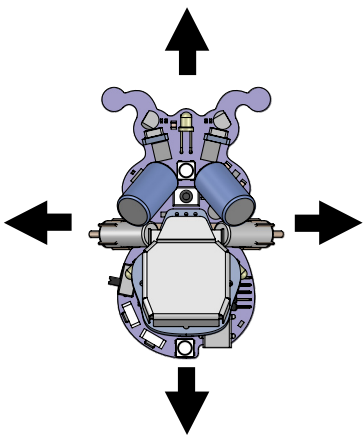
Using Ringo's Navigation

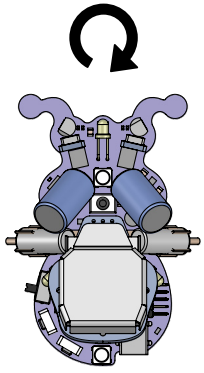
Ringo has two incredible sensors that can be used to sense his motion. The first sensor is called an Accelerometer. An Accelerometer senses movement in front-back, left-right, and up-down motion. It can be used to determine how far Ringo moves. The second sensor is called a Gyroscope. A Gyroscope senses rotational movement. For example, if you keep Ringo in the same location but you rotate him in a turn, the Gyroscope can be used to determine how many degrees he has rotated.

Taken together, these sensors embody an "inertial navigation" system, where Ringo can determine how far he has moved and how his orientation is changing. You could for example, make a 45 degree turn and move forward 1 meter. Awesome!

This is the first "Skill Level 2" lesson. We'll get a bit more advanced in the discussion here but have no fear - the actual functions used to move Ringo are fairly straightforward.



Accelerometer



Gyroscope

2
SKILL LEVEL

Using Ringo's Navigation

Background Information...

Let's start with some background information about how these sensors work. Don't let this part of the discussion scare you - this is good information to help you understand how and why the movement functions work the way they do.

The accelerometer doesn't actually tell you where Ringo is or how far he has moved. The sensor simply measures acceleration - which is how quickly Ringo's speed is changing at any given instant. It measures this 800 times per second, and stores the measurements in a memory inside the sensor. Every so often (about every 80 milliseconds in Ringo's case), Ringo's brain needs to establish a data connection with the sensor and unload these values. Becasue we know how much time elapsed between each measurement, Ringo can do some math and determine that a certain acceleration force will result in a certain speed, and that taken over time can determine distance. If the sensor data isn't read out every 80 milliseconds, the memory inside the sensor (called a "buffer") will "overflow". The overflow doesn't hurt anything, but it will forget the values it has already taken which means when Ringo eventually runs the math on those samples, the end result may not be accurate.

The gyroscope works in a similar way. It doesn't actually tell you what direction Ringo is facing. Instead, it takes snapshots of "angular velocity" at regular intervals timed very close together and stores those values in its own buffer. Ringo establishes a data connection to the gyroscope and unloads these values. Again, some math can be done on these values and determine that a given angular velocity over a given period of time will result in a predictable number of degrees of rotation and thus, Ringo can keep up on what direction he is facing.

Lucky for you, this process of reading values out of the sensors and performing the math on them (which is rather complex) has already been worked out and is performed automatically by calling a few simple functions.

Using Ringo's Navigation

Here is a quick list of the navigation functions along with a short description of each. These functions take care of the data communication between Ringo's brain and the sensors as well as performing the required math on the resulting data so they are easy to use.

Base setup functions:

`NavigationBegin();` This function turns on and initializes the gyroscope and accelerometer. It is important to call this function before running the other navigation functions. It can be called over and over with no harmful effects.

`CalibrateNavigationSensors();` This function calibrates the sensors. Even when the sensors are perfectly still, they do have a slight offset because no sensor is perfect. To calibrate the sensors, we measure some samples when they are still to see what this offset is. Ringo then remembers this offset and subtracts it out of future measurements. The NavigationBegin() function calls this CalibrateNavigationSensors(), so you don't need to call it a second time. It is important that Ringo is perfectly still when calling this CalibrateNavigationSensors(). Vibrations or wiggles on your table will adversly effect the calibration. The sensors should be recalibrated every so often. Keep in mind that it is useful to place a delay() before each calibration to make sure Ringo isn't still moving from a previous motor operation, etc. Wait a second or so to calibrate after the Reset or User buttons are pressed as the presense of your finger on the robot will adversly effect the calibration accuracy.

`ZeroNavigation();` Ringo keeps a memory of his X and Y location as well as his rotation. The X location is left/right of his starting location, with a positive number to the right, and a negative number to the left. The Y location is front/rear of his starting location with positive numbers in the forward direction and negative numbers in the rear direction. Units are in millimeters of travel. Rotation is in units of degrees. Positive degress are in the right (clockwise) direction and negative degrees are in left (counter-clockwise) direction. Note that the degrees do not loop. For example, if you rotate one and a half turns in the clockwise direction, the resulting rotation value will be 540. The ZeroNavigation() function zeros these running X, Y, and rotation numbers to zero.

`GetHeading();` The GetHeading() function returns Ringo's present heading in degrees.

Using Ringo's Navigation

Movement Functions:

Ringo's inertial navigation can be used in lots of interesting ways, and we encourage you to explore the possibilities. The most obviously useful function is in controlling Ringo's motors to move him in predictable ways. We have written a selection of functions that move Ringo's motors while reading the navigation sensors to go certain distances, turn in certain ways, etc. These functions handle running the motors including speed and direction, reading the sensors, performing the math on the sensor data, and adjusting motor speed to achieve the desired movement.

All movement functions use a "PID" software loop, which is a software technique commonly used to move robot arms. Basically, a PID loop causes the motors to move more quickly if they are further from their target, and more slowly as they reach the target. For example, when rotating, if Ringo needs to turn 180 degrees, the motors will run quickly at first and will slow progressivly as Ringo nears the 180 degree point. So the speed is naturally self correcting and minimizes over-shoot.

Rotation:

`DriveArc()` This function causes Ringo to drive forward (or backward) while performing a turning movement. It causes Ringo to drive in an "arc" type movement. The function takes five arguments and is prototyped as follows.

NOTE: `DriveArc()` requires use of Base_Sketch_04 or later.

```
void DriveArc(int TurnDegrees, int left, int right, \
                          int MaxExpectedTurnTime, int MaxExpectedSkidTime);
```

The terms work as follows. `TurnDegrees` is the number of degrees to be rotated during the turn. Positive or negative numbers can be used. Ringo will stop when either the `MaxExpectedTurnTime` expires, or the desired rotation is achieved during the arc movement. `left` and `right` are the speed for each motor. This controls how fast Ringo will move. Normally you'll use different numbers for left and right so Ringo will actually curve during movement. `MaxExpectedTurnTime` is the time (in milliseconds) Ringo will allow to reach his target rotation before giving up and moving on in the code. Without this feature Ringo would drive his motors as long as required until his sensors tell him that he has arrived at the target rotation. Because there is some rotational inertia to Ringo and his motors, he won't stop immediately. He will skid on the surface for some period of time. Once the motors are turned off, Ringo will continue to

Using Ringo's Navigation

evaluate his gyroscope and accelerometer and wait for the readings to indicate that he has in fact come to a complete stop. Similar to the MaxExpectedTurnTime, the MaxExpectedSkidTime is the maximum time Ringo will remain looking for a complete stop. This time can be fairly short as he will normally stop very quickly and move ahead to the next instruction on his own.

Lets consider a working example. This should cause Ringo to move in a star step movement arcing to the right then left then repeating.

```
#include "RingoHardware.h"  //include Ringo background functions

void setup(){
  HardwareBegin();     //initialize Ringo's circuitry
  PlayStartChirp();    //play startup chirp and blink eyes
  delay(1000);         //wait 1 second after restart before starting navigation
  NavigationBegin();   //initialize navigation system
}

void loop(){
             //rotation, left, right, turntime, skidtime
 DriveArc(90, 200, 75, 3000, 100);    // stop driving at 90 degrees
 delay(1000);                              // wait 1 second
              //rotation, left, right, turntime, skidtime
 DriveArc(-90, 75, 200, 3000, 100);  // this time go to the left
 delay(1000);                              // wait 1 second
}
```

In this example, we call NavigationBegin() one time in the setup function. In the main loop, we first tell Ringo to set motors to (200, 75) and stop them once he naturally achieves a 90 degree turn to the right. He will attempt to reach 90 degress for 3 seconds. If he cannot reach 90 in this time, he will skip out of the DriveArc() function and proceed to the next line of code. Once he stops, he will wait 1 second to reach a complete stop. These turn and skid times should be plenty to reach the target. The loop then makes the same movement in reverse and continues in a loop.

Hint: When you experement on your own, if you see Ringo consistantly fails to turn far enough, have a look at the MaxExpectedTurnTime argument and make sure you're allowing him enough time to complete the maneuver.

Using Ringo's Navigation

`RotateAccurate()` This function makes Ringo rotate in his own footprint. If Ringo over-shoots the target rotation, he will reverse direction and attempt to reach the target rotation more accurately. It is prototyped as follows.

```
char RotateAccurate(int Heading, int MaxExpectedTurnTime);
```

The `RotateAccurate()` function accepts two arguments: `Heading` and `MaxExpectedTurnTime`. `Heading` is the target rotation, and like the last example, the `MaxExpectedTurnTime` is the max time Ringo will attempt to complete the maneuver.

Note: The gyroscope on each Ringo unit is slightly more or less sensitive and thus, "RotateAccurate()" may over or under-shoot regularly. Open and load the sketch "Ringo_CalibrateGyroscope_Rev01". This sketch can be used to calibrate the gyroscope on your individual Ringo unit so the navigation functions are much more accurate.

Movement:

`MoveWithOptions()` This is the most basic function for movement. It accepts seven arguments and is prototyped as follows.

```
void MoveWithOptions(int Direction, int Speed, int Distance, \
        int MaxExpectedRunTime, int MaxExpectedSkidTime,    \
         void (*EdgeFunction)(char), char Wiggle);
```

The `MoveWithOptions()` function looks a bit intimidating, but it's actually pretty straighforward. Let's discuss it. The first argument is `Direction`. Ringo will attempt to turn to this direction as he begins driving. `Speed` is the average motor speed he should try to maintain. This will be adjusted up or down slightly between the two motors to produce the desired direction. `Distance` is the overall forward or backward travel distance in millimeters. As with the previous functions, `MaxExpectedRunTime` is the maximum time Ringo will allow to carry out the attempted maneuver, and `MaxExpectedSkidTime` is the maximum time Ringo will wait to arrive at a complete stop once he arrives at the commanded destination. The `(*EdgeFunction)(char)` term allows Ringo to call a certain function if he reaches an edge during his travel. That's a bit more advanced and we'll cover it later, but for now, just pass a `0` in this argument. The Wiggle argument controls how much "buggy swagger" Ringo

will show. A 0 in this argument will cause him to move fairly straight with only a minimal wiggle. A 50 in this argument looks pretty buggy - he sways around a lot more. It is suggested that a maximum value of 100 be passed for this argument. Anything more than 100 will cause him to have a real problem maintaining course, but it can be interesting to watch so go ahead and experement with it.

Lets have a look at a working example of this function.

```
#include "RingoHardware.h"  //include Ringo background functions

void setup(){
  HardwareBegin();    //initialize Ringo's circuitry
  PlayStartChirp();   //play startup chirp and blink eyes
  delay(1000);        //wait 1 second after restart before starting navigation
  NavigationBegin();  //initialize navigation system
}

void loop(){
             //direction, speed, distance, runtime, skidtime, edge, wiggle
 MoveWithOptions(0, 220, 300, 2000, 100, 0, 10);  // go forward 300 mm
 delay(1000);                                     // wait 1 second
 MoveWithOptions(0, -220, -300, 2000, 100, 0, 10);  // go backward 300 mm
 delay(1000);                                     // wait 1 second
}
```

In this example, we call MoveWithOptions() and tell it to drive Ringo in the 0 degree heading (straight ahead) at a motor speed of 220 (motor speeds can be positive or negative 0 to 255) for a distance of about 300 mm. He will allow 2 seconds to complete the maneuver, allow 100 milliseconds to come to a complete stop after the maneuver, and he'll do it all with level 10 buggy swagger. He then does the same in reverse. To drive in reverse you'll specify a negaive motor speed (-200 in this case) as well as a negative distance (-300 in this case).

Now lets combine movement and rotation. Like this:

Using Ringo's Navigation

```
void loop(){
             //direction, speed, distance, runtime, skidtime, edge, wiggle
 MoveWithOptions(GetHeading(), 220, 300, 2000, 100, 0, 10);
                                    // go forward 300 mm at present heading
 delay(500);                         // wait 1/2 second
 RotateAccurate(90, 2000);           // rotate 90 degrees clockwise
 delay(500);                        // wait 1/2 second
}
```

In this example, we've made a change to the `MoveWithOptions()` function. You may notice that instead of going direction `0`, we instead have plugged another function into this spot - `GetHeading()`. Lets talk about this for second.

Lets say we put a `0` instead of `GetHeading()`. You may want to try this and see what happens on your own. When we put a `0` in this function, Ringo is going to try and drive to the heading of `0` referenced from the last time you called either `HardwareBegin()` or `ZeroNavigation()`. Putting a `0` in here the first time through the loop causes him to drive straight as expected. But after we rotate him 90 degrees, he is no longer headed in the `0` degree direction. If we call `MoveWithOptions()` a second time after the rotation, and we put a `0` for direction, Ringo will make a left turn as he starts to drive because he is trying to return to the `0` degree direction.

An easy way to avoid this, is to always place `GetHeading()` in place of the zero. The `GetHeading()` function reads Ringo's current heading and automatically passes this current heading to the `MoveWithOptions()` function each time it is called. By passing `GetHeading()` for the direction, Ringo will always travel in a straight line each time the function is called, no matter how much you rotate him.

You could do other cool things like passing "`GetHeading()+15`" which would cause Ringo to make a 15 degree clockwise turn during each movement.

These are the basics to get you started. We will continue to fill out this guide with additional examples and greater explanation as we move forward. Cheers!!