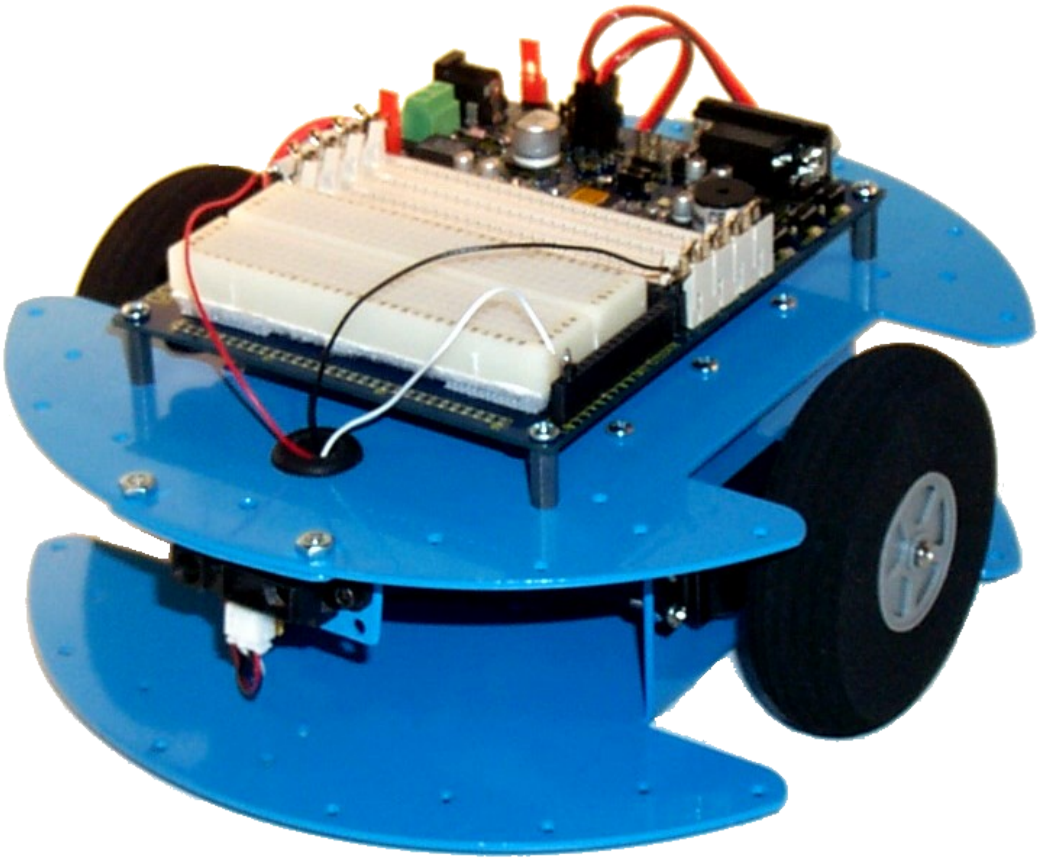
ROGUE robotics

ERS

Educational Robotic System



www.roguerobotics.com

Rogue Blue ERS

Student Guide

Version 1.0

January 2004

Table of Contents

Preface

When we started Rogue Robotics, we wanted to make robotics more accessible to everyone. One hurdle we encountered was the complexity of programming required to make a robot do something interesting. Then we discovered the OOPIC microcontroller from Savage Innovations.

The OOPIC microcontroller has several features that make it ideal for robotics and education in the 21st century. It is an object-oriented processor that can be programmed in either Basic (Visual Basic™ compatible), C++ or Java™ syntax. Many universities and colleges start their first year programs with object-oriented classes in Java or similar.

You can also implement hardware objects into what is called “Virtual circuits” which operate extremely fast and multitask in the background. Robotics is a real-time world and multitasking event-driven programming is a must for good robotics.

Our OOBoard™, which is the brain for our robot, mounts an OOPIC microcontroller at its core as well as various I/O, interface and expansion ports.

This curriculum is intended for Grades 11 and 12 as a supplement to courses where programming and computer interfacing are required. The experiments in this guide are a mixture of programming and microcontroller interfacing, but the emphasis is on programming.

For this student guide, we have chosen to write all the code in the Java syntax. Code is also available in BASIC syntax. C++ syntax is very similar to Java syntax.

This is the first release of this curriculum and we will be constantly improving the content. If you have any suggestions or concerns, please contact us. We appreciate all feedback.

We would like to thank all the reviewers past, and future for their input and revisions, especially the teachers willing to put our initial curriculum in front of their students.

Chapter 1 - Getting Started

The Rogue Blue ERS is a mobile robot, so you probably would like to see it move sooner than later. So, let's get started as quickly as possible.

First you will need a quick introduction to the robot and the software needed to program it.

***If your robot has been used in class before, it may have a program already in memory, which means that it may move when you turn on the power. So, before you start programming, it is a good idea to put the robot on its "experiment parts box" or a similar block to raise the wheels off the ground.

Assemble and configure the robot

Assemble the robot following the instructions in Appendix C: Robot Assembly and Software installation.

Please install the sensor mount and the IR ranger at this time as well and place the removable breadboard on the OOBoard.

When finished your robot should look the same as Figure 1.1.

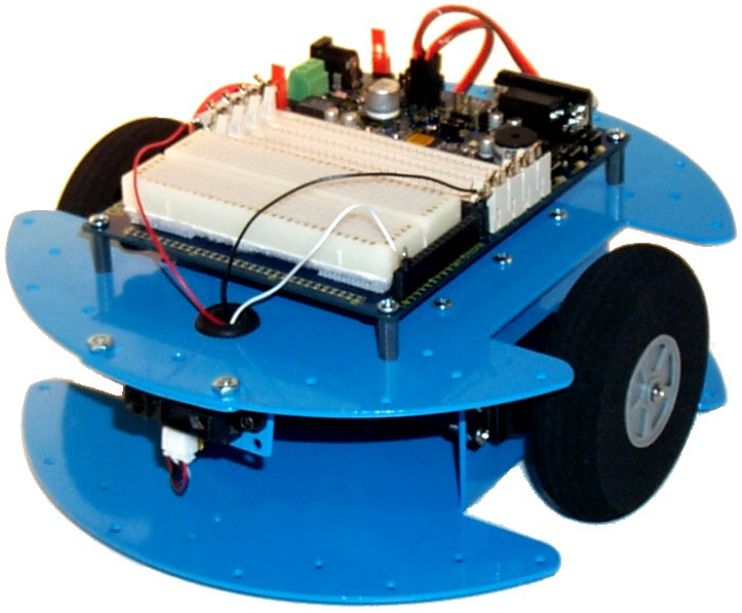


Figure 1.1 - Rogue Blue ERS Assembled

When you are installing the battery wires, double check that you have the black wire closest to the serial (DB9) connector.

When you connect the servo cables (the motors for the wheels), make sure they are connected as in Figure 1.2. The left wheel (servo) should be connected to the connector marked “Servo1”, and the right wheel should be connected to the connector marked “Servo2”.

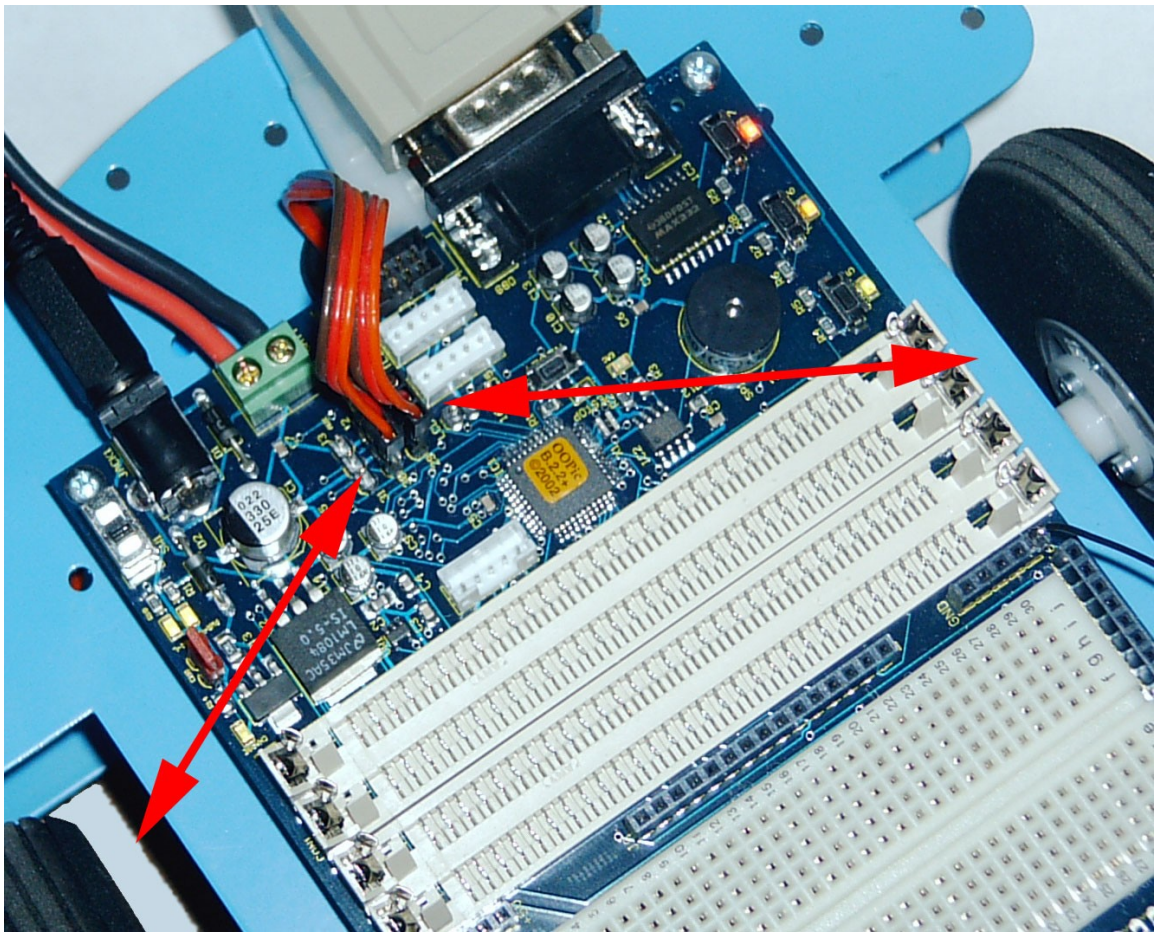


Figure 1.2 - Wheel (Servo) Connections

Introduction to the program editor

First, you will need to install the program editor if it isn't already installed. Please refer to Appendix C: Robot Assembly and Software Installation for instructions on this installation.
Once you have the software installed, click on the OOPIC desktop icon, or **START|ALL PROGRAMS|OOPIC|OOPIC** to launch the program editor.

When it launches it should look like this:

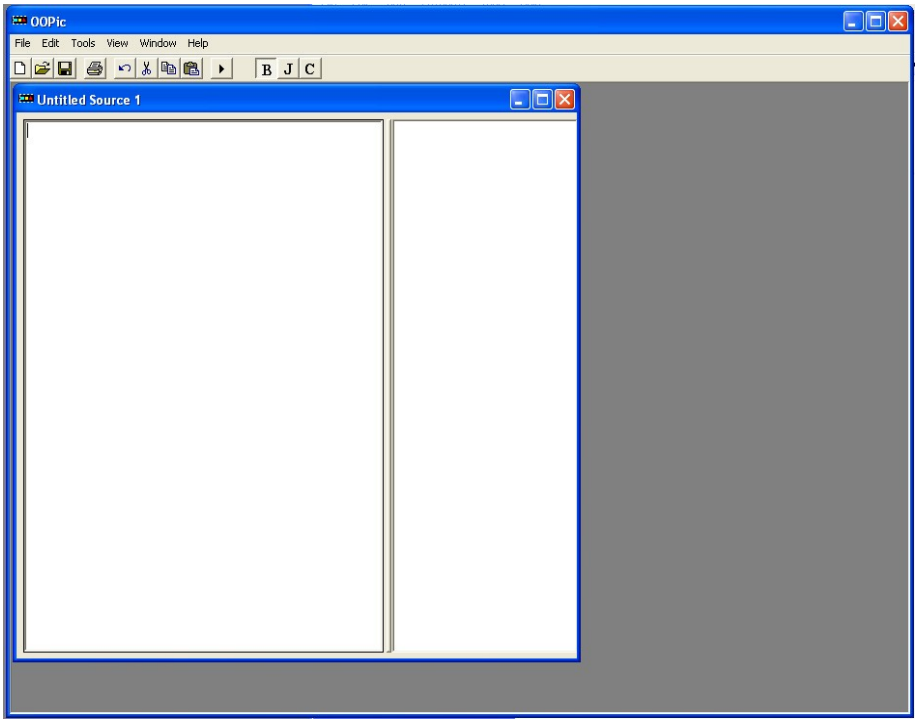


Figure 1.3 – OOPIC Program Editor

Take a closer look at the toolbar:

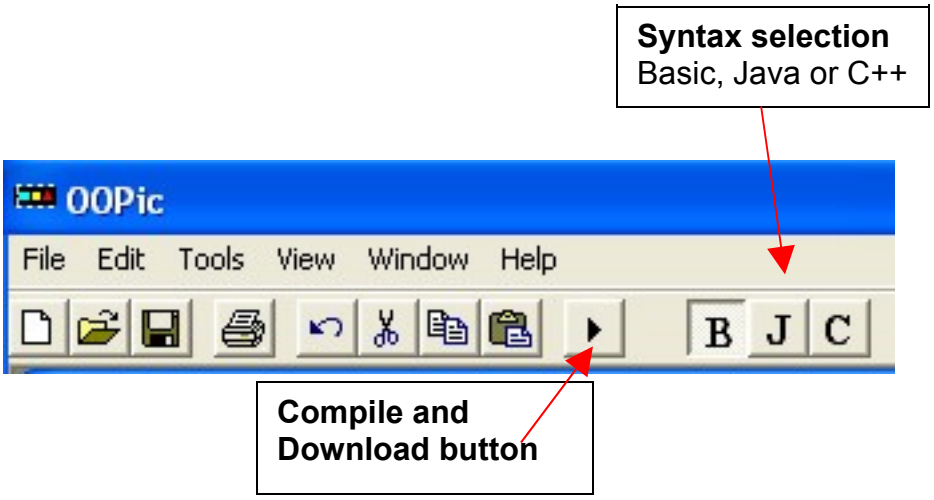


Figure 1.4 – OOPIC Program Editor Toolbar

Choose **Java** from the Syntax selection buttons. Make sure that you always use Java for the code examples from this file.

Before we go too far, click on **Help|Manual.** This will bring up the online help for the program editor in html format in your web browser.

In the manual, you will find a programmer's guide, a language guide, which explains how to implement the different syntaxes and differences, a detail list of objects, OOPIC connectors and mechanical. If you click on the OOPIC Objects link you will get a list of versions - the version used on the OOBoard is the B2.x+ version, so click on that link. You will get a list of hardware, processing, variable, user-definable and system objects that you can use to program the OOBoard with. You will learn more about objects in chapter 3.

Review the OOBoard

The OOBoard is the central brain for our robot. The following picture outlines various features of the board.

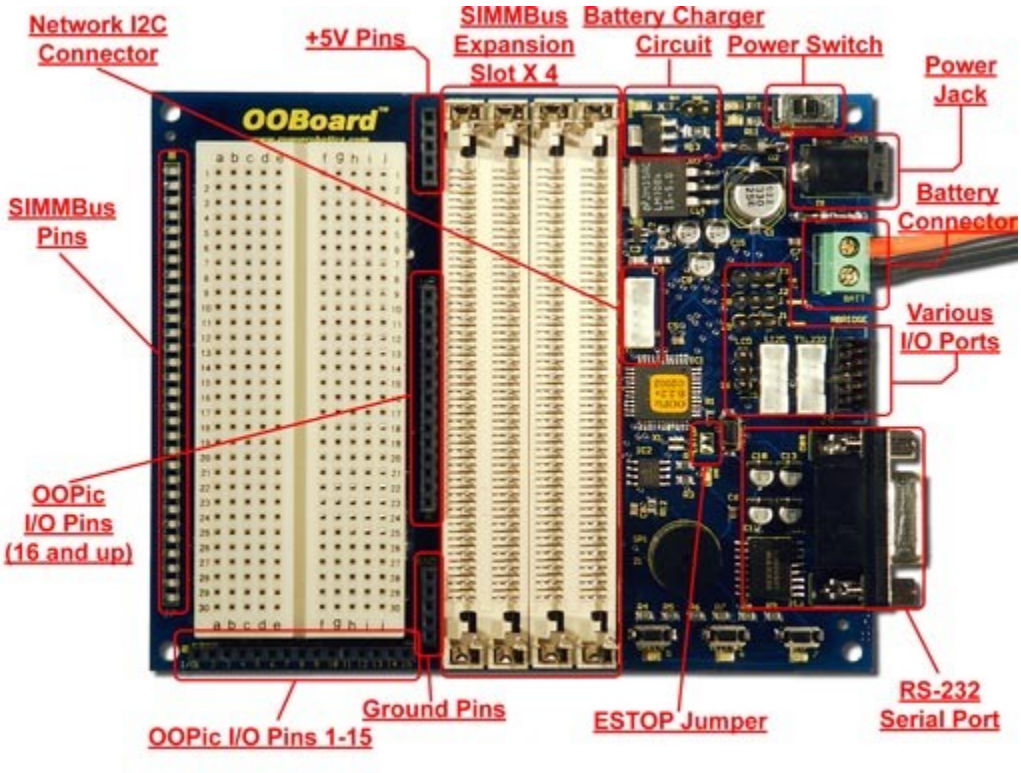


Figure 1.5 – OOBoard Features

For more information on the function and how to use these features and connectors, use the help manual in the program editor and select **OOPIC Connectors and Mechanical|OOBoard.**

Power switch settings

Power ON – Battery | Power Off | Power ON – Wall Supply

Will still charge if wall supply Is plugged in, unplug to stop charging.

Figure 1.6 – Power Settings

Serial Cable

The serial cable plugs into the OOBoard on the robot and the opposite end plugs into your computer. If your computer does not have a DB9 serial connector and only USB, you will need a USB to serial adapter (available separately).

Wall Power Supply

The wall power supply plugs into a standard 120V 60Hz socket and the small barrel connector plugs into the power jack indicated on figure 1.4.

The Wall power supply is 12VDC, rated at 300mA. The power jack is 2.1mm center positive.

Removable Breadboard

The Rogue Blue ERS comes with a removable breadboard. You can change this breadboard easily to allow another experiment to be fitted or another student to run his or her own experiment.

Now that you have an understanding of the robot and tools, we can get started with our first experiment.

Experiment 1-1: Give it a heartbeat

Let's program our robot and indicate it's alive.

In this experiment, you will turn a light (red LED – or Light Emitting Diode) on and off like a pulse or heartbeat.

For this chapter, we will just enter in the program without explanation. You will learn more about the programming and interfacing in the next two chapters.

Select **file|new** (quick key **Ctrl-N**) in the OOPIC Program Editor. Next enter in the following program exactly as typed below.

Program Listing:

```
oDIO1 LED = New oDIO1;

Sub Void Main(Void)
{
  LED.IOLine = 7;
  LED.Direction = cvOutput;

  While (cvTrue)
  {
    LED.Value = 1;
    ooPIC.Delay = 50;
    LED.Value = 0;
    ooPIC.Delay = 50;
  }
}
```

Double-check your typing for spelling mistakes or typos. Click **File|Save As** in the OOPIC Program Editor. Select your preferred directory to save in (My Programs) and call this program "EX1-1". The .osc suffix will be added automatically when you save.

Now you are ready to download the program from your PC to the robot. Check to ensure the programming cable in connected and the robot's power is ON. Then click the **Arrow** on the menu bar or select **FILE|MAKE & DOWNLOAD** (quick key **F5**). This will compile the program into the language the brain on the robot understands and send it over the programming cable to the robot.

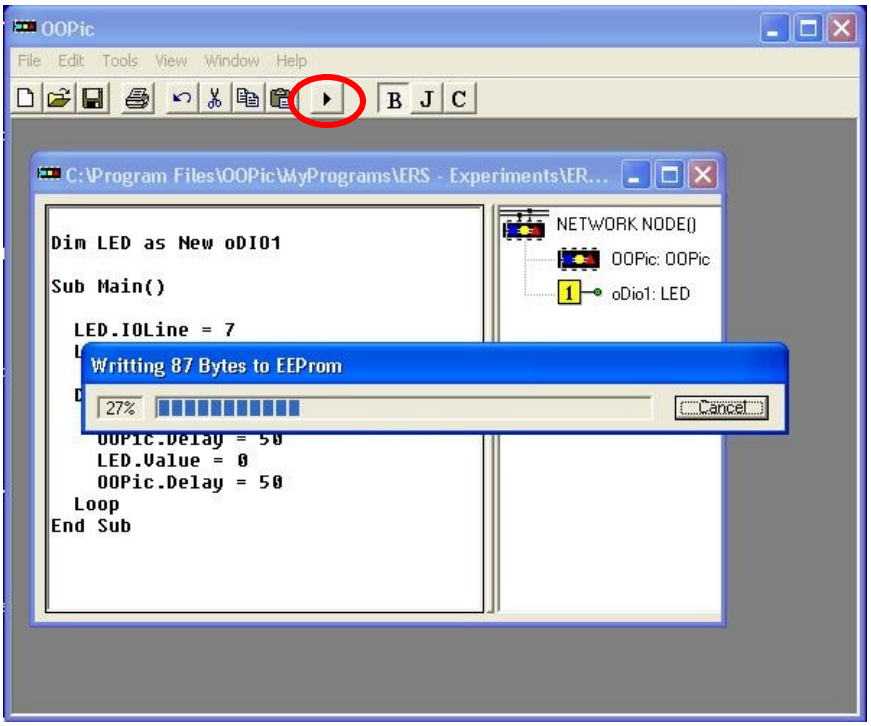


Figure 1.7 - Downloading

If you didn't make any typing errors, then you should see the RED light (LED) blinking on and off once every second. Your robot is alive and its heart is beating.

Experiment 1-2: Make it move

Now let's make it move, after all it is a mobile robot.

But first a tip, take out your "experiment parts kit" snap case and place it between the wheels underneath the robot so when we first program and test the robot it doesn't run away across your desktop.

Program Listing:

```
oButton GoButton = New oButton;
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;
oWire WireL = New oWire;
oWire WireR = New oWire;

Sub Void Main(Void)
{
  GoButton.IOLine = 6;
  LeftServo.IOLine =29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = 127;
  RightServo.Value = 127;

  WireL.Input.Link(GoButton.Value);
  WireR.Input.Link(GoButton.Value);
  WireL.Output.Link(LeftServo.Operate);
  WireR.Output.Link(RightServo.Operate);
  WireL.Operate = cvTrue;
  WireR.Operate = cvTrue;
}
```

When you have entered and double-checked the program. Check to ensure the programming cable is connected and the robot's power is ON. Then click the **Arrow** on the menu bar or select **FILE|MAKE & DOWNLOAD** (quick key **F5**).

Now if you press the button labeled "Yellow", the robot's wheels should both move forward at full speed. If everything seems fine, you can turn off the robot, take the programming cable off, unplug the wall power supply, and then put the robot on the floor and turn it back on. Press the button again and it will move forward. Congratulations, you have made your robot move. Now, let's find out how it all works.

Chapter 2 - About Microcontrollers

You may or may not already know that microcontrollers are all around us. We use them everyday, in fact. When you use your microwave to reheat a meal, or when you pick up the phone to call a friend, you are actually using a microcontroller.

Microcontrollers take input, such as button presses, temperature readings, or light levels, and process the information. It will then use the information to make decisions on what to do next, such as start a timer, turn off a heating element, or activate a motor.

The OOBoard uses a powerful microcontroller called an OOPIC, which will allow you to do a lot of things for your robot. It is very simple to use.

What is a microcontroller?

A microcontroller is an electronic device that processes electrical signals. It will take input, and produce output. What makes each microcontroller unique is how each one is programmed.

Microcontrollers think in binary, or 1's and 0's. A 1 is **5 Volts**, and a 0 is **0 Volts** or **Ground**.

Binary numbers have only 2 possible values for each digit of the number. A **bit** is a single digit of a binary number. A **byte** is made up of 8 bits.

For example, the decimal number **25** is represented in binary as **11001**.

Microcontrollers perform their tasks by being given a series of binary messages, or instructions. We create these instructions by using a **programming language** such as BASIC, C, or Java. For the OOPIC, we create our program in that language, and then it is translated, or **compiled**, into binary messages that the OOPIC microcontroller will understand.

Experiment 2-1a: Basic Output

One of the most important things a robot can do is interact with its environment. The most basic form of interaction is called **digital input** or **digital output** (often referred to as **Digital I/O**). The Rogue Blue ERS OOBoard has 15 easily accessible I/O connections, or **I/O Lines**.

The first type of interaction we will do is making the OOBoard turn on a light.

The light we will use is called an LED, or **L**ight **E**mitting **D**iode.

Although we already have 3 LEDs already on the OOBoard, it is important to know that we can use any I/O Line on the OOBoard to control an LED.

Parts you will need:
1 LED (connected to I/O Line 1)
1 Built-in "Yellow" pushbutton (I/O Line 6) on OOBoard
1 470-Ohm Resistor (Yellow Purple Brown Gold)
Wire

Build it:

Here is a schematic showing you how to connect the parts together for this experiment. For more information on bread boarding rules, see Appendix A.

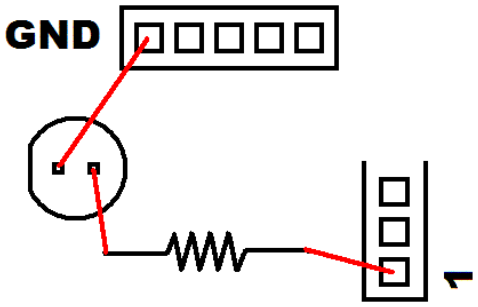


Figure 2.1 – Experiment 2-1a Schematic

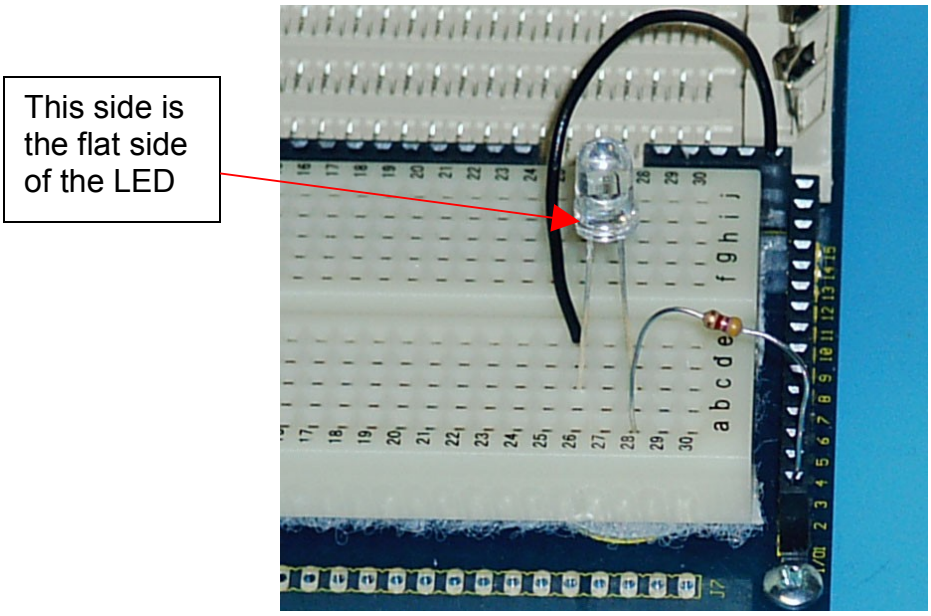


Figure 2.2 – Picture of connections

Type-in the program as listed below.

Program Listing:

```
// Experiment 2-1:  Basic Output

// Single bit Digital Input/Output
oDIO1 LED = New oDIO1;

Sub Void Main(Void)
{
  // Set the OOPIC I/O line for the LED
  LED.IOLine = 1;

  // Set the direction as an output
  LED.Direction = cvOutput;

  // Let there be light!
  LED.Value = 1;
}
```

Walk through the program:

We need to be able to control the voltage that goes to the LED. As you already know, a binary 1 will produce 5 Volts, while a 0 will produce 0 Volts. We use a digital output to control the voltage at the LED.

Programs use variables to change information. The way we create or **declare** variables in the OOPIC Java programming language, is by using the variable type. Each variable has a type.

Our variable named "LED" will allow us to control the voltage at the LED, so we will use a variable type called "oDIO1".

```
oDIO1 LED = New oDIO1;
```

All of our program statements must go between the "Sub Void Main(Void) {" and "}" statements. The OOPIC will start executing the first statement right after the "{" statement. You can think of the "{ " as 'begin' and "}" as 'end.'

```
Sub Void Main(Void)
{

}
```

Now, in between there, we put our program. All we want to do is to turn on the LED, so we will have to tell the OOPIC where the LED is. So we change the information for the LED variable.

```
  ' Set the OOPIC I/O line for the LED
  LED.IOLine = 1;
```

Then we need to tell the OOPIC that we want to **output** a value:

```
  ' Set the direction as an output
  LED.Direction = cvOutput;
```

And finally, we turn on the light:

```
  ' Let there be light!
  LED.Value = 1;
```

Experiment 2-1b: Basic Input

Now, that last experiment was probably not all that interesting. The next one will probably not that much more exciting, but we need to demonstrate the two most important tasks that a microcontroller can perform: input and output.

In the previous experiment, we made the OOPIC **output** a value to control a light. Now we will program the OOPIC to **input** a value using a button. Specifically, we will turn on the light when the button is pressed.

Parts you will need:

(The same parts as the last experiment)

Enter the program as listed below.

Program Listing:

```
// Single bit Digital Input/Output
oDIO1 LED = New oDIO1;

oDIO1 Button = New oDIO1;

Sub Void Main(Void)
{
  LED.IOLine = 1;
  LED.Direction = cvOutput;

  Button.IOLine = 6;
  Button.Direction = cvInput;

  While (cvTrue)
  {
    If (Button.Value == 1)
      LED.Value = 1;
    Else
      LED.Value = 0;
  }
}
```

Walk through the program:

You will see the similarities between this program and the last experiment. In this experiment though, we will use another variable, “Button” to get the **input** from the pushbutton (the “Yellow” or middle pushbutton on the OOBoard).

Once again, we need to tell the OOPIC where the button is, and that it is an **input**:

```
Button.IOLine = 6;
Button.Direction = cvInput;
```

Now we will continuously check the pushbutton to see if it has been pressed, and if it has, we will turn on the light. As well, if the button has been released, we will turn the light off:

```
While (cvTrue)
{
  If (Button.Value == 1)
    LED.Value = 1;
  Else
    LED.Value = 0;
}
```

That’s it! The statements within the “While (cvTrue) { }” are done continuously until the OOBoard is powered off, or reset.

Experiment 2-2: Measuring an Input: Sensing Light

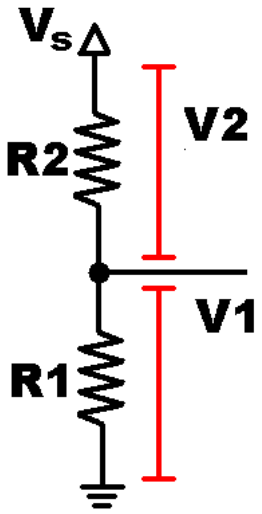
Digital I/O is the most basic way for a microcontroller to interact with the outside world, but there are other ways in which a microcontroller can get and send information.

One other way is called **analog to digital conversion** (or A2D). Simply put, A2D is representing a voltage as a number. For example, if 5 Volts is represented as 100, and 0 Volts is represented as 0, 3 Volts would be represented as 60.

The OOPIC has seven A2D converters (only on I/O lines 1 through 7). Each one can read a voltage and convert it to a number (usually a value between 0 and 255).

A useful input for a microcontroller is how much light is in a room. We can use what is called a **photo-sensor or photo-resistor** to measure how bright or dark an area is. We will connect the photo-resistor in a voltage-divider fashion so that we can use an A2D converter on the OOPIC to measure the level of light.

A Voltage Divider



It may be obvious, but a voltage divider just does what it says: it divides a voltage.

Whatever resistors you use, the voltage across each resistor (voltage drop), V1 and V2 must add up to the supply voltage, Vs.

The voltage V1 is calculated as follows:

V1 = Vs X R1 / (R1 + R2)

Let's say our supply voltage, Vs, is 5 volts, the bottom resistor, R1, is 3 Ohms, and the top resistor, R2, is 12 Ohms. The voltage at V1 would be 5 X 3 / 15 = 1 volt. Now if we changed R2 to 2 Ohms, V1 would be 5 X 3 / 5 = 3 volts.

Figure 2.3 – Voltage Divider

You can see that if we replace R2 with the photo-resistor (the resistance varies between 100 and 5000 Ohms, depending on the amount of light it gets), and use a 1000 (or 1K) Ohm resistor for R1, we can see that the voltage at V1 will be

larger when there is more light shining on the photo-resistor, and a lower voltage when it is dark.

This experiment will make the OOPIC monitor the value from the photo-sensor and turn on an LED when it has fallen below a certain value.

When the photo-sensor has a lot of light shining on it, the resistance is very low, which causes a higher voltage at the input pin. This higher voltage is converted, using the A2D converter, to a high number (around 120 to 200).

When we cast a shadow over the photo-sensor, the resistance is very high, which causes a lower voltage at the input pin, and hence a lower value from the A2D converter (around 10 to 30).

Using this, we can make a program to check the value to see if it is in that lower range and make an LED turn on.

Parts you will need:

Built-in Red LED (I/O Line 7) on OOBoard
1 5K Ohm photo-resistor (connected to I/O line 2)
1 1K Ohm resistor
Wire

Build it:

Here is a schematic showing you how to connect the parts together for this experiment.

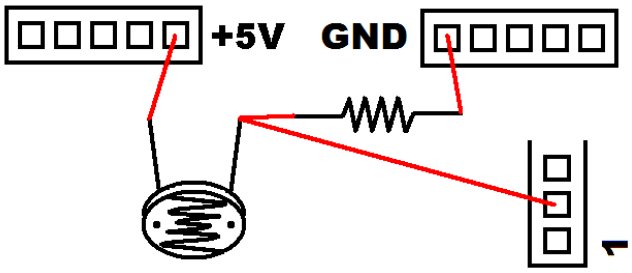


Figure 2.4 – Schematic for Experiment 2-2

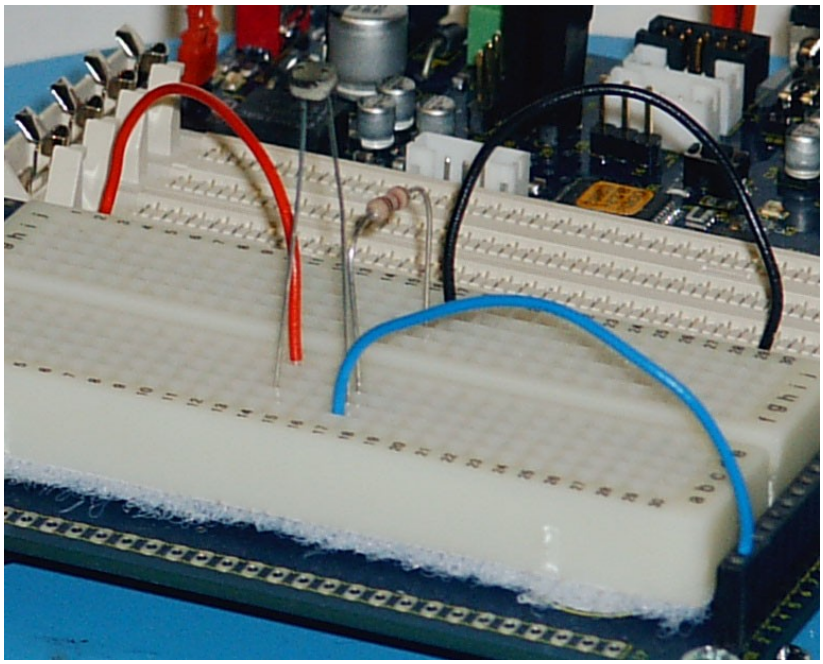


Figure 2.5 – Picture of connections

Enter the program as listed below.

Program Listing:

```
oDIO1 DarkLED = New oDIO1;

oA2D LightSensor = New oA2D;

Final DarkLevel = 60;

Sub Void Main(Void)
{
  LightSensor.IOLine = 2;
  LightSensor.Operate = 1;

  DarkLED.IOLine = 7;
  DarkLED.Direction = cvOutput;

  While (cvTrue)
  {
    If (LightSensor.Value < DarkLevel)
      DarkLED.Value = 1;
    Else
      DarkLED.Value = 0;
  }
}
```

Walk through the program:

As in the other programs we need to tell the OOPIC where it should use an A2D converter to measure the photo-sensor voltage.

```
LightSensor.IOLine = 2;
LightSensor.Operate = 1;
```

In the main loop, we simply check the value from the A2D conversion to see if it has fallen below a certain level (DarkLevel)

```
If (LightSensor.Value < DarkLevel)
  DarkLED.Value = 1;
Else
  DarkLED.Value = 0;
```

DarkLevel is what is called a **Final**. It is a value is entered by the programmer, but cannot be changed by the program. Now, you are probably asking, "How can I know what value to use?"

```
Final DarkLevel = 60;
```

In the above program, we have used the value of 60 as our DarkLevel. When the light sensor is in a room without any bright lights shining directly on it, or any shadows casting on it, it is in what is called **ambient** lighting. We need to find the value from the light sensor in this condition, and then we can figure out what level to use for our DarkLevel.

Here is where we can use the OOPIC program editor to help us see what value to use.

After we compile and download our program to the OOPIC, the program editor displays all of our "objects" that we are using in the program on the right hand side of the editor window. Keep the serial cable connected to the OOBoard and the power on.

If we **single-click** the "LightSensor" object, a window will pop up with some information about the light sensor. The value we are interested in is the Value property. If the lighting conditions are as we described earlier (no bright lights or shadows), then we can use that value (if you need to move the robot around to get the right lighting conditions, you can press the Refresh button to give you a new reading). This is our ambient value.

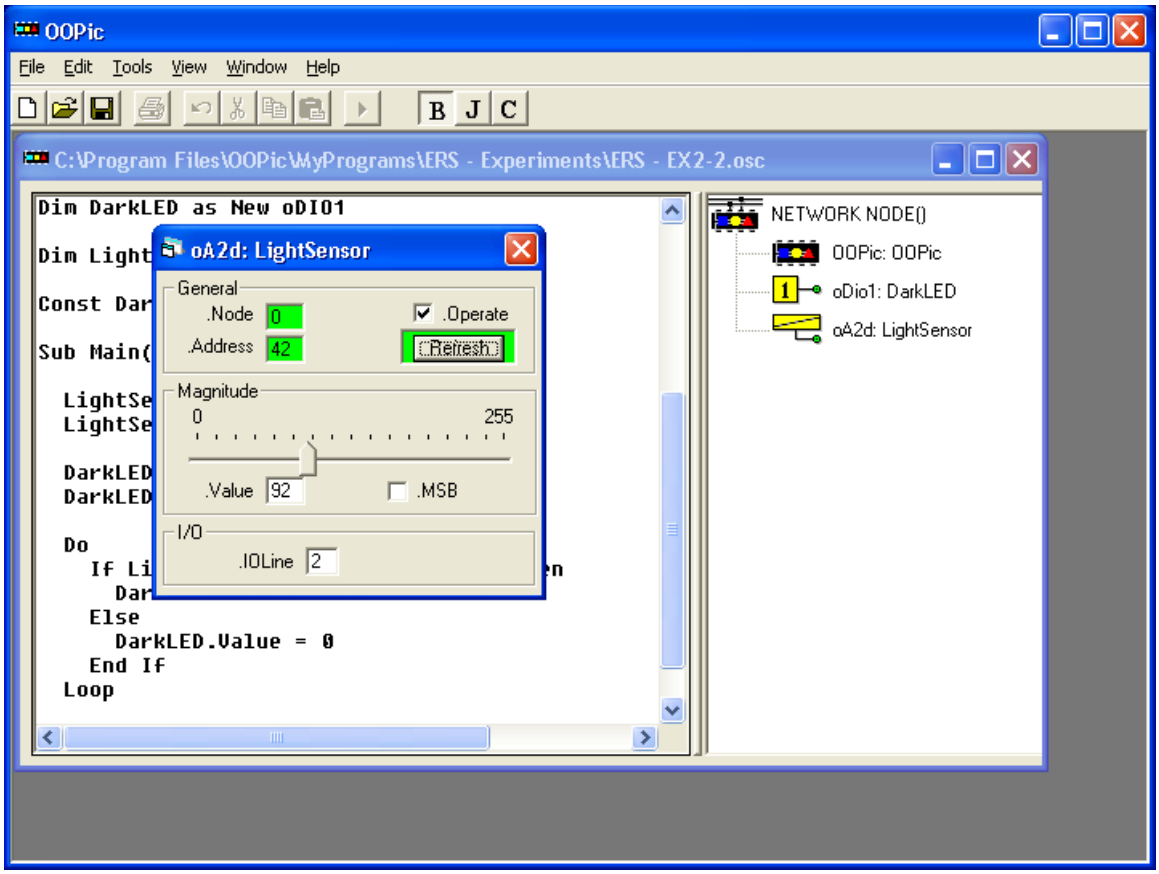


Figure 2.5 – Using the Object Viewer

Now that we have an ambient value for the light in the room let's subtract a value, say 30, and use this value for our DarkLevel (you can test this as well, by putting your hand over the sensor and reading the values – the sensor should read something well below this value to indicate it is dark). For example, if our ambient value for the room lighting was about 90, if we subtract 30, we get 60 for our DarkLevel. So when the light sensor value falls below 60, then we know that a shadow has been cast over the sensor.

Experiment 2-3: A Robot Reflex

A mobile robot interacts with the world in much the same way we do. There are some instances where you would want a robot to have a reflexive action to avoid a dangerous situation.
To demonstrate a simple reflex action, we will have our robot run away when it senses a shadow perhaps indicating a imminent foot coming down on it, after all it is small. We could call this a dangerous situation for the robot, so it will run away 'reflexively'.

We will use the photo-sensor and the wheels to demonstrate a simple robot reflex.

The wheels of the robot are controlled by servos. Using an **oServoX** object, the value 127 makes the wheel go as fast as it can in forward direction, and the value –127 will make it go as fast in the backward direction.

Since they are mounted in opposite directions, we need to make one servo go backwards, so we use the InvertOut property of the oServoX object.

Parts you will need:

Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
Built-in Red LED (I/O Line 7) on OOBoard
1 5K Ohm photo-resistor (connected to I/O line 2)
1 1K Ohm resistor (Brown Black Red Gold)
Wire

Build it:

Here is a schematic showing you how to connect the parts together for this experiment.

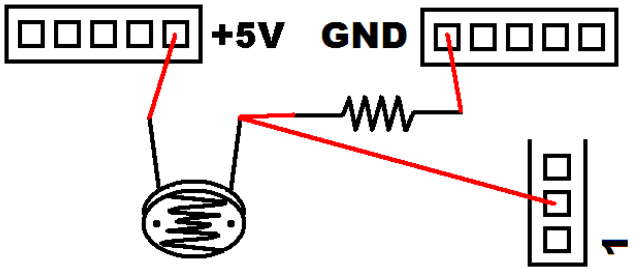


Figure 2.6 – Schematic for experiment 2-3

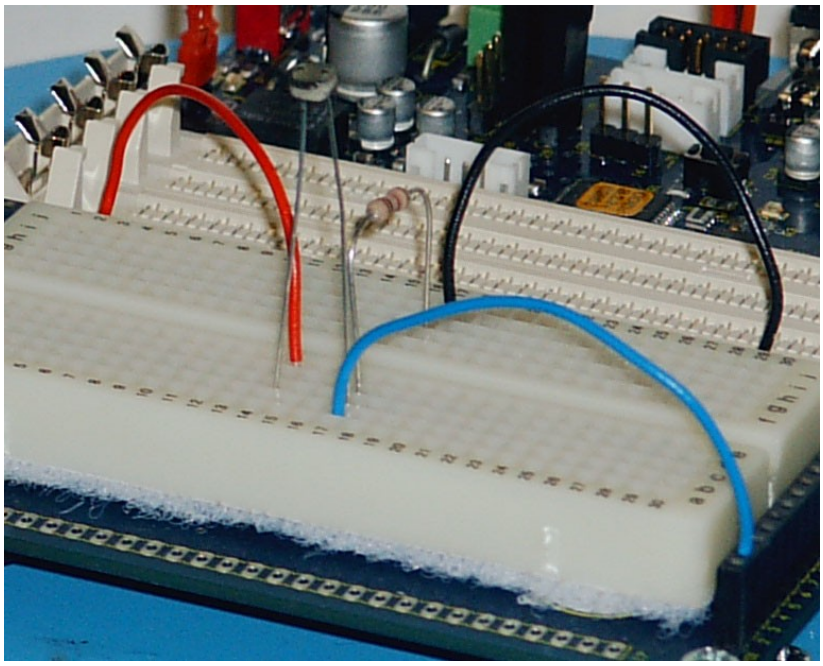


Figure 2.7 – Picture of connections

Enter the program as listed below.

Program Listing:

```
oDIO1 DarkLED = New oDIO1;

oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oA2D LightSensor = New oA2D;

Final DarkLevel = 60;

Sub Void Main(Void)
{
  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = -127;
  RightServo.Value = -127;

  LightSensor.IOLine = 2;
  LightSensor.Operate = 1;

  DarkLED.IOLine = 7;
  DarkLED.Direction = cvOutput;
```

```
While (cvTrue)
  {
    If (LightSensor.Value < DarkLevel)
    {
      DarkLED.Value = 1;
      LeftServo.Operate = cvTrue;
      RightServo.Operate = cvTrue;
    }
    Else
    {
      DarkLED.Value = 0;
      LeftServo.Operate = cvFalse;
      RightServo.Operate = cvFalse;
    }
  }
}
```

Walk through the program:

The wheels of the robot are controlled by servos. They are connected on lines 29 (Left) and 30 (Right). We invert the right servo because it is mounted opposite of the left servo (which goes forward with a positive value).

```
  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;
```

We will set the initial values for the servos to make the wheels go as fast as they can backwards (to run away from the shadow).

```
  LeftServo.Value = -127;
  RightServo.Value = -127;
```

Now, in the main loop, we will check to see if the light level has fallen below our DarkLevel value, and if it has, we will turn the servos on and make the robot move backward. Likewise, if the value of the sensor is above our DarkValue, we will stop moving. If you haven't found the value for DarkValue yet, you should use the object viewer in the programming editor to find it.

```
    If (LightSensor.Value < DarkLevel)
    {
      DarkLED.Value = 1;
      LeftServo.Operate = cvTrue;
      RightServo.Operate = cvTrue;
    }
    Else
    {
      DarkLED.Value = 0;
      LeftServo.Operate = cvFalse;
      RightServo.Operate = cvFalse;
    }
```

Once you are ready, download the program. If all goes well, you should be able to put your hand over the robots sensor and it will scurry away for a couple seconds.

What next?

Here are some challenges based on what you just learned:

3. Push a button and light up three LEDs.
4. Could you make an LED turn off when the light on the sensor is dark?
5. Can you make the robot run from a bright light?

Chapter 3 - Programming Principles

Most programming that we use is called sequential programming. This means the programming code that we write is executed one line after another by our microcontroller. There is no way for something else to happen at the same time as we are executing this code.

For example:

```
While (cvTrue)
{
  LED1.Value = 1;
  OOPic.Delay = 200;
  LED1.Value = 0;
  OOPic.Delay = 200;
}
```

In the above code, where we blink an LED once every 4 seconds (2 seconds on, 2 seconds off), if we wanted to have the microcontroller count the number of times a button is pressed at the same time as it is doing this code, it becomes a more difficult task.

```
While (cvTrue)
{
  If (RedButton.Value == 1)
    ClickCount.Value = ClickCount.Value + 1;
  LED1.Value = 1;
  OOPic.Delay = 200;
  LED1.Value = 0;
  OOPic.Delay = 200;
}
```

What may happen is during the 2 seconds that we are waiting until we turn the LED on or off, the button may have been pressed, but will not be counted because we are not at the part in the program that will check if the button has been pressed.

We need some way to have the microcontroller do other tasks at the same time as the task we have in our main program. Having more than one task perform at the same time is called **multitasking**. The OOPIC is capable of this. We program the OOPIC with objects and make those objects interact with each other with what is called **virtual circuits**.

Object-Oriented programming

An object is something that has properties. If the object can also perform a function, it might also have methods. The best way to describe how objects work is to use an example.

A tea cup can have a **property** named “Full”. So, if the tea cup is full of tea, the Full property would be true. On the other hand, if the cup were empty, the Full property would be false. Now, if the tea cup were full, we may decide to drink it. The tea cup object could have the **method** named Drink. Once we have drunk a Full cup of tea, it will be empty, and thus the Full property will be false.

```
If (MyTeaCup.Full == cvTrue)
  MyTeaCup.Drink;
```

As you can see, methods will most likely have effects on properties. In this example, the Drink method altered the value of the Full property.

Objects can have many properties and methods. All of these properties and methods are always with the object. When we declare an object at the beginning of our program

```
TeaCup MyTeaCup = New TeaCup;
```

All of its properties and methods are available for us to use in the main program.

Virtual circuits

A virtual circuit is actually unique to the OOPIC. To make a program execute more than one task at a time can require a lot of coding because of the nature of microcontrollers in that they need to keep track of the information of each task. The OOPIC makes this simple by allowing us to create connections between objects in our program, and the OOPIC itself will keep track of all the information required to have all the tasks operating at the same time.

The idea of a virtual circuit is to abstract the idea that we have many inputs coming into our microcontroller to process, and we want to react to that information in some way.

Returning to our previous example, where we were blinking a light once every 4 seconds and trying to count the number of button presses could be rewritten as follows:

```
...

oDIO1 Button = New oDIO1;
oMathIC Add1 = New oMathIC;
oByte ClickCount = New oByte;

... set up our I/O Lines

Add1.Mode = cvAdd;
Add1.Input1.Link(ClickCount.Value);
Add1.Value = 1;
Add1.Output.Link(ClickCount.Value);
Add1.ClockIn.Link(Button.Value);
Add1.Operate = cvTrue;

While (cvTrue)
{
  LED1.Value = 1;
  OOPic.Delay = 200;
  LED1.Value = 0;
  OOPic.Delay = 200;
}
```

Virtual circuits require links between the objects, much like we use metal wires to connect electronics on our breadboard. A link will pass or update a value from one object to the other.

The oMathIC object will allow us to add 1 to our ClickCount each time the button is pressed. Have a look at the details for the oMath object in the OOPIC Help to see how it works.

Once we set all the properties for our Add1 object, it will operate each time the button is pressed, even at the same time as we are blinking our LED.

Try it!

Experiment 3-1: Heartbeat revisited

OOPic.Hz1 is a property of the OOPIC object (an object which defines some properties and methods of the OOPIC itself) which changes from 0 to 1 once every second. We will use this property to make our light blink once a second. The obvious way would be set the value of the LED to the value of OOPic.Hz1.

```
  HeartLED.Value = OOPic.Hz1;
```

But what happens when we do this? In fact, the value property of HeartLED will be set to whatever OOPic.Hz1 is at the time that this statement is executed and will remain this value. Why? Just as was described earlier, the OOPic.Hz1 property is being constantly changed, but since we only execute this statement once, the value is only set once. We need to create a virtual circuit that will constantly update the value of the HeartLED with the OOPic.Hz1 property.

Parts you will need:

Built-in Red LED (I/O Line 7) on OOBoard

Enter the program as listed below.

Program Listing:

```
// Single bit Digital Input/Output
oDIO1 HeartLED = New oDIO1;

oWire HeartWire = New oWire;

Sub Void Main(Void)
{
  HeartLED.IOLine = 7;
  HeartLED.Direction = cvOutput;
  HeartWire.Input.Link(ooPIC.Hz1);
  HeartWire.Output.Link(HeartLED.Value);
  HeartWire.Operate = cvTrue;
}
```

Walk through the program:

Now that we know that each object has properties and methods, we can now understand that the properties and methods will help define our objects. The

“HeartLED”, which is our heartbeat indicator, is defined at the beginning as a **oDIO1** object, or a single bit digital input/output object.

```
oDIO1 HeartLED = New oDIO1;
```

Now, to create a virtual circuit, which will carry the value of OOPic.Hz1 to our HeartLED, we need to use a virtual wire to transfer the value.

```
oWire HeartWire = New oWire;
```

It’s as simple as connecting all the inputs and outputs of our diagram to complete the virtual circuit. We connect the input of the HeartWire to OOPic.Hz1, and the output of the wire to HeartLED. Then we set the Operate property of the wire to cvTrue to make the connection.

```
  HeartLED.IOLine = 7;
  HeartLED.Direction = cvOutput;
  HeartWire.Input.Link(ooPIC.Hz1);
  HeartWire.Output.Link(HeartLED.Value);
  HeartWire.Operate = cvTrue;
```

This will continuously transfer the value of OOPic.Hz1 to the HeartLED.Value property to make it blink once a second, and hence our heartbeat.

You will see in the next experiment that there is an even easier way to make the light blink once per second.

Experiment 3-2: Push on, Push Off

This experiment will demonstrate some more things about properties of objects. The built-in pushbuttons on the OOBoard can actually perform two functions at the same time. They can be used as an input (pushbutton) or as an LED, even at the same time. This is due a fancy object called an **oButton** on the OOPIC.

The oButton object will actually check the status of the pushbutton and also display its Value property on the LED (either on or off).

If that wasn't enough, there are more properties that the object can take on which defines how the button is used and how the LED displays its value.

For instance, if we set the Mode property to 1, then the pushbutton is treated as a toggle switch (pressed once is on, and pressed a second time is off). And if we set the Style property to 2, and the Value property is set to 1, then the LED will blink at 4 times per second. The details for all of the values that the properties can take on are found in the OOPIC Object Manual.

Let's try it.

Parts you will need:
Built-in "Yellow" pushbutton (I/O Line 6) on OOBoard

Enter the program as listed below.

Program Listing:

```
oButton YellowButton= New oButton;

Sub VoidMain(Void)
{
  YellowButton.IOLine= 6;
  YellowButton.Mode= 1;
  YellowButton.Style= 2;
}
```

Walk through the program:

What you will find is that when you press the button the first time, the LED begins to blink 4 times per second. When you press it a second time, it turns off.

If you look at some more of the values for the Style property of the oButton object in the OOPic Object manual, you will find that setting the Style to 1 will make the LED blink once per second, just what we wanted to do in Experiment 3-1!

Many objects have various properties which affect how the object operates. There are common properties that most objects possess (such as the Operate property) and there are specific properties, which make the object unique.

Experiment 3-3: A True Reflex

Now that you know more about objects and virtual circuits, you can revisit experiment 2-3 and implement a true reflex. What is a true reflex? Imagine someone startling you in a scary movie. You jump – reflexively. You didn't think about jumping, a part of your brain responded instantaneously and you jumped.

In experiment 2-3, we used an If-Then structure to "think and decide" inline with the main program loop. If we had a LED blinking it would only blink after we decided what we had to do because it has to wait for the decision. Using virtual circuits and objects, we can take advantage of the multitasking capability of the OOPIC and these objects will always be running and "interrupt" our main program when a condition (property) is met. For the reflex, we will setup our virtual circuit to watch for a shadow and if it sees a shadow, then run away.

The key is that it will happen without the main program having to act upon it and in parallel to other actions. This is exactly like our startling jump - a true reflex.

We will need to compare the light sensor value with our DarkLevel to know when to make the robot run away, so we will use the **oCompare** object. It continuously compares the input (LightSensor.Value) to its reference value (DarkLevelReference). If the input value is above or below the reference value, the Above or Below properties are set, respectively. That is, if the LightSensor.Value goes below our DarkLevelReference value, LightCompare.Below is set to 1. We will use that property to control our servos and make them move if the light level drops below our DarkLevel.

Parts you will need:

Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
1 5K Ohm photo-resistor (connected to I/O line 2)
1 1K Ohm resistor (Brown Black Red Gold)
Wire

Build it:

Here is a schematic showing you how to connect the parts together for this experiment.

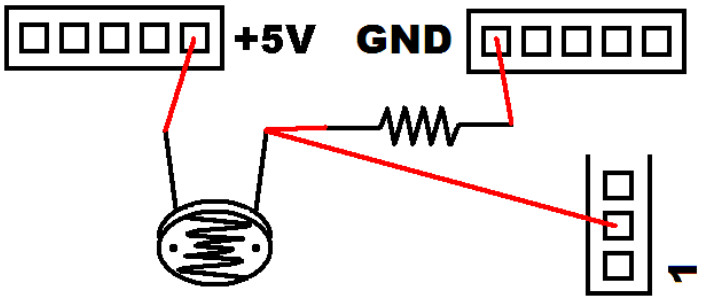


Figure 3.1 – Schematic of experiment 3-3

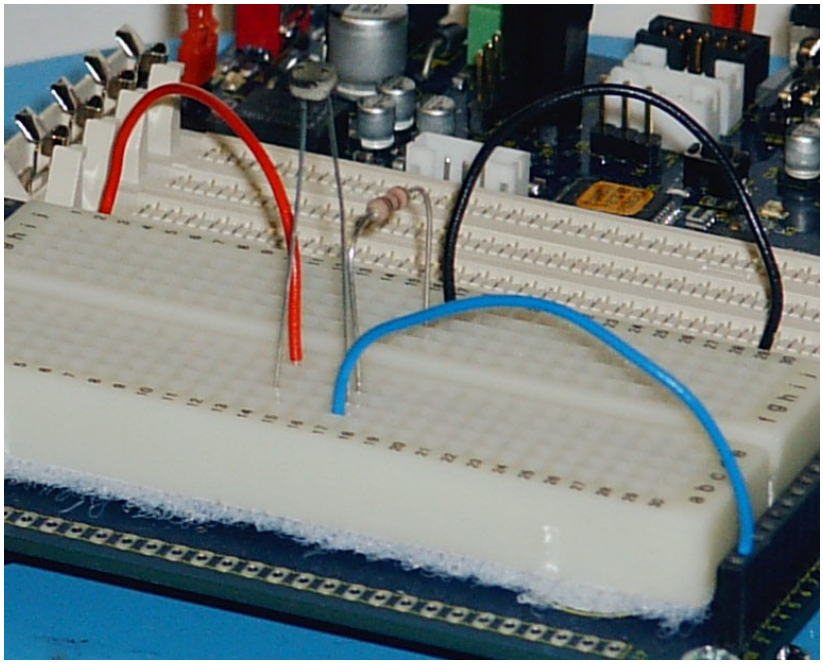


Figure 3.2 – Picture of connections

Enter the program as listed below.

Program Listing:

```
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oA2D LightSensor = New oA2D;

oCompare LightCompare = New oCompare;

oByte DarkLevelReference = New oByte;

oWire LeftWire = New oWire;
oWire RightWire = New oWire;

Final DarkLevel = 60;

Sub Void Main(Void)
{
  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = -127;
  RightServo.Value = -127;

  LightSensor.IOLine = 2;
  LightSensor.Operate = 1;

  DarkLevelReference = DarkLevel;

  LightCompare.Input.Link(LightSensor.Value);
  LightCompare.ReferenceIn.Link(DarkLevelReference.Value);
  LightCompare.Operate = cvTrue;

  LeftWire.Input.Link(LightCompare.Below);
  LeftWire.Output.Link(LeftServo.Operate);
  LeftWire.Operate = cvTrue;

  RightWire.Input.Link(LightCompare.Below);
  RightWire.Output.Link(RightServo.Operate);
  RightWire.Operate = cvTrue;
}
```

Walk through the program:

It's important to note that the ReferenceIn property of the oCompare object expects to read the value of another object, not to just accept a value (that is, a constant, so "LightCompare.ReferenceIn = 60" will not work). This is why we have the DarkLevelReference variable (oByte). We can copy the DarkLevel constant into the Value property of DarkLevelReference, which is Linked to the ReferenceIn property of LightCompare.

The LightCompare object continuously checks the LightSensor.Value against the DarkLevelReference.Value. If it falls below the reference, then LightCompare.Below flag is set, which is connected to both the LeftServo, and RightServo, using two virtual wires.

What next?

1. How could you make the heartbeat start and stop with the push of a button?
2. What would you do to make a different LED light up when you push a button?
3. Using all three buttons, can you make the robot go forward, backward and stop?
4. What would you change in the last experiment to make the robot stop when it sees a shadow and spin when it doesn't?

Chapter 4 - Simple Behaviours

What are behaviours?

First, behaviours are more complex than a reflexive action, generally, because they incorporate more decision-making. But this does not necessarily mean that they are thought intensive actions that require significant understanding of artificial intelligence.

The best way to describe a behaviour may be by example. In our first experiment, we are going to make our robot seek out a bright light source. To do this our robot must evaluate two different sensors (eyes) and decide which way the light source is and whether to turn or go forward. In a way, it could be viewed as a collection of simpler reflexes working together to achieve a goal, in this case, seek light.

For our robot, we are going to virtually connect up one light sensors output to the servo motor on the opposite side of the robot and vice versa for the other sensor and servo motor. We will have a two-eyed robot seeking light. When a bright light is shined on only one of the sensors (eye), it will turn on the opposite motor (causing the robot to turn towards the light), until both eyes see the bright light and then the other motor will turn on as well and the robot will move forward.

Two simple reflexes, combined into a much more complex (appearing) behaviour of seeking light.

Experiment 4-1: Seek Light

This experiment will demonstrate how we can make the robot exhibit a behaviour. The robot will follow a light beam (using a flashlight). If you shine the light on both sensors, it will come forward, but if you flash it on only one sensor, it will turn in that direction, hence seeking the light.

Parts you will need:

Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
2 5K Ohm photo-resistors (connected to I/O line 2)
2 1K Ohm resistors (Brown Black Red Gold)
Wire
****A flashlight or bright light source (not included)

Build it:

Here is a schematic showing you how to connect the parts together for this experiment.

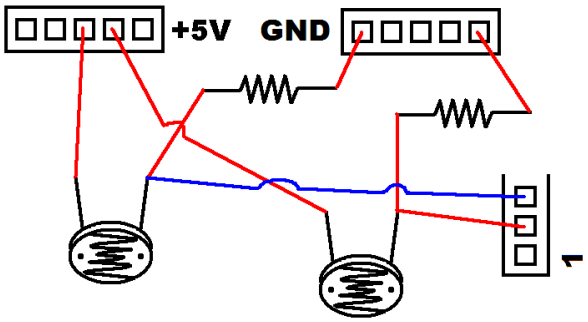


Figure 4.1 – Schematic of experiment 4-1.

When you hook up the two sensors, they should be reasonably apart and at different ends of the breadboard and facing forward mostly.

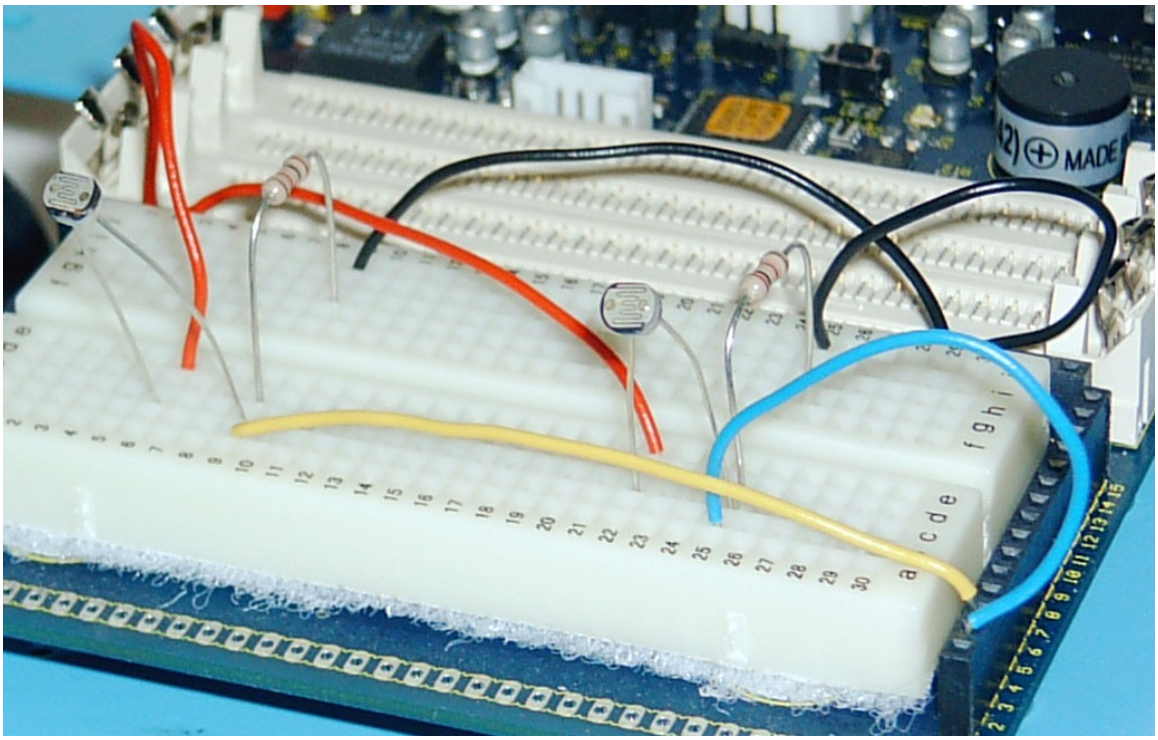


Figure 4.2 – Picture of connections

Enter the program as listed below.

Program Listing:

```
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oA2D LeftLightSensor = New oA2D;
oA2D RightLightSensor = New oA2D;

oCompare LeftLightCompare = New oCompare;
oCompare RightLightCompare = New oCompare;

oByte LeftBrightLevelReference = New oByte;
oByte RightBrightLevelReference = New oByte;

oWire LeftWire = New oWire;
oWire RightWire = New oWire;

Final LeftBrightLevel = 120;
Final RightBrightLevel = 120;

Sub Void Main(Void)
{
  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;
```

```
  LeftServo.Value = 127;
  RightServo.Value = 127;

  LeftLightSensor.IOLine = 2;
  LeftLightSensor.Operate = cvTrue;

  RightLightSensor.IOLine = 3;
  RightLightSensor.Operate = cvTrue;

  LeftBrightLevelReference = LeftBrightLevel;
  RightBrightLevelReference = RightBrightLevel;

  LeftLightCompare.Input.Link(LeftLightSensor.Value);
  LeftLightCompare.ReferenceIn.Link(LeftBrightLevelReference.Value);
  LeftLightCompare.Operate = cvTrue;

  RightLightCompare.Input.Link(RightLightSensor.Value);
  RightLightCompare.ReferenceIn.Link(RightBrightLevelReference.Value);
  RightLightCompare.Operate = cvTrue;

  LeftWire.Input.Link(LeftLightCompare.Above);
  LeftWire.Output.Link(RightServo.Operate);
  LeftWire.Operate = cvTrue;

  RightWire.Input.Link(RightLightCompare.Above);
  RightWire.Output.Link(LeftServo.Operate);
  RightWire.Operate = cvTrue;
}
```

Walk through the program:

This experiment is different from our previous experiments in a few ways. We are now adding a second sensor to allow more motions for our robot. Also, instead of running away from light, the robot will move towards the light, so we need to find the value for our BrightLevels.

It is important to note that both light sensors will have different ambient levels, so we should use the object viewer in the programming environment to identify what the values are for both of them. Once we have determined what the ambient levels are, we should add 30 (a nice level of brightness over the ambient level) and assign it LeftBrightLevel and RightBrightLevel.

```
Final LeftBrightLevel = 120;
Final RightBrightLevel = 120;
```

As described before, we use some oByte objects to hold these reference values for the oCompare objects (LeftLightCompare and RightLightCompare).

```
  LeftBrightLevelReference = LeftBrightLevel;
  RightBrightLevelReference = RightBrightLevel;
```

Now, the action is simple. When the left light sensor is above LeftBrightLevel, we make the right wheel move. Likewise, when the right light sensor is above RightBrightLevel, we make the left wheel move. As before, we have already set the speed of the servos, and all we do is turn them on and off using the Operate property of each servo.

```
LeftLightCompare.Input.Link(LeftLightSensor.Value);
LeftLightCompare.ReferenceIn.Link(LeftBrightLevelReference.Value);
LeftLightCompare.Operate = cvTrue;

RightLightCompare.Input.Link(RightLightSensor.Value);
RightLightCompare.ReferenceIn.Link(RightBrightLevelReference.Value);
RightLightCompare.Operate = cvTrue;

LeftWire.Input.Link(LeftLightCompare.Above);
LeftWire.Output.Link(RightServo.Operate);
LeftWire.Operate = cvTrue;

RightWire.Input.Link(RightLightCompare.Above);
RightWire.Output.Link(LeftServo.Operate);
RightWire.Operate = cvTrue;
```

Now, download the program to the robot and you can grab your flashlight and take your robot for a walk. Fun!

Experiment 4-2: Scream and Run

A group of reactions to an event are sometimes called behaviours as well. For this example, we are going to modify our experiment 3-3 to add in another reaction to the shadow – a scream (well at least a faint whimper), to indicate the startle at the same time as the servos carry the robot away from the danger.

In this experiment, we will introduce the **oEvent** object, which will allow us to perform a series of statements when a property changes. This is very useful since we may need to perform some complex calculations or an awkward set of statements to make the robot perform what we intend it to do.

Simply, when the Operate property of the oEvent object is set to 1, the OOPIC begins executing the statements found in the subroutine named the same as the object with "_Code" after it. For example, if we had an oEvent object named "BlinkLights", the subroutine name associated with it is named "BlinkLights_Code".

Another object we will introduce is the **oFreq** object. It will generate and output a high frequency tone on the built-in speaker of the OOBoard. The calculation for the tone is shown in the OOPIC Object Manual. We use this object to demonstrate that we can perform several actions when the oEvent object is triggered.

Parts you will need:

Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
Built-in Speaker (I/O Line 21) on OOBoard
1 5K Ohm photo-resistor (connected to I/O line 2)
1 1K Ohm resistor (Brown Black Red Gold)
Wire

Build it:

Here is a schematic showing you how to connect the parts together for this experiment.

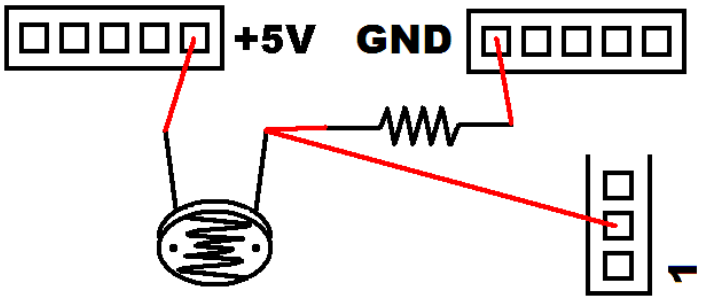


Figure 4.3 – Schematic for experiment 4-2

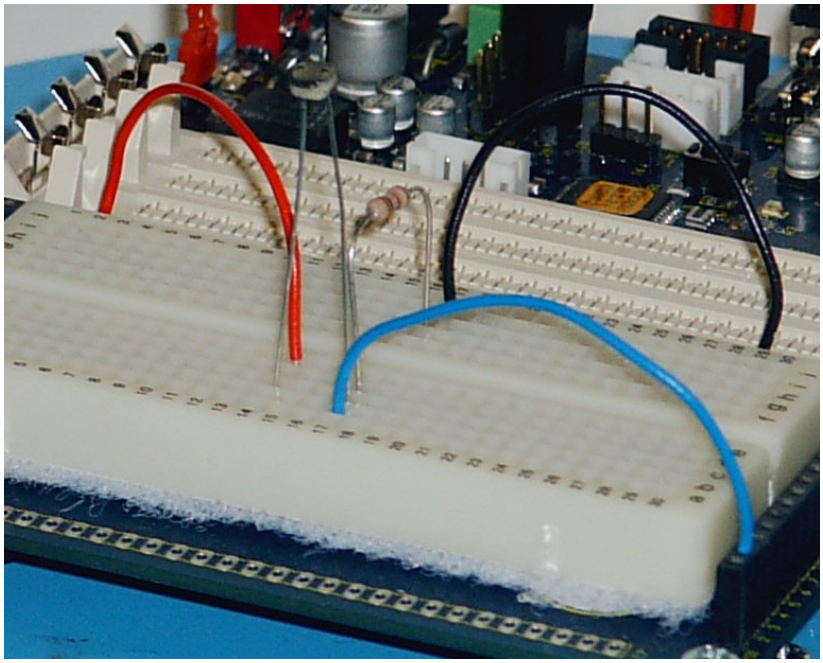


Figure 4.4 – Picture of connections

Enter the program as listed below.

Program Listing:

```
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oA2D LightSensor = New oA2D;

oCompare LightCompare = New oCompare;

oByte DarkLevelReference = New oByte;

oWire Connect = New oWire;

oFreq Scream = New oFreq;

oEvent RunAway = New oEvent;

Final DarkLevel = 60;

Sub Void Main(Void)
{
  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = -127;
  RightServo.Value = -127;

  LightSensor.IOLine = 2;
  LightSensor.Operate = 1;

  DarkLevelReference = DarkLevel;

  LightCompare.Input.Link(LightSensor.Value);
  LightCompare.ReferenceIn.Link(DarkLevelReference.Value);
  LightCompare.Operate = cvTrue;

  Connect.Input.Link(LightCompare.Below);
  Connect.Output.Link(RunAway.Operate);
  Connect.Operate = cvTrue;

  Scream.Value = 61473;

}

Sub Void RunAway_Code(Void)
{
  Scream.Operate = cvTrue;
  ooPIC.Delay = 50;
  LeftServo.Operate = cvTrue;
  RightServo.Operate = cvTrue;
```

```
  ooPIC.Delay = 200;
  LeftServo.Operate = cvFalse;
  RightServo.Operate = cvFalse;
  Scream.Operate = cvFalse;
}
```

Walk through the program:

Most of this program is similar to Experiment 3-3, except that we are using an oEvent object. The only difference is that instead of directly manipulating the Operate properties of the servos, we set the Operate property of the RunAway event object.

```
  Connect.Input.Link(LightCompare.Below);
  Connect.Output.Link(RunAway.Operate);
  Connect.Operate = cvTrue;
```

The only preparation for the oEvent object that is needed, is the subroutine with the code that we intend to run when the Operate property is set to 1.

```
Sub Void RunAway_Code(Void)
{
  Scream.Operate = cvTrue;
  ooPIC.Delay = 50;
  LeftServo.Operate = cvTrue;
  RightServo.Operate = cvTrue;
  ooPIC.Delay = 200;
  LeftServo.Operate = cvFalse;
  RightServo.Operate = cvFalse;
  Scream.Operate = cvFalse;
}
```

What this will do is turn on the speaker for a half a second (OOPic.Delay = 50), then turn on the wheels for 2 seconds (OOPic.Delay = 200), and then turn everything off.

When you are ready, download the program. And test out your robots newest behaviour, screaming and running away from a shadow. Silly robot!

What next?

1. Using the principles of experiment 4-1, could you make a robot that avoids light (is photophobic or scared of light, like a cockroach)?
2. Instead of running away from a shadow, could you make your robot play 5 different tones? (Hint: oFreq object)

Chapter 5 - Learning to Drive

By necessity, a mobile robot must be able to get around in its environment. This is not a particularly easy task, which is probably why we don't see robots wandering around our school hallways everyday. Humans have many extremely evolved sensors to help us navigate around our environment, but for your robot – it only has one simple sensor, so the best we can expect is simple navigation abilities.

Simple Navigation

With a limited number of sensors to help, the robot must be able to at least avoid running into objects, walls and such. For this experiment, the robot is outfitted with an IR ranger unit on its front sensor mounting point. This sensor detects objects and how far they are away and converts it to an analog voltage.

You can then use that information to decide whether or not we should turn to avoid it, yet, or keep on going forward.

The IR ranger works by sending out a pulse of Infrared light and then timing how long before it gets bounced back. Our ranger has a maximum range of 30 cm, so anything farther away than that it will not "see". It also has a minimum range of 4 cm. Anything inside this range cannot be measured as well.

Now, for our robot to navigate we must make a program that will decide when to turn and when to go forward. For these experiments, you will only turn the robot on direction when detecting an obstacle.

Experiment 5-1: Measuring Distance

Before you start navigating around the world, you should learn how to determine how far away something is. Measuring the distance to an object is an important skill in any environment.

You can use your robot's IR ranger to measure distances to objects. The ranger produces a voltage (between 0 and +5 volts) depending on how far away an object is from it. We will use it to make our robot move forward until it is only 5 centimeters away from a wall (or another obstacle).

The OOPIC has an object called **oIRRange** specifically for this sensor, which will make the process easier to implement.

Parts you will need:

Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
1 Infrared Ranger (I/O Line 4)
Wire
A ruler (not included)

******If you have not installed the IR ranger module and sensor mount on the robot yet, please install them now following the instructions from Appendix C.

Build it:

Here is a picture showing you how to connect the parts together for this experiment.

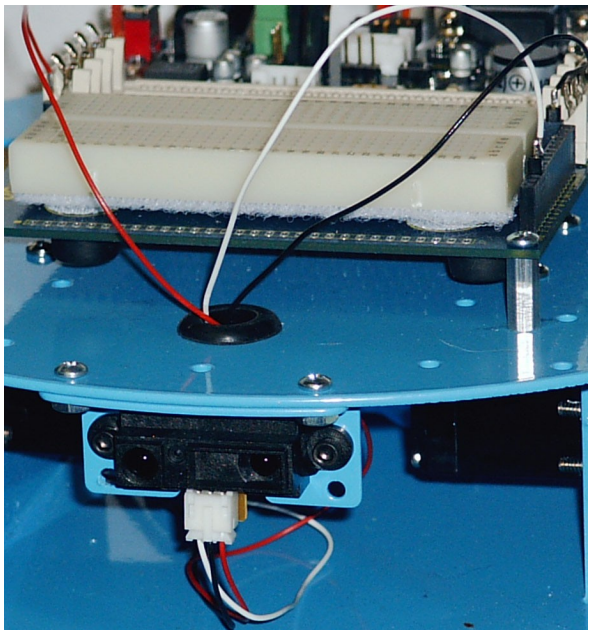


Figure 5.1 – Picture of connections

The red wire connects to +5v, black wire to GND, and the white wire to IOline 4.

Enter the program as listed below.

Program Listing:

```
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oIRRange Measure = New oIRRange;

oByte StopAt5cm = New oByte;

oCompare DistanceCompare = New oCompare;

oWire LeftWire = New oWire;
oWire RightWire = New oWire;

Final ValueAt5cm = 30;

Sub Void Main(Void)
{
  Measure.IOLine = 4;
```

```
  Measure.Center = 20;
  Measure.Operate = cvTrue;

  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = 127;
  RightServo.Value = 127;

  StopAt5cm.Value = ValueAt5cm;

  DistanceCompare.Input.Link(Measure.Value);
  DistanceCompare.ReferenceIn.Link(StopAt5cm.Value);
  DistanceCompare.Operate = cvTrue;

  LeftWire.Input.Link(DistanceCompare.Above);
  LeftWire.Output.Link(LeftServo.Operate);
  LeftWire.Operate = cvTrue;

  RightWire.Input.Link(DistanceCompare.Above);
  RightWire.Output.Link(RightServo.Operate);
  RightWire.Operate = cvTrue;
}
```

Walk through the program:

We will call our oIRRange object Measure (since that's what we are doing).

```
oIRRange Measure = New oIRRange;
```

We will need to find out the value of the oIRRange object when it is only 5 centimeters from an obstacle, so using a ruler, put something (like a book) in front of the infrared ranger at a distance of 5 centimeters and look at the value of the Measure object (our oIRRange object). We can set our constant named ValueAt5cm to that value.

```
Final ValueAt5cm = 30;
```

Now, using the oCompare object, we simply compare the value from the Measure object to our ValueAt5cm, and while it is above that value, we keep our wheels moving. If it equals, or falls below that value, the wheels stop.

```
  DistanceCompare.Input.Link(Measure.Value);
  DistanceCompare.ReferenceIn.Link(StopAt5cm.Value);
  DistanceCompare.Operate = cvTrue;

  LeftWire.Input.Link(DistanceCompare.Above);
  LeftWire.Output.Link(LeftServo.Operate);
```

```
LeftWire.Operate = cvTrue;

RightWire.Input.Link(DistanceCompare.Above);
RightWire.Output.Link(RightServo.Operate);
RightWire.Operate = cvTrue;
```

Once you are ready, download your program. You can also use the object viewer for the oIRRange object to check your values for your measurements. When you turn on your robot, it should move forward until it gets 5cm away from your hand or object.

This is the first step in navigation – detecting the obstacles and knowing how far away they are.

Experiment 5-2: Avoiding Obstacles

To make our robot more interesting, we will use the oIRRange object to help us avoid obstacles and move around them.

This is very similar to Experiment 5-1, except that instead of stopping, the robot turns until there is nothing in front of it so that it may continue moving forward.

Parts you will need:
Two built-in servos (I/O Line 29 for Left, I/O Line 30 for Right) with wheels on Rogue Blue ERS.
1 Infrared Ranger (I/O Line 4)
Wire

Build it:

Same as 5-1.

Enter the program as listed below.

Program Listing:

```
oServoX LeftServo = New oServoX;
oServoX RightServo = New oServoX;

oIRRange Measure = New oIRRange;

Final TooClose = 70;

Sub Void Main(Void)
{

  Measure.IOLine = 4;
  Measure.Center = 20;
  Measure.Operate = cvTrue;

  LeftServo.IOLine = 29;
  RightServo.IOLine = 30;
  RightServo.InvertOut = cvTrue;

  LeftServo.Value = 127;
  RightServo.Value = 127;

  LeftServo.Operate = cvTrue;
  RightServo.Operate = cvTrue;
```

```
  While (cvTrue)
  {
    If (Measure.Value < TooClose)
    {
      // Start Turning
      RightServo.Value = -127;
      While (Measure.Value < TooClose)
      {
        // Keep Turning
      }
      // Now continue turning for a little longer
      ooPIC.Delay = 50;
      // Now go forward
      RightServo.Value = 127;
    }
  }
}
```

Walk through the program:

What we are trying to accomplish in this experiment is more thinking than reacting. In the previous experiments, when a condition was met (such as a light value was below a certain level), the robot automatically did something. In this experiment, we actually process information, and consider what should be done.

First off, we check to see if we are too close to an object.

```
    If (Measure.Value < TooClose)
```

If we are, then we need to start turning.

```
      RightServo.Value = -127;
```

Now, instead of just turning for a certain amount of time and then continuing forward, we keep checking until the obstacle is out of the way.

```
      While (Measure.Value < TooClose)
      {
        // Keep Turning
      }
```

It would seem to be a good idea to just to start moving forward now, but here is where some thinking is required on the programmer's part. Since we are only using a single sensor mounted in the middle of the robot, if we just started moving forward when the value was acceptable, more than likely, the obstacle would just be to the side of the sensor. So what we should do is continue turning just a little bit more so that we go past the obstacle.

```
OOPic.Delay = 50;
```

Now we can continue forward.

```
RightServo.Value = 127;
```

And we can begin doing the same process over again.

When you are ready, download your program to the robot. When your robot is ready, turn on the robot. It will go forward until it sees and obstacle and then turn right until it does not see an obstacle. Unfortunately, our little robot has only the one eye for seeing obstacles so it will still occasionally run into or skid around objects.

What next?

1. Our navigation required the robot to “think”, could you implement simple navigation as a reflexive behaviour? Why or why not?
2. We have all the pieces of a complex robot. Now, it’s your turn. Put them all together into one program. Heartbeat, Scream and Run, Seek Light and Simple Navigation.

Appendix A – Electrical and Mechanical Reference

Robot Notes

A note about storage and organization

The robot comes in a storage case with snap close lid. You can store the assembled robot in this case between classes. Make sure that the power is off before packing it away and also, remove the servo wires as they will hit the top of the case and may damage the wires with closure. You can replace the servo wires in position when you remove the robot from the case again.

The large snap case, experiment parts snap case and the robot all have labels that allow them to numbered, lettered, or named to help with organization of the robot parts.

****After each complete set of experiments, it is a good idea to clear the memory of the robot for the next class. Select **Tools|Erase EEPROM|Erase**. Select 8K for EEPROM size.

Battery charging and care

The NiCad battery is charged with the on-board charger. You can charge the battery even when the power is off to the robot as long as you leave the wall power supply connected. The green LED on the charger circuit will light when the unit is charging (or capable of charging if a battery is not present). The charger is a constant current charger and it can be left charging a battery pack for up to 14 hours. If, for any reason, a battery pack seems excessively warm, remove it from the charger immediately and contact support.

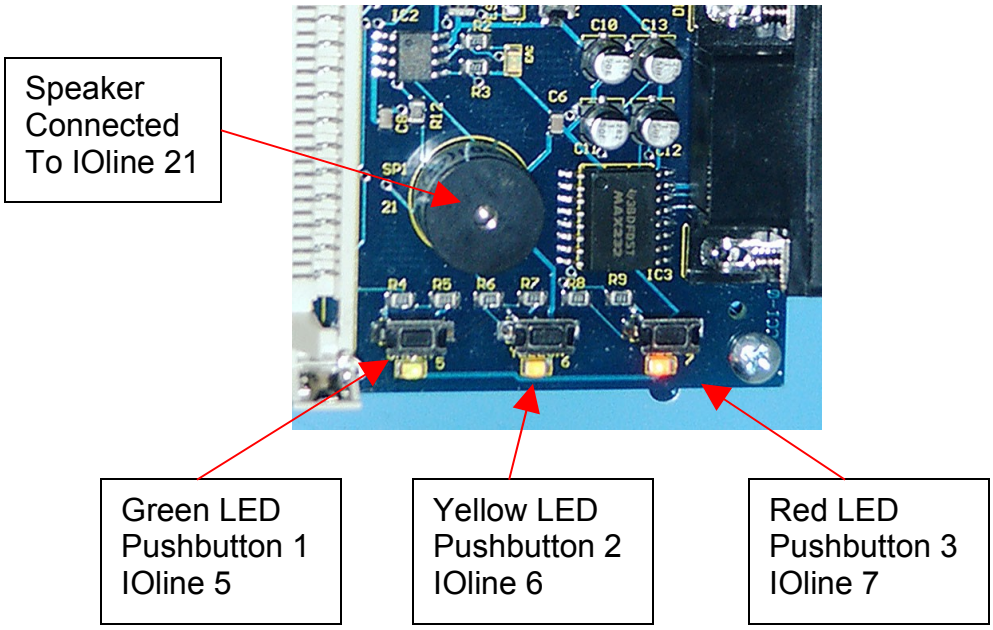
****IMPORTANT** – no other battery type should be connected to the battery connector as serious damage may occur to the robot or battery. If the battery needs replacing, replace only with the Rogue Robotics battery pack recommended for this unit.

When you first receive your kit, your battery will have a partial charge. You can charge the battery anytime after you receive it, or use the current battery until it is low.

When the battery gets low, the robot may start acting unpredictably and not seem to respond when programming. You can check your battery charge with a voltmeter, on occasion, if you do not charge often. It will take up to 3 charge/discharge cycles (8-10 hours) for you battery to reach peak voltage.

OOBoard Notes

Pushbuttons, LEDS and Speaker on-board



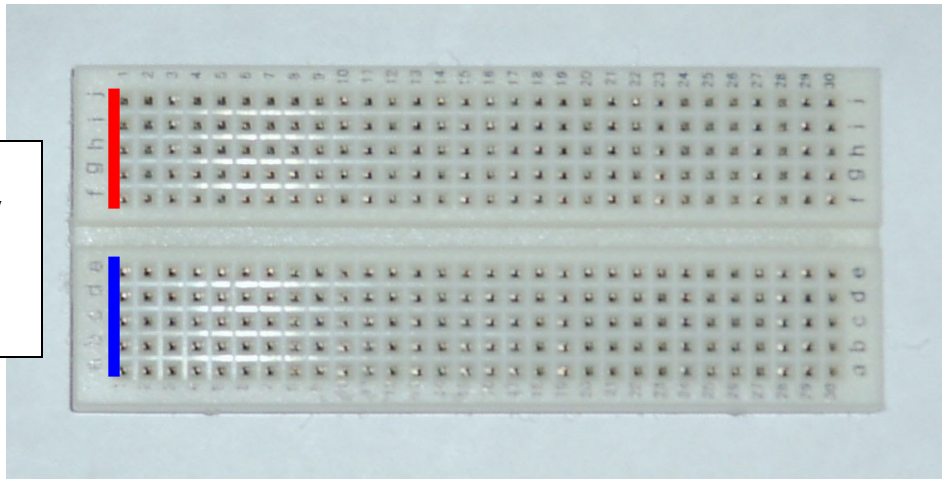
OOBoard Mechanical and Electrical Reference

You can find a detailed reference for the OOBoard in the IDE help manual under **OOPIC Connectors and Mechanical**, and then **OOBoard Specifications**.

Reset Button

Beside the processor (the chip in the middle with a yellow sticker on it) there is a button with RESET beside it. If your robot doesn't seem to respond correctly or not at all, you should first try pressing this button to restart the processor. Pushing this button will also restart your program, so if you want to start your program from the beginning again, this is the button to push.

Bread boarding rules



These bars indicate they are connected electrically

All the vertical columns indicated in blue and red are connected electrically to all 5 of the holes in each column (red or blue). They are NOT connected to the adjacent columns; this is where you use your jumper wires when you want to connect two columns.

Resistor Color Code

A resistor has 4 bands of color on it. These color indicate its value of resistance in Ohms.

Using the table below you can calculate the value of a resistor from its colored bands. The first two bands are values (0-9), the third band is a multiplier (10^color band), and the final band is the tolerance (+/- %).

COLOR	1st BAND	2nd BAND	MULTIPLIER	TOLERANCE
BLACK	0	0	1	
BROWN	1	1	10 (10^1)	
RED	2	2	100(10^2)	
ORANGE	3	3	1,000(10^3)	
YELLOW	4	4	10,000(10^4)	
GREEN	5	5	100,000(10^5)	
BLUE	6	6	1,000,000(10^6)	
VIOLET	7	7	10,000,000(10^7)	
GRAY	8	8	100,000,000(10^8)	
WHITE	9	9	1,000,000,000(10^9)	
GOLD				5%
SILVER				10%

For example:

You have a resistor with the colors ORANGE RED GREEN and GOLD. This translates to 32 x 10^5 = 3,200,000 ohms or 3.2 Million ohms or 3.2MΩ (Mega Ohms).

Appendix B - Parts List

When you receive your robot kit, you should check to make sure you received all your parts as soon as possible. And, if you should lose any parts, you can order replacement parts using the reference below.

Robot kit

QTY 1	Robot Base with servo mounts
QTY 1	Level 2 plate with grommets
QTY 2	Continuous Rotation Servos
QTY 2	3” Robot Wheels with hubs mounted
QTY 1	Battery extension cable
QTY 1	7.2V NiCad battery pack
QTY 1	12VDC, 200mA wall power supply
QTY 1	OOBoard brain board
QTY 1	Removable breadboard
QTY 1	Universal Sensor Mount Kit
QTY 1	6’ serial cable
QTY 1	Combination screwdriver
QTY 1	Software and Documentation CD-ROM
QTY 1	Robot Brain mounting kit
QTY 1	Base/Servo/Battery hardware kit
QTY 1	Extra hardware kit
QTY 1	Experiment Parts kit (see below)
QTY 1	Snap Storage Case for robot and parts storage

Experiment Parts Kit

QTY 1	Sharp GP2D120 IR Ranger Module
QTY 1	3 conductor cable with JST connector for GP2D120
QTY 1	LED (RED)
QTY 1	Resistor 470 Ohm, 1/8W
QTY 2	Photo resistor – 5K
QTY 2	Resistor 1K Ohm, 1/4W
QTY 1	10 pack of jumper wires stripped and tinned at each end
QTY 1	Snap storage case (5/7 compartment)

Appendix C - Robot Assembly and Software Installation

Assembling the Robot

The assembly manual for the Rogue Blue ERS robot is located on the CD-ROM in HTML format under the directory called “Assembly Guide”.

Installing the OOPIC Program Editor

The program editor is supplied by Savage Innovations and is free for use with this robotic system. Updates are available online from www.oopic.com or www.roguerobotics.com when available.

The program editor files for installation are located on the CDROM under the directory called “Software”.

Windows 95,98, ME

To install this version, you will need the full install.

- OOPicOOB.exe - (Full install)

Windows 2000, XP, NT

In order to use the OOPIC multi-language compiler with Windows NT or 2000, an additional file needs to be installed. To install it, follow these steps:

1. **You MUST do this first**: Run Port95nt.exe that will install a Windows NT printer port driver.
2. Install the full version (OOPicOOB.exe).

If your install encounters problems, you can contact support@roguerobotics.com for assistance.

Support Information

For product support questions, you can use any of the following free support methods:

Contact Information

Phone: 1-866-99ROGUE (1-866-997-6483) 9am-5pm EST weekdays
Email: support@roguerobotics.com
Web: FAQ available soon on www.roguerobotics.com

Discussion forums

The OOPIC discussion forum for programming related topics can be found at:

http://groups.yahoo.com/group/oopic/

Updates

Our curriculum will have various updates available from time to time. Check www.roguerobotics.com for new updates. All updates are free for download, although, it is possible that some updates may require new parts for the "experiment parts kit".

Other references

Dennis Clark's – **Programming and Customizing the OOPIC microcontroller** – Book – ISBN:0-07-142084-3 Published by Mc Graw-Hill.

While this book was published before the Rogue Blue ERS or OOBoard from Rogue Robotics were available, it is still a great reference for programming techniques and interfacing for the OOPIC microcontroller.

Warranty

Rogue Robotics Corporation Inc. warrants its products against defects in materials and workmanship for a period of 90 days. If a warranty problem should occur in this period contact us at either support@roguerobotics.com or 1-866-99ROGUE (1-866-997-6483) for a Return Material Authorization (RMA) number and instructions on how to return the product for repair or replacement at the discretion of Rogue Robotics Corporation Inc.

Disclaimer of Liability

Rogue Robotics Corporation Inc. is not responsible for any special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, good-will, damage to or replacement of equipment or property, and any costs or recovering of any material or goods associated with the assembly or use of our products.

Rogue Robotics Corporation Inc. reserves the right to make substitutions and changes to our products without prior notice.

Copyrights and Trademarks

Accuracy Notice

This document may contain technical inaccuracies or typographical errors. The robot or parts kit as illustrated or indicated may change without notice due to product changes and improvements.