Programming with Robots

Albert W. Schueller
Whitman College

October 12, 2011

Thanks to Patricia "Alex" Robinson for reading this over and helping me to keep it clean.

Chapter 1

Introduction

1.1 External References

Throughout these notes the reader is directed to external references. Unless otherwise specified, these external references were all created by the developers of the RobotC software at Carnegie-Mellon's Robotics Laboratory. The materials are distributed with the software and are copyrighted and unedited. Because RobotC is still actively being developed, there are cases in which the documentation does not match the RobotC behavior. The references are stored locally to improve access to the materials and to ensure that they match the version of the software that we are using.

1.2 Why Robots?

Why learn the basics of programming using robots instead of more traditional method? For the last 50 years mainstream computer science has centered on the manipulation of abstract digital information. Programming for devices that interact with the physical world has always been an area of specialization for individuals that have already run the gauntlet of abstract information-based computer science.

In recent years, we have seen a proliferation of processing devices that collect and manage information from their real-time environments via some physical interface component–among them, anti-lock brakes, Mars rovers, tele-surgery, artificial limbs, and even iPods. As these devices become ubiquitous, a liberally educated person should have some familiarity with the ways in which such devices work–their capabilities and limitations.

Chapter 2

Hardware and Software

Much of computer science lies at the interface between hardware and software. **Hardware** is electronic equipment that is controlled by a set of abstract instructions called **software**. Both categories have a variety of subcategories.

2.1 Hardware

Computer hardware is typically electronic equipment that responds in well-defined ways to specific commands. Over the years, a collection of useful kinds of hardware has developed:

1. **Central processing unit** (CPU) - a specialized integrated circuit that accepts certain electronic inputs and, through a series of logic circuits, produces measurable computational outputs.

2. **Random access memory** (RAM) - stores information in integrated circuits that reset if power is lost. The CPU has fast access to this information and uses it for "short-term" memory during computation.

3. **Hard disk drive** (HDD) - stores information on magnetized platters that spin rapidly. Information is stored and retrieved by a collection of arms that swing back and forth across the surfaces of the platters touching down periodically to read from or write to the platters. These devices fall into the category of "secondary storage" because the CPU does not have direct access to the information. Typically, information from the HDD must be loaded into RAM before being processed by the CPU. Reading and writing information from HDD's is slower than RAM.

4. Other kinds of **secondary storage** - optical disks like CD's or DVD's where light (lasers) are used to read information from disks; flash memory where information is stored in integrated circuits that, unlike RAM, do not reset if power is lost; all of these are slower than HDD's or RAM.

5. **Video card** - is a specialized collection of CPU's and RAM tailored for rendering images to a video display.

6. **Motherboard** - a collection of interconnected slots that integrates and facilitates the passing of information between other standardized pieces of hardware. The channels of communication between the CPU and the RAM lie in the motherboard. The rate at which information can travel between different hardware elements is not only determined by the hardware elements themselves, but by the speed of the interconnections provided by the motherboard.

7. **Interfaces** - include the equipment humans use to receive information from or provide information to a computing device. For example, we receive information through the video display, printer, and the sound card. We provide information through the keyboard, mouse, microphone, or touchscreen.

In robotics, some of these terms take on expanded meanings. The most significant being the definition of interface. Robots are designed to interface with some aspect of the physical world other than humans (motors, sensors).

2.2 Software

Software is a collection of abstract (intangible) information that represents instructions for a particular collection of hardware to accomplish a specific task. Writing such instructions relies on knowing the capabilities of the hardware, the specific commands necessary to elicit those capabilities, and a method of delivering those commands to the hardware.

For example, we know that one of a HDD's capabilities is to store information. If we wish to write a set of instructions to store information, we must learn the specific commands required to spin up the platters, locate an empty place to write the information to be stored, move the read/write arms to the correct location, lower the arm to touch the platter etc. Finally, we must convey our instructions to the HDD.

Generally, software instructions may be written at three different levels:

1. **Machine language** - not human readable and matches exactly what the CPU expects in order to elicit a particular capability–think 0's and 1's.

2. **Assembly language** - human readable representations of CPU instructions. While assembly language is human readable, its command set, like the CPU's, is primitive. Even the simplest instructions, like those required to multiply two numbers, can be quite tedious to write.

   Most modern CPU's and/or motherboards have interpreters that translate assembly language to machine language before feeding instructions to the CPU.

3. **High-level language** - human readable and usually has a much richer set of commands available (though those commands necessarily can only be combinations of assembly commands). Translating the high-level language to machine language is too complicated for the CPU's built in interpreter so a separate piece of software called a **compiler** is required. A compiler translates the high-level instructions to assembly or machine instructions which are then fed to the CPU for execution.

   Examples of high-level languages are: C, C++, Fortran, or RobotC to name a few.

A **robot** is a programmable device that can both sense and change aspects of its environment.

2.3 Exercises

1. Who coined the term “robot”? Give a little history.
2. What are some more formal definitions of robot?
3. Who manufactures and what model is the CPU in a Mindstorm NXT robot?
4. Who manufactures and what model is the CPU in an iPod?
5. What is a bit? A byte? A kilobyte? A megabyte? A gigabyte?
6. What kind of hardware is a scanner?
7. What kind of hardware is an ethernet card (used for connecting to the Internet)?

Chapter 3

The Display

The NXT "brick" has a display that is 100 pixels wide and 64 pixels high. Unlike the latest and greatest game consoles, the display is monochrome, meaning that a particular pixel is either on or off. While simple, the display provides an invaluable tool for communicating information from within a running program.

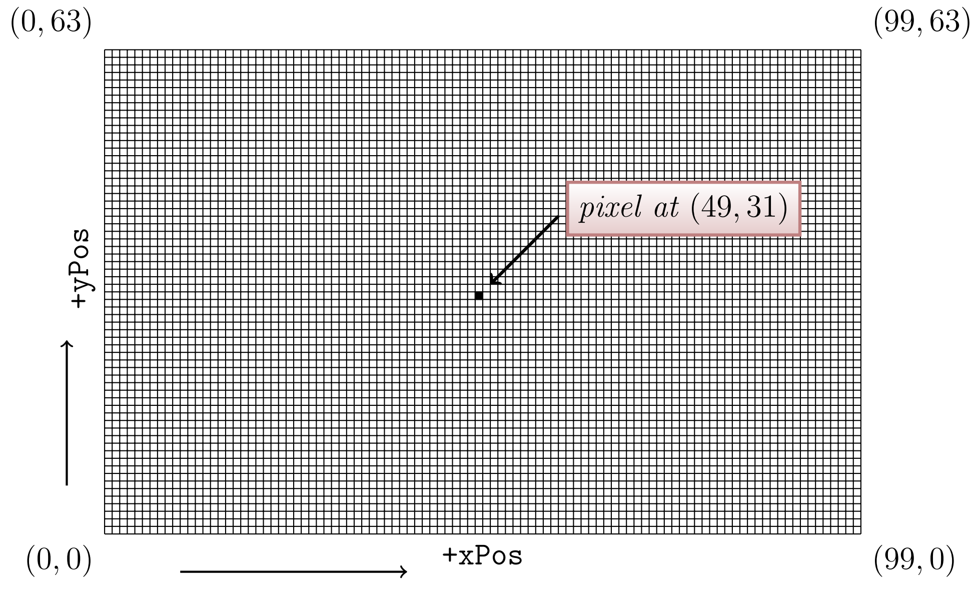


Figure 3.1: NXT display screen coordinate system.

3.1 Hello World!

An old tradition in computer science is the "Hello World!" program (HWP). The HWP is a simple program whose primary purpose is to introduce the programmer to the details of writing, saving, compiling, and running a program. It helps the programmer learn the ins and outs of the system they will be using. Our HWP will print the words “Hello World!” to the NXT display.

```
// Displays the words "Hello World!" on the NXT
// display for 5 seconds and exits.
task main() {
 nxtDisplayString(4,"Hello␣World!");
 wait1Msec(5000);
}
```

Listing 3.1: A simple “Hello World!” program for the NXT.

To execute these instructions on the NXT, run the RobotC program. Type the text exactly as it appears in Listing 3.1 into the editor window. Save your program under the name “HelloWorld”. Turn on the NXT brick and connect it to the USB port of the computer. Under the Robot menu, choose Download Program. Behind the scenes, the HWP is compiled and transferred to the NXT. Now, on the NXT, go to My Files → Software Files → HelloWorld → HelloWorld Run. If successful, the words “Hello World!” will appear on the display.

3.2 Program Dissection

Nearly every character in the HWP has meaning. The arrangement of the characters is important so that the compiler can translate the program into machine language. The rules of arrangement are called the **syntax**. If the syntax rules are violated, the compilation and download step will fail and the compiler will try to suggest ways to correct the mistake.

To start, we have `task main()`, signifying that this is the first section of instructions to be executed. A program may have up to 10 tasks, but the main task always starts first. The open and close curly braces (`{`, `}`) enclose a **block** of instructions. Blocks will be discussed later in the context of program variables.

The first instruction is a **call** to the **function** `nxtDisplayString()`. Enclosed in the parentheses are the **arguments** to the function. The first argument, `4`, specifies the line on which to place the words (there are 8 lines labeled 0 through 7 from top to bottom). The second argument, `"Hello␣World!"`, enclosed in double quotes, is the collection of characters, also known as a **string**, to be displayed. The instruction is **delimited** by a semi-colon, `;`. The delimiter makes it easy for the compiler to determine where one instruction ends and the next one begins. All instructions must end with a semi-colon.

The second instruction is a call to the `wait1Msec()` function. This causes the program to pause by the number of milliseconds (1 millisecond = 1/1000th of a second) specified in its argument before proceeding to the next instruction. In this case, the program pauses 5,000 milliseconds (or 5 seconds) before proceeding. If this pause is not included, the program will exit as soon as the string is displayed and it will seem as if the program does nothing at all.

The two lines at the top of Listing 3.1 are **comments** and are ignored by the compiler. Comments are useful in large programs to remind us what is going on in a program or in a particular section of the program. The characters `//` cause any characters that follow to the end of the line to be ignored by the compiler. Additional information about comments in RobotC is available here[1].

3.3 Beyond Words

There are a number of other objects, other than strings, that can easily be rendered on the display–ellipses, rectangles, lines, and circles. A summary of all of the display commands is available in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Display` section.

An important step in learning to use these commands is to understand the display's coordinate system. As mentioned earlier, the screen is 100 pixels wide and 64 pixels high. Each pixel has a unique position given by the ordered pair `(xPos,yPos)`. The origin is located at the lower-left corner of the screen and has coordinates `(0,0)`. The `xPos` coordinate moves the location left and right and ranges from 0 to 99. The `yPos` coordinate moves the location up and down and ranges from 0 to 63. Coordinates that are outside of this range are still recognized, but only the pieces of a particular object that land inside the display range will be visible.

The program in Listing 3.2 draws a filled ellipse. After a second, it clears out a rectangle from within the ellipse and displays the string `"Boo!"`. After another second, the program exits.

```
// A more advanced display program.
task main() {
 nxtFillEllipse(0,63,99,0);
 wait1Msec(1000);
 nxtDisplayBigStringAt(29,41,"Boo!");
 wait1Msec(1000);
}
```

Listing 3.2: A (slightly) more advanced demonstration of the display instructions.

[1]http://carrot.whitman.edu/Robots/PDF/Comments.pdf

3.4 Exercises

1. What are the coordinates of the corners of the display? What are the coordinates of the center of the display?

2. What command will render a diagonal line across the display going from the upper-left corner to the lower-right corner?

3. What would the arguments to the `nxtDrawEllipse()` function look like if you were to use it to render a circle of radius 5 centered at pixel `(15,30)`?

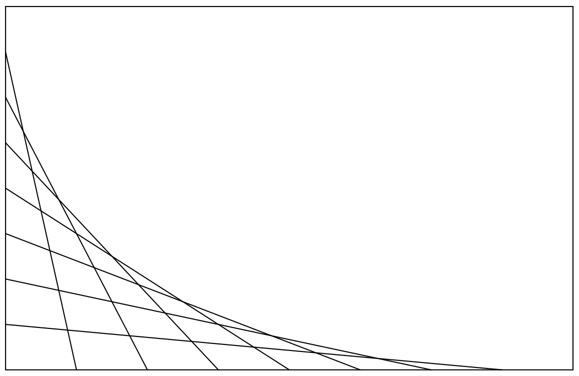
4. What is the largest ellipse that can be rendered in the display (give the command to render it)?

5. Write a program that draws the largest possible rectangle on the display and, moving inward two pixels, draws a second rectangle inside.

6. Write a program that displays the 5 Olympic rings centered in the screen. This may require some scratch paper and some hand sketching to figure out the correct positions of the circles. (Diagram #2 on this page is useful.)

7. Write a program that displays the string `"Hello␣World`"! on line 0 for 1 second, line 1 for 1 second, etc, up to line 7.

8. Write a program that will display a figure similar to



   on the NXT display screen. (Hint: Use the `nxtDrawLine()` function a few times.)

9. By including pauses between the rendering of each line, a kind of animation can be achieved. With carefully placed `wait1Msec()` function calls, animate the process of drawing the figure in Exercise 8 line by line.

10. Animate a bouncing ball on the NXT display. This may require a lot of `nxtDrawCircle()` function calls (and a lot of copy and paste). It will also require the use of the `eraseDisplay()` function.

11. Animate a pulsating circle. This will require the `eraseDisplay()` function.

12. Create an interesting display of your own.
13. Create an interesting animation of your own.

Chapter 4

Sensors and Functions

Like the display, sensors provide another kind of interface with the robot. Each of these supply information to the robot about the environment. There are four sensors available.

1. sound – measures the amplitude of sound received by its microphone.
2. light – measures the brightness of light.
3. sonar – measures the distance from the sensor to a nearby object.
4. touch – measures whether its button is depressed or not.

The first two give integer values between 0 and 100 to represent the measured quantity. The third gives integer values for distance, in centimeters, from the target (up to around a meter). The last is a Boolean value that is true if depressed and false otherwise.

4.1 Variables

The value of a sensor changes over time. Because of this, the programmer can never be sure what the value of a sensor will be when the user decides to run their program–it depends on the circumstances. An **indeterminate** is a quantity in a program whose value is not known to the programmer at the time they write the program. To handle indeterminacy, programming languages provide the ability to use **variables**. Variables act as place holders in the program for the indeterminate quantity.

For example, suppose the programmer wants to display the light sensor value on the display. Unlike earlier examples where we displayed specific shapes and strings, the value of the light sensor is not known in advance. To get around this problem, the programmer defines a variable in their program to hold the light sensor value, writes an instruction to store the current light sensor value in that variable, and prints the contents of the variable to the display. The variable plays the role of the light sensor value.

To define a variable, the programmer must give it a name and know what kind of information is to be stored in the variable. The **name** is the string the programmer types in order to refer to the variable in a program. Names must respect the following rules:

Type	**Description**	**Syntax**	**Examples**
integer	positive and negative whole numbers (and zero)	`int`	`3`, `0`, or `-1`
float	decimal values	`float`	`3.14`, `2`, or `-0.33`
character	a single character	`char`	`v`, `H`, or `2`
string	an ordered collection of characters	`string`	`Georgia`, `house`, or `a`
boolean	a value that is either true or false	`bool`	`true`, `false`

Table 4.1: The five basic datatypes.

1. no spaces.
2. no special symbols.
3. cannot start with a digit character.
4. cannot be the same as another command, e.g. `nxtDrawCircle`.

Furthermore, names are case-sensitive, e.g. the variable names `apple` and `Apple` represent different variables.

The kind of information stored in a variable is the **datatype** of the variable. There are 5 basic datatypes available as summarized in Table 4.1.

To inform the compiler about a variable, the programmer must **declare** it. A variable declaration has the general form:

[type] [variable name];

A variable must be declared *before* it is used in a program. Because of this, it is traditional to place all variable declarations near the top of the program block (the instructions enclosed by matching {}'s) where the variable is first used.

The **scope** of a variable refers to those places in a program where the variable name is recognized. In RobotC, like ordinary C, the scope of a variable is the section *after* the declaration statement of the inner-most program block containing the variable declaration. The scope extends into any sub-blocks of the block, but defers to any variables of the same name that may be declared in the sub-block, see Listing 4.1. When the program reaches the end of a block of instructions, all of the variables declared inside that block **pass out of scope**. The information in those variables is lost and the computer memory used by those variables is freed.

```
task main() {
 // inner-most block
 // containing declaration
 int n1;
 n1 = 10;
 // from here to the end of this block,
 // n1 has the value 10
 { // sub-block
   // in this sub-block, n1 has
   // the value 10.
 }

 { // sub-block
   int n1;
   n1 = -2;
   // from here to the end of this block,
   // n1 has the value -2
   // at the end of this block, the second
   // declaration of n1 passes "out of scope"
 }

 // references to n1 in this part of
 // the block use the first declaration
}
```

Listing 4.1: Variable scoping rules.