. In effect, these two lines cause the robot 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

```
robot.motionWait();
```

The `motionWait()` function will wait until all robot 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 robot in the middle of one of its motions.

## 8.4 Preprogrammed Motion Demos with Non-Blocking Functions

### 8.4.1 `unstand2.ch` Source Code

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

```
CMobot robot;

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

robot.moveJointToNB(ROBOT_JOINT2, -85);
robot.moveJointToNB(ROBOT_JOINT3, 45);
robot.moveWait();
robot.moveToZero();
```

### 8.4.2 `unstand2.ch` Source Code Explanation

The first block of code,

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

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

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

```
robot.moveJointToNB(ROBOT_JOINT2, -85);
robot.moveJointToNB(ROBOT_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 robot to complete its motions, as done in the next line:

```
robot.moveWait();
```

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

```
robot.moveToZero();
```

### 8.4.3 `tumble2.ch` Source Code

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

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

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

/* Begin tumbling for "num" times */
int i, num = 2;
for(i = 0; i < num; i++) {
```

```
    /* First lift and tumble */
    robot.moveJointTo(ROBOT_JOINT2, -85);
    robot.moveJointTo(ROBOT_JOINT3, 80);
    robot.moveJointTo(ROBOT_JOINT2, 0);
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT2, 45);
    /* Second lift and tumble */
    robot.moveJointTo(ROBOT_JOINT3, -85);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 0);
    robot.moveJointTo(ROBOT_JOINT3, 80);
    if(i != (num-1)) { /* Do not perform this motion on the last tumble */
        robot.moveJointTo(ROBOT_JOINT3, 45);
    }
}

/* Unstand the robot */
robot.moveJointToNB(ROBOT_JOINT2, 0);
robot.moveJointToNB(ROBOT_JOINT3, 0);
robot.moveWait();
robot.moveToZero();
```

### 8.4.4 `tumble2.ch` Source Code Explanation

The first lines of the program,

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

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

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot.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 robot flipping over twice. The reason the robot flips twice per tumble is so that when the tumbling motion is done, the robot 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 */
    robot.moveJointTo(ROBOT_JOINT2, -85);
    robot.moveJointTo(ROBOT_JOINT3, 80);
    robot.moveJointTo(ROBOT_JOINT2, 0);
```

```
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT2, 45);
```

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

```
    /* Second lift and tumble */
    robot.moveJointTo(ROBOT_JOINT3, -85);
    robot.moveJointTo(ROBOT_JOINT2, 80);
    robot.moveJointTo(ROBOT_JOINT3, 0);
    robot.moveJointTo(ROBOT_JOINT2, 0);
    robot.moveJointTo(ROBOT_JOINT3, 80);
    if(i != (num-1)) { /* Do not perform this motion on the last tumble */
        robot.moveJointTo(ROBOT_JOINT3, 45);
    }
}
```

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

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

```
/* Unstand the robot */
robot.moveJointToNB(ROBOT_JOINT2, 0);
robot.moveJointToNB(ROBOT_JOINT3, 0);
robot.moveWait();
robot.moveToZero();
```

# 9 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.

## 9.1 `twoModules.ch` Source Code

```
/* Filename: twoModules.ch
 * Control two modules and make them stand and inchworm simultaneously. */
#include <mobot.h>
CMobot robot1;
CMobot robot2;

/* Connect robot variables to the robot modules. The */
robot1.connect();
robot2.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot1.moveToZeroNB();
```

```
robot2.moveToZeroNB();

robot1.moveWait();
robot2.moveWait();

/* Instruct the first robot to stand and the second robot to inchworm left four
 * times simultaneously. As soon as the first robot stands up, rotate its joints
 * 1 and 4 360 degrees. */
robot1.motionStandNB();
robot2.motionInchwormLeftNB(4);
robot1.motionWait();
robot1.moveNB(360, 0, 0, 360);
robot1.moveWait();
robot2.motionWait();
/* Instruct the first robot unstand and the second robot inchworm right four
 * times simultaneously. */
robot1.motionUnstandNB();
robot2.motionInchwormRightNB(4);
robot1.motionWait();
robot1.motionTumble(1);
robot2.motionWait();
```

## 9.2 Demo Explanation

The first two lines of interest appear as such:

```
CMobot robot1;
CMobot robot2;
```

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.

```
robot1.connect();
robot2.connect();
```

These two lines connect the robots to the first two addresses of the known robot addresses. The list of the computer's known robot addresses may be configured in the process detailed in Section 2 on page 10. For each separate control program, the first call to the `connect()` member function will connect to the first robot listed in the configuration file. Each successive call to the `connect()` function will connect to successive robots listed in the configuration file. The order in which they are connected may be modified using the "Configure Robot Bluetooth" dialog, as discussed in Section 2.

```
robot1.moveToZeroNB();
robot2.moveToZeroNB();
```

These two lines command the two robots 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 robot to finish completing the motion before commanding the second robot to begin. In a normal program, this effectively causes both robots to move to their zero positions simultaneously.

```
robot1.moveWait();
robot2.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 robots, we can be assured that the robots have finished moving before the program continues.

```
robot1.motionStandNB();
robot2.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 robots will effectively perform the motions simultaneously.

When the first robot finishes standing, we want it to spin around on its faceplate while the second robot 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 robot, as in the following lines of code.

```
robot1.motionWait();
robot1.moveNB(360, 0, 0, 360);
```

Before we continue with the program, we wish to ensure that motion has stopped on both robots. Since the the last command sent to `robot1` was a `moveNB()` command, we use the `moveWait()` function to wait for that movement to finish. Similarly, the last command sent to `robot2` 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:

```
robot1.moveWait();
robot2.motionWait();
```

Finally, the following lines,

```
robot1.motionUnstandNB();
robot2.motionInchwormRightNB(4);
robot1.motionWait();
robot1.motionTumble(1);
robot2.motionWait();
```

make the first robot come back down from its standing position and the second robot 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 `robot1` finishes unstanding, it will begin a tumbling motion. Since the tumbling motion function is called before the call to `robot2.motionWait()`, `robot1` will begin tumbling even if `robot2` has finished its previous motion, which was inchworming 4 times.

## 9.3 Controlling Multiple Connected Modules

### 9.3.1 `lift.ch`, Lifting Demo

```
/* Filename: lift.ch
   Lift two connected robots.
   Joint 4 of the 1st robot should be connected to
   Joint 1 of the 2nd robot as
           1st                       2nd
  |---------|--------|      |---------|--------|
 1|   2     |   3    | 4 X 1|    2    |   3    | 4
  |---------|--------|      |---------|--------|
*/
```

```
#include <mobot.h>
CMobot robot1;
CMobot robot2;

/* Connect robot variables to the robot modules. */
robot1.connect();
robot2.connect();

/* Set the robot to "home" position, where all joint angles are 0 degrees. */
robot1.moveToZeroNB();
robot2.moveToZeroNB();

robot1.moveWait();
robot2.moveWait();

/* First lift */
robot1.moveToNB(0, 0, 90,  0);
robot2.moveToNB(0,  -90, 0, 0);
robot1.moveWait();
robot2.moveWait();
sleep(1);
robot1.moveToNB(0, -90,  0, 0);
robot2.moveToNB(0, 0, 90, 0);
robot1.moveWait();
robot2.moveWait();
sleep(1);
robot1.moveToNB(0, 0, 90,  0);
robot2.moveToNB(0,  -90, 0, 0);
robot1.moveWait();
robot2.moveWait();
/* Second lift */
/* Move to zero position */
robot1.moveToZeroNB();
robot2.moveToZeroNB();
robot1.moveWait();
robot2.moveWait();
```

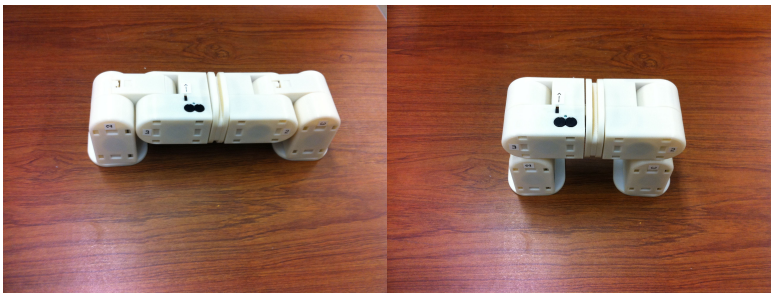
### 9.3.2 `lift.ch` Source Code Explanation

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

The first portion of the program initialize two variables called `robot1` and `robot2`, which will be used to control the two connected modules. Once the robots are initialized, they are both moved into a flat zero position.

The next lines,

```
/* First lift */
robot1.moveNB(0, -90,  0, 0);
robot2.moveNB(0, 0, 90, 0);
robot1.moveWait();
robot2.moveWait();
```



(a) First lift (b) Second lift

Figure 13: Lifting motion with two connected modules

rotate the joints on either end of the compound robot to perform the first portion of the lift, as shown in Figure 13a. 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 robot to lift the robot one step higher, as shown is Figure 13b.

```
/* Second lift */
robot1.moveToNB(0, 0, 90,  0);
robot2.moveToNB(0,  -90, 0, 0);
robot1.moveWait();
robot2.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 robots back to their zero positions, which drops the compound robot back onto the ground.

```
robot1.moveToZeroNB();
robot2.moveToZeroNB();
robot1.moveWait();
robot2.moveWait();
```

# 10 Commanding Multiple Robots to Perform Identical Tasks

The class called `CMobotGroup` can be used to control multiple modules simultaneously. The `CMobotGroup` represents a group of robots. 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 C. Following is a complete listing of the available member functions in the `CMobotGroup` class.

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

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

## 10.1 Demo program `group.ch`

### 10.1.1 Source Code

```
/* Filename: group.ch
 * Control multiple Mobot modules simultaneously using the CMobotGroup class */
#include <mobot.h>
CMobot robot1;
CMobot robot2;
CMobotGroup group;

/* Connect to the robots listed in the configuration file. */
robot1.connect();
robot2.connect();

/* Add the two modules to be members of our group */
group.addRobot(robot1);
group.addRobot(robot2);

/* Now, any commands given to "group" will cause both robot1 and robot2 to
 * execute the command. */
group.motionInchwormLeft(4);  /* Both robots inchworm left 4 times */
group.motionStand();          /* Both robots stand */
group.move(360, 0, 0, 360);   /* Joints 1 and 4 rotate 360 degrees */
sleep(3);                     /* Robots stand still for 3 seconds */
```

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

### 10.1.2 Demo Explanation

The first lines of interest appear as such:

```
CMobot robot1;
CMobot robot2;
CMobotGroup group;
```

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

```
robot1.connect();
robot2.connect();
```

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

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

Finally, we wish for all of the robots in our robot group, namely `robot1` and `robot2`, to peform an inchworm motion four times, followed by a standing motion. This is done with the following lines:

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

After the robots stand up, the line

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

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

The next line,

```
sleep(3);                     /* Robots stand still for 3 seconds */
```

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

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

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

## 10.2 Demo Program `groups.ch`

### 10.2.1 `groups.ch` Source Code

```
/* Filename: groups.ch
 * Control multiple Mobot groups simultaneously using the CMobotGroup class */
#include <mobot.h>
CMobot robot1;
CMobot robot2;
CMobot robot3;
CMobot robot4;
CMobotGroup groupA;
CMobotGroup groupB;
```

```
CMobotGroup groupC;
CMobotGroup groupD;

/* Connect to the robots listed in the configuration file. */
robot1.connect();
robot2.connect();
robot3.connect();
robot4.connect();

/* Add the robots 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(robot1);
groupA.addRobot(robot2);
groupA.addRobot(robot3);
groupA.addRobot(robot4);

/* Group B */
groupB.addRobot(robot1);
groupB.addRobot(robot2);

/* Group C */
groupC.addRobot(robot3);
groupC.addRobot(robot4);

/* Group D */
groupD.addRobot(robot1);
groupD.addRobot(robot2);
groupD.addRobot(robot3);

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

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

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

/* Make robot 4 unstand and inchworm while the remaining robots spin. */
```

```
groupD.moveContinuousNB(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD);
robot4.motionUnstand();
robot4.motionInchwormLeft(2);
groupD.motionUnstand();
```

### 10.2.2 Demo Explanation

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

```
#include <mobot.h>
CMobot robot1;
CMobot robot2;
CMobot robot3;
CMobot robot4;
CMobotGroup groupA;
CMobotGroup groupB;
CMobotGroup groupC;
CMobotGroup groupD;
```

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

```
robot1.connect();
robot2.connect();
robot3.connect();
robot4.connect();
```

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

- Group A: robot1, robot2, robot3, robot4
- Group B: robot1, robot2
- Group C: robot3, robot4
- Group D: robot1, robot2, robot3

The following code divides the robots up into our groups.

```
/* Group A */
groupA.addRobot(robot1);
groupA.addRobot(robot2);
groupA.addRobot(robot3);
groupA.addRobot(robot4);

/* Group B */
groupB.addRobot(robot1);
groupB.addRobot(robot2);

/* Group C */
groupC.addRobot(robot3);
groupC.addRobot(robot4);

/* Group D */
```

```
groupD.addRobot(robot1);
groupD.addRobot(robot2);
groupD.addRobot(robot3);
```

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 robots, which are members of the group `groupA`, stand up.

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

While the robots 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 robots 1 and 2 (Group B) rotate counter-clockwise and robots 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 robots 1, 2, and 3 to spin while robot 4 unstands and inchworms twice. As soon as robot 4 is done inchworming, we want robots 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 robot 4 and inchworm it twice. After robot 4 is done inchworming, we unstand Group D.

```
/* Make robot 4 unstand and inchworm while the remaining robots spin. */
groupD.moveContinuousNB(ROBOT_FORWARD, ROBOT_HOLD, ROBOT_HOLD, ROBOT_FORWARD);
robot4.motionUnstand();
robot4.motionInchwormLeft(2);
groupD.motionUnstand();
```

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

## 11.1 Example 1

### 11.1.1 Problem Statement

Roll a robot 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 $\theta = 45t$.

### 11.1.2 `dataAcquisition.ch` Source Code

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

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

/* Connect to the robot */
robot.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;
array double angles1Unwrapped[numDataPoints];

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

/* Start the motion. First, move robot to zero position */
robot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

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

/* Wait for recording to finish */
robot.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.data2D(time, angles1);
plot1.grid(PLOT_ON);
plot1.plotting();

/* Plot the unwrapped data */
unwrapdeg(angles1Unwrapped, angles1);
plot2.title("Unwrapped Data for Joint Angle 1 versus Time");
plot2.label(PLOT_AXIS_X, "Time (seconds)");
plot2.label(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2D(time, angles1Unwrapped);
plot2.grid(PLOT_ON);
plot2.plotting();

/* Adjust the time delay */
/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time, angles1Unwrapped);

/* Plot the data */
plot3.title("Unwrapped and shifted Data for Joint Angle 1 versus Time");
plot3.label(PLOT_AXIS_X, "Time (seconds)");
plot3.label(PLOT_AXIS_Y, "Angle (degrees)");
plot3.data2D(time, angles1Unwrapped);
plot3.grid(PLOT_ON);
plot3.plotting();
```

### 11.1.3 `dataAcquisition.ch` Explained

The first block of code,

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

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

includes header files and declares our robot 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 are covered in the 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 robot 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 robot 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 robot. 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 robot. 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 robot'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 robot to zero position */
robot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);
```

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

```
robot.recordAngle(ROBOT_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:

```
robot.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:

```
robot.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.label(PLOT_AXIS_X, "Time (seconds)");
plot1.label(PLOT_AXIS_Y, "Angle (degrees)");
plot1.data2D(time, angles1);
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 14.

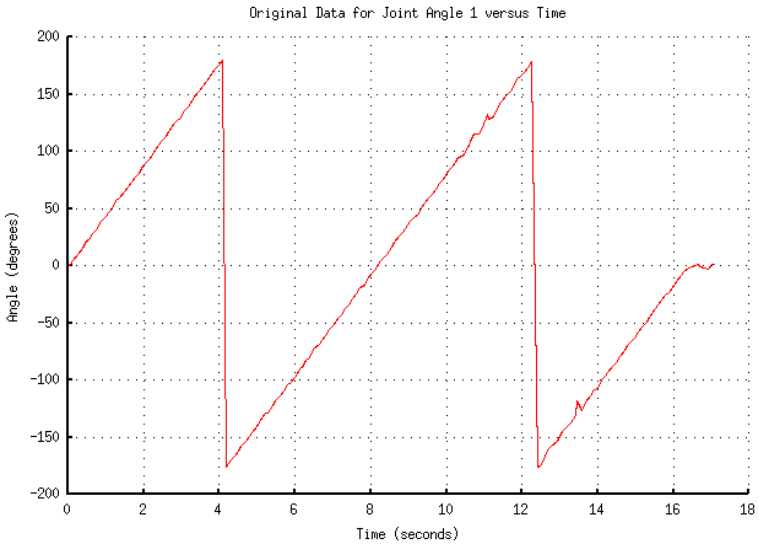



Figure 14: The original data recorded by `dataAcquisition.ch`.

Figure 14 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.label(PLOT_AXIS_X, "Time (seconds)");
plot2.label(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2D(time, angles1_unwrapped);
plot2.grid(PLOT_ON);
plot2.plotting();
```

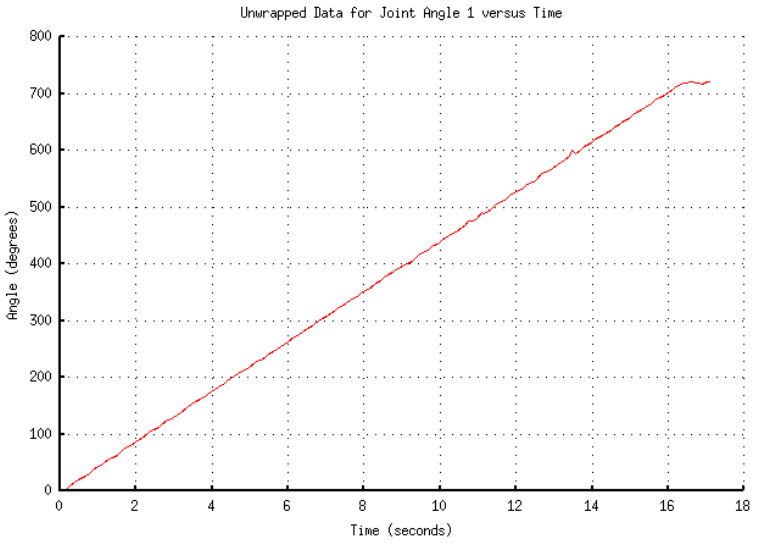



Figure 15: The unwrapped data recorded by `dataAcquisition.ch`.

Figure 15 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 15, 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.label(PLOT_AXIS_X, "Time (seconds)");
plot3.label(PLOT_AXIS_Y, "Angle (degrees)");
plot3.data2D(time, angles1_unwrapped);
plot3.grid(PLOT_ON);
plot3.plotting();
```

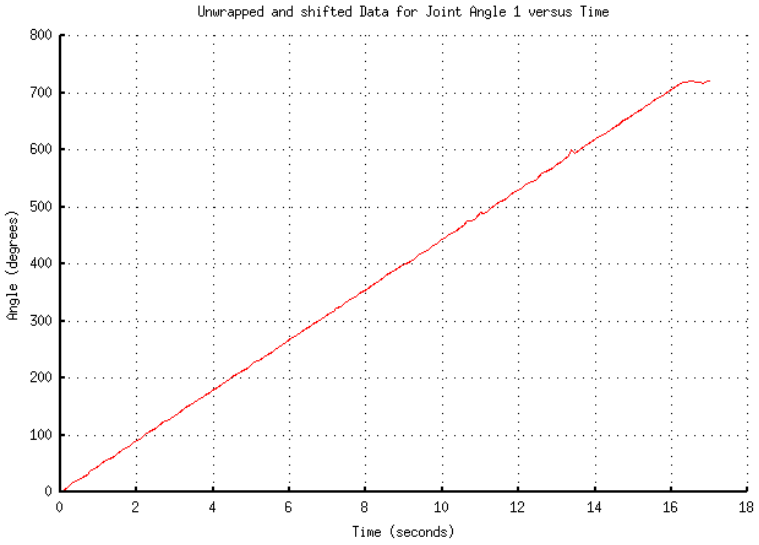



Figure 16: The unwrapped and shifted data recorded by `dataAcquisition.ch`.

Figure 16 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.

### 11.1.4 `dataAcquisition1.ch` Source Code

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

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

/* Connect to the robot */
robot.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 robot to zero position */
robot.moveToZero();
/* Set the joint 1 speed to 45 degrees/second */
robot.setJointSpeed(ROBOT_JOINT1, speed);
robot.setJointSpeed(ROBOT_JOINT4, speed);

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

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

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

/* Plot the unwrapped data */
unwrapdeg(angles1Unwrapped, angles1);

/* Adjust the time delay */
/* Shift the time so the movement starts at time 0 */
shiftTime(tolerance, numDataPoints, time, angles1Unwrapped);

/* 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.data2D(time, angles1Unwrapped);
plot.grid(PLOT_ON);
plot.plotting();
```

## 11.2 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.

### 11.2.1 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.

### 11.2.2 `dataAcquisition2.ch` Source Code

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

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

/* Connect to the robot */
robot.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;
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 */

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

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

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

/* Perform the standing and unstanding motions */
robot.motionTurnRight(360);
```

```
robot.motionInchwormLeft(2);

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

/* Plot the unwrapped 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);
plot.title("Unwrapped Data for Joint Angles versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Angle (degrees)");
plot.data2D(time, angles1_unwrapped);
plot.data2D(time, angles2_unwrapped);
plot.data2D(time, angles3_unwrapped);
plot.data2D(time, angles4_unwrapped);
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();
```

### 11.2.3 `dataAcquisition2.ch` Explained

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

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

/* Connect to the robot */
robot.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 robot into its zero postition.

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

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

Next, we begin recording the data.

```
robot.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:

```
robot.motionTurnRight(360);
robot.motionInchwormLeft(2);
```

Finally, we wait for the recording to finish.

```
robot.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.label(PLOT_AXIS_X, "Time (seconds)");
plot2.label(PLOT_AXIS_Y, "Angle (degrees)");
plot2.data2D(time, angles1_unwrapped);
plot2.data2D(time, angles2_unwrapped);
plot2.data2D(time, angles3_unwrapped);
plot2.data2D(time, angles4_unwrapped);
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 occurance 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.

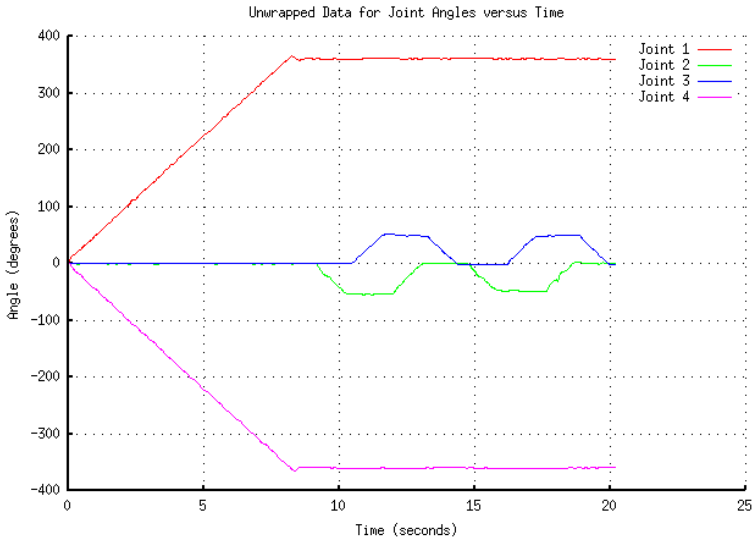



Figure 17: The unwrapped and shifted data recorded by `dataAcquisition2.ch`.

Figure 17 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 robot to the right. After rotating 360 degrees, joints 2 and 3 move to inchworm the robot to the left twice.

## 11.3 Example 3

### 11.3.1 Problem Statement

A robot is equipped with 3.5 inch diameter wheels. Write a piece of code which rolls the robot forward at 2.5 inches per second for a distance of 12 inches.

### 11.3.2 `dataAcquisition3.ch` Source Code

```
/* Filename: dataAcquisition3.ch
```

```
 * Make a graph of the robot's joint angle versus time */

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

/* Connect to the robot */
robot.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;
array double angles1Unwrapped[numDataPoints];
double tolerance = 1.0; /* Degrees */

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

/* Start capturing data */
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);

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

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

/* Unwrap the data */
unwrapdeg(angles1Unwrapped, angles1);
/* Shift the data */
```

```
shiftTime(tolerance, numDataPoints, time, angles1Unwrapped);
/* Convert angles to displacement */
distances = angle2distance(radius, angles1Unwrapped);

/* Plot the unwrapped data */
plot.title("Displacement versus Time");
plot.label(PLOT_AXIS_X, "Time (seconds)");
plot.label(PLOT_AXIS_Y, "Displacement (inches)");
plot.data2D(time, distances);
plot.grid(PLOT_ON);
plot.plotting();
```

### 11.3.3 `dataAcquisition3.ch` Explained

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

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

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

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

```
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 */
```

Is 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 robot to zero position and set the wheel speeds.

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

Start recording the angle of the first joint.

```
robot.recordAngle(ROBOT_JOINT1, time, angles1, numDataPoints, timeInterval);
```

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

```
robot.motionRollForward(angle);
```

Wait for the recording to finish.

```
robot.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.

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

where $d$ is the distance travelled, $\theta$ is the angle rotated, 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.data2D(time, distances);
plot.grid(PLOT_ON);
plot.plotting();
```

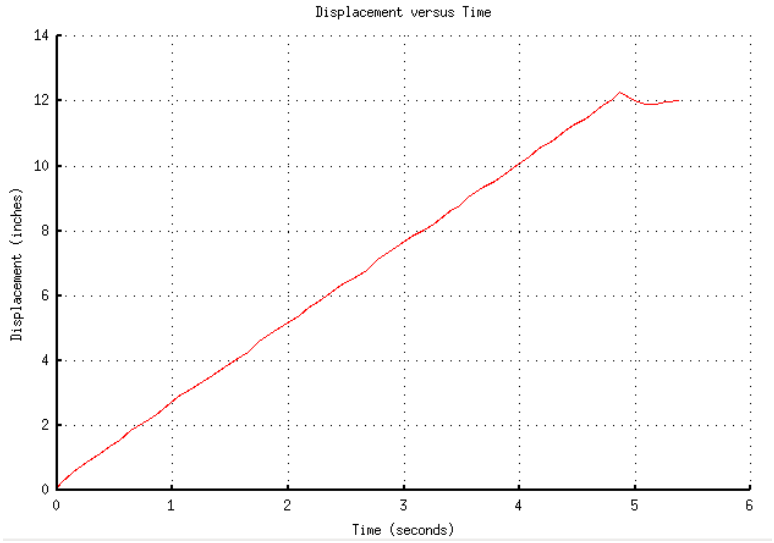



Figure 18: The unwrapped data recorded by `dataAcquisition2.ch`.

Figure 18 displays the distance data acquired during the motion. It can be seen that the robot 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

$$t = d/2.5 = 12/2.5 = 4.8\text{seconds} \tag{1}$$

which may also be verified from the graph.

# 12 Recalibrating the Mobot

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

- Press both of the Mobot's "A" and "B" buttons simultaneously. The blue status LED light on the Mobot should blink several times and then remain off.
- Power off the Mobot.
- Manually move the Mobot joints into the correct zero position, as shown in Figure 11. This may be done visually, or by using flat surfaces as a guide.
- Power on the Mobot while it is in the correct zero position. After calibration, the Mobot will perform a short movement routine to test all of its joints. The Mobot has now been recalibrated.

# A Data Types

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

| Data Type | Description |
|---|---|
| `robotJointId_t` | An enumerated value that indicates a Mobot joint. |
| `robotJointState_t` | The current state of a Mobot joint. |

## A.1 `robotJointId_t`

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

```
typedef enum mobot_joints_e {
  ROBOT_JOINT1 = 1,
  ROBOT_JOINT2 = 2,
  ROBOT_JOINT3 = 3,
  ROBOT_JOINT4 = 4
} robotJointId_t;
```

| Value | Description |
|---|---|
| `ROBOT_JOINT1` | Joint number 1 on the Mobot, which is a faceplate joint. |
| `ROBOT_JOINT2` | Joint number 2 on the Mobot, which is a body joint. |
| `ROBOT_JOINT3` | Joint number 3 on the Mobot, which is a body joint. |
| `ROBOT_JOINT4` | Joint number 4 on the Mobot, which is a faceplate joint. |

## A.2 `robotJointState_t`

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

```
typedef enum mobot_joint_state_e {
  ROBOT_NEUTRAL   = 0,
  ROBOT_FORWARD   = 1,
  ROBOT_BACKWARD  = 2,
  ROBOT_HOLD      = 3
} robotJointState_t;
```

| Value | Description |
|---|---|
| `ROBOT_NEUTRAL` | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| `ROBOT_FORWARD` | This value indicates that the joint is currently moving forward. |
| `ROBOT_BACKWARD` | This value indicates that the joint is currently moving backward. |
| `ROBOT_HOLD` | This value indicates that the joint is currently not moving and is holding its current position. The joint is not currently backdrivable. |

# B CMobot API

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

Table 1: CMobot Member Functions.

| Function | Description |
|---|---|
| `CMobot()` | The CMobot constructor function. This function is called automatically and should not be called explicitly. |
| `~CMobot()` | The CMobot destructor function. This function is called automatically and should not be called explicitly. |
| `connect()` | Connect to a remote robot module. This function connects to the first robot listed in the Barobo configuration file. To edit the configuration file, use the robot control graphical user interface, and select the menu item "Robot → Configure Robot Bluetooth". |
| `connectWithAddress()` | Connect to a robot module by specifying its Bluetooth address. |
| `disconnect()` | Disconnect from a robot module. |
| `getJointAngle()` | Get a joint's angle. |
| `getJointMaxSpeed()` | Get a joint's maximum speed in radians per second. |
| `getJointSpeed()` | Get a motor's current speed setting in radians per second. |
| `getJointSpeeds()` | Get all motor's current speed settings in radians per second. |
| `getJointSpeedRatio()` | Get a motor's speed as a ratio of the motor's maximum speed. |
| `getJointSpeedRatios()` | Get all motor speeds as ratios of the motor's maximum speed. |
| `getJointState()` | Get a motor's current status. |
| `isConnected()` | This function is used to check the connection to a robot. |
| `isMoving()` | This function is used to check if any joints are currently in motion. |
| `move()` | Move all four joints of the robot by specified angles. |
| `moveNB()` | Identical to `move()` but non-blocking. |
| `moveContinuousNB()` | Move joints continuously. Joints will move untill stopped. |
| `moveContinuousTime()` | Move joints continuously for a certain amount of time. |
| `moveJoint()` | Move a joint. |
| `moveJointNB()` | Move a joint. |
| `moveJointTo()` | Set the desired motor position for a joint. |
| `moveJointToNB()` | Identical to `moveJointTo()` but non-blocking. |
| `moveJointWait()` | Wait until the specified motor has stopped moving. |
| `moveTo()` | Move all four joints of the robot to specified absolute angles. |
| `moveToNB()` | Identical to `moveTo()` but non-blocking. |
| `moveToZero()` | Instruct all motors to go to their zero positions. |
| `moveToZeroNB()` | Identical to `moveToZero()` but non-blocking. |
| `moveWait()` | Wait until all motors have stopped moving. |
| `recordAngle()` | Begin recording the angle of a joint. |
| `recordAngles()` | Begin recording the angle of all joints. |
| `recordWait()` | Wait for recording operations to finish. |
| `setJointSpeed()` | Set a motor's speed setting in radians per second. |
| `setJointSpeeds()` | Set all motor speeds in radians per second. |
| `setJointSpeedRatio()` | Set a joints speed setting to a fraction of its maximum speed, a value between 0 and 1. |
| `setJointSpeedRatios()` | Set all joint speed settings to a fraction of its maximum speed, expressed as a value from 0 to 1. |
| `stop()` | Stop all currently executing motions of the robot. |

Table 2: CMobot Member Functions for Compound Motions.

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

## CMobot::connect()

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

**Purpose**
Connect to a remote robot 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 robot. The function looks inside of a Barobo configuration file and connects to the first robot listed in the file. The configuration file may be created and/or modified using the Robot Controller Interface, and selecting the "Robot → Configure Robot Bluetooth" menu item.
**Example**
Please see the example in Section 4.1.2 on page 19.

**See Also**
`connectWithAddress(), disconnect()`

## CMobot::connectWithAddress()

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

**Purpose**
Connect to a remote robot 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 robot.
`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 robot.
**Example**

**See Also**
`connect(), disconnect()`

## CMobot::disconnect()

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

**Purpose**
Disconnect from a remote robot.

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

**Parameters**
None.

**Description**
This function is used to disconnect from a robot. 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::getJointAngle()

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

**Purpose**
Retrieve a robot joint's current angle.

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

**Parameters**

| | |
|---|---|
| `id` | The joint number. This is an enumerated type discussed in Section A.1 on page 66. |
| `angle` | A variable to store the current angle of the robot 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 robot's motor. The angle returned is in units of degrees and is accurate to roughly $\pm 0.17$ degrees.

**Example**

**See Also**

---

## CMobot::getJointMaxSpeed()

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

**Purpose**
Get the maximum speed of a joint on the robot.

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

**Parameters**

| | |
|---|---|
| `id` | The joint number. This is an enumerated type discussed in Section A.1 on page 66. |
| `speed` | A variable of type `double`. 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::getJointSpeed()

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

**Purpose**
Get the speed of a joint on the robot.

**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.1 on page 66. |
| `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(robotJointId_t id, double &ratio);
```

**Purpose**
Get the speed ratio settings of a joint on the robot.

**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.1 on page 66. |
| `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 robot.

**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 robot 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 robot.

**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 robot simultaneously. The speeds are in degrees per second.

**Example**

**See Also**
`setJointSpeeds()`, `getJointSpeedRatios()`, `getJointSpeed()`

---

# CMobot::getJointState()

**Synopsis**
```
#include <mobot.h>
int CMobot::getJointState(robotJointId_t id, robotJointState_t &state);
```

**Purpose**
Determine whether a motor is moving or not.

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

**Parameters**
- `id` The joint number. This is an enumerated type discussed in Section A.1 on page 66.
- `state` An integer variable which will be overwritten with the current state of the motor. This is an enumerated type discussed in Section A.2 on page 66.

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

| Value | Description |
|---|---|
| `ROBOT_NEUTRAL` | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| `ROBOT_FORWARD` | This value indicates that the joint is currently moving forward. |
| `ROBOT_BACKWARD` | This value indicates that the joint is currently moving backward. |
| `ROBOT_HOLD` | 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 robot via Bluetooth.

**Return Value**
The function returns zero if it is not currently connected to a robot or if an error has occured, or 1 if the robot is connected.

**Parameters**
None.

**Description**
This function is used to check if the software is currently connected to a robot.

**Example**

**See Also**
`connect(), disconnect()`

---

## CMobot::isMoving()

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

**Purpose**
Check to see if a robot 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 occured, or 1 if any joint is being driven. A value of 1 is equivalent to either a joint state of `ROBOT_FORWARD` or `ROBOT_BACKWARD` from `getJointState()`

**Parameters**
None.

**Description**
This function is used to determine if a robot 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 robot 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 robot to Arch up for better ground clearance while rolling.

**CMobot::motionArchNB()**
This function causes the robot 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 robot to perform a single cycle of the inchworm gait to the left.

**CMobot::motionInchwormLeftNB()**
This function causes the robot 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 robot to perform a single cycle of the inchworm gait to the right.

**CMobot::motionInchwormRightNB()**
This function causes the robot 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 robot 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 robot 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 robot 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 robot 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 robot 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 robot 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 robot 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 robot 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 robot assume a skinny rolling profile.

**CMobot::motionSkinnyNB()**
This function makes the robot assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the robot 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 robot 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 robot to motionStand up into the camera platform.

**CMobot::motionStandNB()**
This function causes the robot 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 robot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

# CMobot::motionTumble()
# CMobot::motionTumbleNB()

**Synopsis**
```
#include <mobot.h>
int CMobot::motionTumble(int num);
int CMobot::motionTumbleNB(int num);
```

**Purpose**
Make the robot 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::motionTumble()**
This causes the robot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

**CMobot::motionTumbleNB()**
This causes the robot 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, `motionTumble()`, will block until the robot motion has completed. The non-blocking version, `motionTumbleNB()`, 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 robot 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 specifid by this argument in order to turn the robot to the left.

**Description**

**CMobot::motionTurnLeft()**
This function causes the robot to rotate the faceplates in opposite directions to cause the robot to rotate counter-clockwise.

**CMobot::motionTurnLeftNB()**
This function causes the robot to rotate the faceplates in opposite directions to cause the robot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnLeft()`, will block until the robot 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 robot 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 specifid by this argument in order to turn the robot to the right.

**Description**

**CMobot::motionTurnRight()**
This function causes the robot to rotate the faceplates in opposite directions to cause the robot to rotate clockwise.

**CMobot::motionTurnRightNB()**
This function causes the robot to rotate the faceplates in opposite directions to cause the robot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnRight()`, will block until the robot 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 robot 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 robot to move down from the camera platform.

**CMobot::motionUnstandNB()**
This function causes the robot 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 robot 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 robot by specified angles.

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

**Parameters**

| | |
|---|---|
| `angle1` | The amount to move joint 1, expressed in degrees, relative to the current position. |
| `angle2` | The amount to move joint 2, expressed in degrees, relative to the current position. |
| `angle3` | The amount to move joint 3, expressed in degrees, relative to the current position. |
| `angle4` | 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 robot by the specified number of degrees from their current positions.

**CMobot::moveNB()**
This function moves all of the joints of a robot 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 robot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the demo at Section 4.1.2 on page 19.

**See Also**

## CMobot::moveContinuousNB()

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

**Purpose**
Move the joints of a robot 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 |
|---|---|
| `ROBOT_NEUTRAL` | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| `ROBOT_FORWARD` | This value indicates that the joint is currently moving forward. |
| `ROBOT_BACKWARD` | This value indicates that the joint is currently moving backward. |
| `ROBOT_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.2 on page 66.

**Description**
This function causes joints of a robot 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( robotJointState_t dir1,
                                robotJointState_t dir2,
```

```
                                robotJointState_t dir3,
                                robotJointState_t dir4,
                                double seconds);
```

**Purpose**
Move the joints of a robot 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 |
|---|---|
| `ROBOT_NEUTRAL` | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| `ROBOT_FORWARD` | This value indicates that the joint is currently moving forward. |
| `ROBOT_BACKWARD` | This value indicates that the joint is currently moving backward. |
| `ROBOT_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 robot 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(robotJointId_t id, double angle);
int CMobot::moveJointNB(robotJointId_t id, double angle);
```

**Purpose**
Move a joint on the robot 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the example in Section 4.1.2 on page 19.

**See Also**
`connectWithAddress()`

---

# CMobot::moveJointTo()
# CMobot::moveJointToNB()

**Synopsis**
```
#include <mobot.h>
int CMobot::moveJointTo(robotJointId_t id, double angle);
int CMobot::moveJointToNB(robotJointId_t id, double angle);
```

**Purpose**
Move a joint on the robot 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the example in Section 4.1.2 on page 19.

**See Also**
`connectWithAddress()`

---

# CMobot::moveJointWait()

**Synopsis**
```
#include <mobot.h>
int CMobot::moveJointWait(robotJointId_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 robotic 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 4.1.2 on page 19.

**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 robot 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 robot to the specified absolute positions.

**CMobot::moveToNB()**
This function moves all of the joints of a robot 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the demo at Section 4.1.2 on page 19.

**See Also**

---

# CMobot::moveToZero()
# CMobot::moveToZeroNB()

**Synopsis**
```
#include <mobot.h>
int CMobot::moveToZero();
int CMobot::moveToZeroNB();
```

**Purpose**
Move all of the joints of a robot 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 robot to their zero position.

**CMobot::moveToZeroNB()**
This function moves all of the joints of a robot 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the demo at Section 4.1.2 on page 19.

**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 robotic 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 4.1.2 on page 19.
**See Also**
`moveWait(), moveJointWait()`

---

## CMobot::recordAngle()

**Synopsis**
```
#include <mobot.h>
int CMobot::recordAngle(robotJointId_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**

| | |
|---|---|
| `id` | The joint number. This is an enumerated type discussed in Section A.1 on page 66. |
| `time` | An array which will store time values for each of the angle readings. |
| `angle` | An array which will store the angle values for each time. |
| `num` | The size of the arrays. |
| `seconds` | 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 robot.

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::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**

| | |
|---|---|
| `time` | An array which will store time values for each of the angle readings. |
| `angle1` | An array which will store the angle values for joint 1. |
| `angle2` | An array which will store the angle values for joint 2. |
| `angle3` | An array which will store the angle values for joint 3. |
| `angle4` | An array which will store the angle values for joint 4. |
| `num` | The number of elements of the arrays. |
| `seconds` | 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::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::setJointSpeed()

**Synopsis**
```
#include <mobot.h>
int CMobot::setJointSpeed(robotJointId_t id, double speed);
```

**Purpose**
Set the speed of a joint on the robot.

**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 a robot. 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(robotJointId_t id, double ratio);
```

**Purpose**
Set the speed ratio settings of a joint on the robot.

**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.1 on page 66.
`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 robot.

**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 robot. 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 robot.

**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 robot.

**Example**

**See Also**
`getJointSpeeds()`, `setJointSpeed()`, `getJointSpeed()`

---

## CMobot::setTwoWheelRobotSpeed()

**Synopsis**
```
#include <mobot.h>
int CMobot::setTwoWheelRobotSpeed(double speed, double radius);
```

**Purpose**
Roll the robot 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 robot. The units used will be the units specified in the `unit` parameter.

`radius` The radius of the wheels attached to the robot. 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 robot 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::stop()

**Synopsis**
```
#include <mobot.h>
int CMobot::stop();
```

**Purpose**
Stop all current motions on the robot.

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

**Description**
This function stops all currently occuring movements on the robot. 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 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 robot, whereas the member functions of the `CMobotGroup` class move and affect a group of many robots.

Table 3: CMobotGroup Member Functions.

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

Table 4: CMobotGroup Member Functions for Compound Motions.

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

## CMobotGroup::addRobot()

**Synopsis**
```
#include <mobot.h>
int CMobotGroup::addRobot(CMobot &robot);
```

**Purpose**
Add a robot to a robot group.

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

**Parameters**
A robot handle attached to the robot to add to the group.

**Description**
This function is used to add a robot to a robot 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 robots 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 robots to Arch up for better ground clearance while rolling.

**CMobot::motionArch()**
This function causes the robots 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 robot 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 robots 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 robots to perform a single cycle of the inchworm gait to the left.

**CMobot::motionInchwormLeftNB()**
This function causes the robots 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 robots 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 robots to perform a single cycle of the inchworm gait to the right.

**CMobot::motionInchwormRightNB()**
This function causes the robots 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 robot 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 robots 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 robots backward.

**CMobot::motionRollBackwardNB()**
This function causes each of the faceplates to rotate 90 degrees to roll the robots backward.

This function has both a blocking and non-blocking version. The blocking version, `motionRollBackward()`, will block until the robot 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 robots 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 robots forward.

**CMobot::motionRollForwardNB()**
This function causes each of the faceplates to rotate 90 degrees to roll the robots forward.

This function has both a blocking and non-blocking version. The blocking version, `motionRollForward()`, will block until the robot 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 robots 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 robots assume a skinny rolling profile.

**CMobot::motionSkinnyNB()**
This function makes the robots assume a skinny rolling profile.

This function has both a blocking and non-blocking version. The blocking version, `motionSkinny()`, will block until the robot 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 robots 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 robots to stand up into the camera platform.

**CMobot::motionStandNB()**
This function causes the robots 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 robot motion has completed. The non-blocking version, `motionStandNB()`, will return immediately, and the motion will be performed asynchronously.

**See Also**

---

# CMobotGroup::motionTumble()
# CMobotGroup::motionTumbleNB()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::motionTumble(int num);
int CMobotGroup::motionTumbleNB(int num);
```

**Purpose**

Make the robots 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::motionTumble()**
This causes the robot to tumble end over end. The argument, `num`, indicates the number of times to tumble.

**CMobot::motionTumbleNB()**
This causes the robot 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, `motionTumble()`, will block until the robot motion has completed. The non-blocking version, `motionTumbleNB()`, 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 robots 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 specifid by this argument in order to turn the robot to the left.

**Description**
This function causes the robots to rotate the faceplates in opposite directions to cause the robot to rotate counter-clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnLeft()`, will block until the robot 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 robots 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 specifid by this argument in order to turn the robot to the right.

**Description**
This function causes the robots to rotate the faceplates in opposite directions to cause the robot to rotate clockwise.

This function has both a blocking and non-blocking version. The blocking version, `motionTurnRight()`, will block until the robot 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 robots 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 robot to move down from the camera platform.

**CMobot::motionUnstandNB()**
This function causes the robot 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 robot 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 robotic 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 6 for a list of all preprogrammed robotic motions.

**Example**

See the sample program in Section 4.1.2 on page 19.

**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 robots in a group by specified angles.

**Return Value**

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

**Parameters**

| | |
|---|---|
| `angle1` | The amount to move joint 1, expressed in degrees relative to the current position. |
| `angle2` | The amount to move joint 2, expressed in degrees relative to the current position. |
| `angle3` | The amount to move joint 3, expressed in degrees relative to the current position. |
| `angle4` | 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 robot by the specified number of degrees from their current positions.

**CMobot::moveNB()**
This function moves all of the joints of a robot 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 robot motion will occur asynchronously. For more information on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**

**See Also**

---

## CMobotGroup::moveContinuousNB()

**Synopsis**

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

**Purpose**
Move the joints of grouped robots 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 |
|---|---|
| ROBOT_NEUTRAL | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| ROBOT_FORWARD | This value indicates that the joint is currently moving forward. |
| ROBOT_BACKWARD | This value indicates that the joint is currently moving backward. |
| ROBOT_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.2 on page 66.

**Description**

This function causes joints of robots 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(robotJointState_t dir1,
                                    robotJointState_t dir2,
                                    robotJointState_t dir3,
                                    robotJointState_t dir4,
                                    double seconds);
```

**Purpose**

Move the joints of robots 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 |
|---|---|
| ROBOT_NEUTRAL | This value indicates that the joint is not moving and is not actuated. The joint is freely backdrivable. |
| ROBOT_FORWARD | This value indicates that the joint is currently moving forward. |
| ROBOT_BACKWARD | This value indicates that the joint is currently moving backward. |
| ROBOT_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 robots 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(robotJointId_t id, double angle);
int CMobotGroup::moveJointNB(robotJointId_t id, double angle);
```

**Purpose**

Move a joint on the robots 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**

Please see the example in Section 4.1.2 on page 19.

**See Also**

`connectWithAddress()`

## CMobotGroup::moveJointTo()
## CMobotGroup::moveJointToNB()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveJointTo(robotJointId_t id, double angle);
int CMobotGroup::moveJointToNB(robotJointId_t id, double angle);
```

**Purpose**

Move a joint on robots 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 robots 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 robots 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**

Please see the example in Section 4.1.2 on page 19.

**See Also**

## CMobotGroup::moveJointWait()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::moveJointWait(robotJointId_t id);
```

**Purpose**
Wait for a joint to stop moving on all robots 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 robotic 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 4.1.2 on page 19.

**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 robots 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 robots in the group to the specified absolute positions.

**CMobot::moveToNB()**
This function moves all of the joints of robots 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the demo at Section 4.1.2 on page 19.

**See Also**

---

# CMobotGroup::moveToZero()
# CMobotGroup::moveToZeroNB()

**Synopsis**
```
#include <mobot.h>
int CMobotGroup::moveToZero();
int CMobotGroup::moveToZeroNB();
```

**Purpose**
Move all of the joints of robots 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 robots 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 robots 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 robot motion will occur asynchronously. For more details on blocking and non-blocking functions, please refer to Section 8 on page 33.

**Example**
Please see the demo at Section 4.1.2 on page 19.

**See Also**

---

## CMobotGroup::moveWait()

**Synopsis**
```
#include <mobot.h>
int CMobotGroup::moveWait();
```

**Purpose**
Wait for all joints of all robots 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 robotic 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 4.1.2 on page 19.
**See Also**
`moveWait(), moveJointWait()`

---

## CMobotGroup::setJointSpeed()

**Synopsis**

```
\vspace{-8pt}
#include <mobot.h>
int CMobotGroup::setJointSpeed(robotJointId_t id, double speed);
```

**Purpose**
Set the speed of a joint on all robots 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 robots in the group.
**Example**

**See Also**

---

## CMobotGroup::setJointSpeedRatio()

**Synopsis**

```
#include <mobot.h>
int CMobotGroup::setJointSpeedRatio(robotJointId_t id, double ratio);
```

**Purpose**
Set the speed ratio settings of a joint on all robots 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.1 on page 66.
- `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 robots 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 robots 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 robot for all robots 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 robot 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 robots 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 robots 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 robot. The units used will be the units specified in the `unit` parameter.
- `radius` The radius of the wheels attached to the robot. 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 robot 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 robot in the group.

**Return Value**

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

**Description**

This function stops all currently occuring movements on the robot. 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()`

# D Miscellaneous Utility Functions

There are several utility functions which are useful when programming for the Mobot.

Table 5: Mobot Utility Functions.

| Function | Description |
|---|---|
| `angle2distance()` | Calculates the angle a wheel has turned from the radius and distance traveled. |
| `deg2rad()` | Converts degrees to radians. |
| `distance2angle()` | Calculates the distance traveled by a wheel from the wheel's radius and angle turned. |
| `rad2deg()` | Converts radians to degrees. |
| `shiftTime()` | 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 `double` 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()`

---

# 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**

| | |
|---|---|
| `radius` | The radius of the wheel. |
| `distance` | This value is the distance the wheel has traveled. This parameter may be of `double` 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()`

---

# 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 robot 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