Programming with Robots

Albert W. Schueller
Whitman College

October 12, 2011

 This text was initially written by Albert Schueller and supported by a grant from Whitman College, Walla Walla, WA USA.

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.

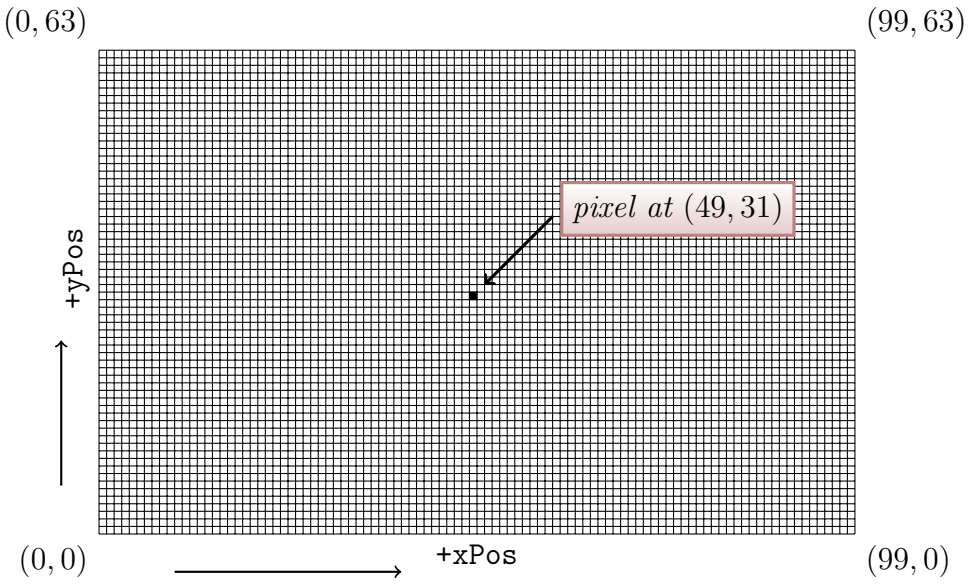


Figure 3.1: NXT display screen coordinate system.

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)`?

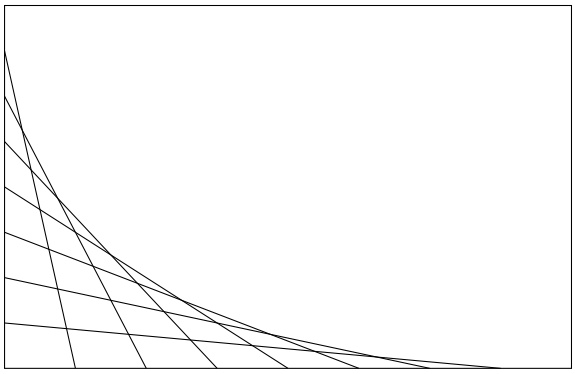
4. What is the largest ellipse that can be rendered in the display (give the command to render it)?

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.

4.2 Assignments

Variables are **assigned** values using the assignment operator, `=`. The assignment operator is used twice in Listing 4.1. Assignments always take the value on the right-hand side and place a copy into the variable on the left-hand side. Right-hand sides can be literal values like `10` or `-2`, or they can be other variables of the same type as the left-hand side. Assignments can also be made at declaration time. Some examples of assignments are in Listing 4.2. Additional information on variable naming, declaring and assignment can be found here[1].

```
task main() {
 // variable declarations,
 // some with assignments
 int n1, n2;
 string name = "Joe␣Strummer";
 string band = "The␣Clash";
 // a few more assignments
 n1 = 3;
 n2 = n1;
 n1 = 4;
 name = band;
}
```

Listing 4.2: A few examples of variable assignments.

Read the instructions in Listing 4.2 closely and try to determine the values of each of the variables just before they pass out of scope. At the end of the block of instructions, `n1` holds the value `4`, `n2` holds `3`, `name` holds `"The␣Clash"`, and `band` holds `"The␣Clash"`.

Nowhere is the difference between assignment and equality more stark than in the snippet of instruction

```
int count = 0;
count = count + 1;
```

This seemingly nonsensical assignment is actually quite useful in that it *increments* the value of the variable, `count`, by `1`. Remember that the assignment operator first reduces the right-hand side to a single value, then copies the result into the variable on the left-hand side. In this case, the right-hand side is the sum of the variable, `count`, and the literal value, `1`. The previous line set the value of `count` to `0`, so the result of the assignment is to set the value of `count` to `1`. This kind of assignment is great for counting events–keeping track of how many times something has happened. It is important that the variable being incremented has a meaningful value before doing this kind of assignment. Otherwise, the result is unpredictable.

Incrementing by 1 is just one example of this type of assignment. The following snippet increments by 2's–handy if we want a sequence of even numbers.

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

```
int count = 0;
count = count + 2;
```

Decrementing is also possible. A `decrement` involves decreasing the value of the variable by a fixed amount.

Incrementing and decrementing are so common in programming that most languages provide shortcuts for them. In RobotC, we can also use

```
int count = 0;
count += 1;
```

to increment by 1 (or any value we choose). Even shorter, we can use

```
int count = 0;
count++;
```

to increment by 1 (and only 1 with this syntax). There are similar operators for decrementing, `-=` and `--`.

4.3 Formatted Output

A common task in programming is to print the value of a particular variable to the display. It is also common for the values of the variables to be labeled with informative strings. For example, when displaying the value of the light sensor on the screen, it might be useful to print,

`Light level: 33`

to distinguish the light level number from, say, the sound level number. Generating the printed statement above requires a mix of literal strings and variables and a knowledge of format codes. **Format codes** are special strings, embedded in larger strings, that act as place holders for variables. The formatted string for the light level output would be

`"Light␣level:␣%d"`

When used with `nxtDisplayString()` as in

`nxtDisplayString(2,"Light␣level:␣%d", LightLevel);`

the `%d` is replaced with the value of the variable `LightLevel` given as the third argument which presumably had a value assigned to it earlier. Most of the display instructions can take up to two variables. For example,

```
nxtDisplayString(7,"Light␣level:␣%d,␣sound␣level:␣%d",
                           LightLevel, SoundLevel);
```

will replace the first `%d` with the value of `LightLevel` and the second `%d` with the value of `SoundLevel`. Because the robot display is not able to fit many characters on a single line, only being able to use two variables in a formatted string is not a significant limitation. In fact, the last example is too wide to fit on the robot's modest display. To get it to fit, we would have to adapt it as

```
nxtDisplayString(6,"Light␣level:␣%d", LightLevel);
nxtDisplayString(7,"Sound␣level:␣%d", SoundLevel);
```

The `%d` format code is useful for printing integer variables. It automatically opens enough space in the output string to fit the value of the variable inside. For the very particular programmer, there is an extra parameter you can use with `%d` to force it to open up a fixed amount of space for the value of the variable no matter how many digits it might contain. The syntax is `%nd` where `n` is an integer specifying the number of spaces to reserve for the number. This is useful, for example, if the variable to be printed might have 1, 2, or 3 digits. In that case, the format code `%3d` will always reserve 3 spaces for the variable. This is useful when trying to line up a column of numbers. An example is in Listing 4.3.

```
task main() {
 int n1, n2, n3; n1 = 100;
 n2 = -3; n3 = 24;
 nxtDisplayString(0,"␣MotorA:␣%3d", n1);
 nxtDisplayString(1,"␣MotorB:␣%3d", n2);
 nxtDisplayString(2,"␣MotorC:␣%3d", n3);
}
```

Listing 4.3: Formatted output. This snippet will print the numbers neatly in a right justified column.

A word of caution, if there is not enough space reserved to hold the value of the variable, then the space will be expanded to accommodate, potentially ruining any carefully crafted formatting. The `%d` format code is for integer-type variables. The `%f`, `%c`, and `%s` codes are used with float-, character-, and string-type variables respectively. Additional information on format codes can be found in the RobotC On-line Support on the left side-bar under `NXT Functions` → `Display` section.

String variables are special because they do not need format codes to be included in formatted output. Instead, the formatted output string can be constructed using the addition operator, `+`. When two strings are added together, the result is a third string that is the **concatenation** of the original strings reading from left to right. Consider the snippet of instructions in Listing 4.4. Here we do not need a format code in order to include the string variables in the formatted output. Instead, we build the string to be displayed by "adding" together existing strings so that it appears the way we want. In this case, we want to display

```
string first = "Grace";
string last = "Hopper";
nxtDisplayString(0,first + "␣" + last);
wait1Msec(1000);
```

Listing 4.4: Concatenation of strings and the string addition operator, +.

the first name followed by the last name with a space character in between. Adding the three pieces together in the correct order, left to right, gives us the desired result.

Another important distinction between strings and character datatypes is how literal values are recognized by the compiler. A literal string must be surrounded by double-quote characters, ". A literal character must be surrounded by apostrophe characters, '. Sometimes using one when we mean to use the other will cause an error. Unfortunately, sometimes those kinds of errors are not detected by the compiler and only manifest as misbehavior when the program is executed.

4.4 Using Sensors

Listing 4.5 shows a program that will display the value of the light sensor. In order to write the program in Listing 4.5, the programmer must "declare" a light sensor variable. Fortunately, RobotC makes it easy to do this. Just select the menu item Robot→ Motors and Sensors Setup. In the resulting window, select the Sensors tab. There are 4 lines labeled S1, S2, S3, and S4. Each line corresponds to the ports on the NXT brick labeled 1, 2, 3, and 4 respectively. With the light sensor plugged into Port 1, we fill in the S1 line. In the first field, we give a variable name to the sensor, e.g. `LightSensor`. Any name may be used as long as it follows the variable naming rules of Section 4.1. Next we choose the sensor type: `Light Inactive`. (`Light Active` uses the same sensor, but in a mode where a small light is turned on to aid in low-light situations.) When completed, RobotC will insert the sensor declaration line as seen at the top of Listing 4.5.

```
#pragma config(Sensor, S1, LightSensor, sensorLightInactive)

task main() {
 int LightLevel = 0;
 LightLevel = SensorValue[LightSensor];
 nxtDisplayString(3,"Light␣level:␣%d", LightLevel);
 wait10Msec(300);
}
```

Listing 4.5: This example will take a single reading from the light sensor and display the value in a formatted string. Notice the instruction at the top of the program inserted by RobotC to declare the light sensor.

A light sensor reading is of integer-type. The current value of the light sensor is always

available in `SensorValue[LightSensor]` (this is actually one element of a special array of values that we will discuss later). In the square brackets, we find the variable name that we have declared for the light sensor. For simplicity and readability, we store the light sensor reading in the integer-type variable `LightLevel` (an indeterminate value). We use this variable to display the light level in a formatted string.

4.5 Functions

The built in functions of RobotC, like `nxtDisplayString()`, are able to perform many tasks. For a number of reasons, however, it is desirable to construct new functions out of the existing functions. A **function** is a self-contained collection of instructions that, when called, accepts a specified number of arguments, performs some task, and returns a value back to the calling program. There are three main reasons for writing new functions:

1. Procedural abstraction – allows the programmer to think of a program as a collection of sub tasks and to focus on making each sub-task work correctly before "gluing" them together to accomplish the complete task.

2. Reusability – allows the programmer to write and debug a set of instructions that can be used over and over in different contexts rather than re-writing the same set of instructions in numerous places.

3. Readability – even if reusability is not an issue, in large programs it is often desirable to "encapsulate" instructions for a particular sub task into an aptly named function. Doing so makes the main part of the program a sequence of function calls that is quite readable. Encapsulation also helps to isolate programming errors. Once a program exceeds one hundred lines or so, the programmer should consider using functions to encapsulate related sections of instructions for the sake of readability.

Function syntax is similar to variable syntax in that a function must be defined before it is called. The scoping rules of a function, the places in the program where the name of the function is recognized, are the same as for variables. The naming rules for functions also match those of variables. The general syntax for a function declaration is

```
// function comment
[return type] [name]([argument list]) {
 // body block, instructions to carry out
 // function's task
}
```

Like variable declarations, function definitions may occur anywhere, but tradition has the programmer put all of the function definitions at the top of the program before the start of the main program block. Tradition also dictates that a descriptive comment above each declaration describe the purpose, inputs, and outputs of the function.

```
// renders a simple 10 by 10 smiley face
// on the display centered about the
// coordinates (x,y), no return value
void DisplaySmiley(int x, int y) {
 nxtDrawEllipse(x-5,y+5,x+5,y-5);
 nxtSetPixel(x-2,y+2);
 nxtSetPixel(x+2,y+2);
 nxtSetPixel(x,y-2);
 nxtSetPixel(x-1,y-2);
 nxtSetPixel(x+1,y-2);
 nxtSetPixel(x-2,y-1);
 nxtSetPixel(x+2,y-1);
 return;
}
```

Listing 4.6: A function that draws a 10×10 pixel smiley face centered at the coordinates `(x,y)`.

Consider, for example, the carefully crafted instructions in Listing 4.6 that use RobotC commands to create a smiley face on the display.

The obvious advantage of this function is that numerous smiley faces can be positioned around the display area, but the work of how to render the smiley face relative to the center coordinates `(x,y)` only needs to be done once. The descriptive name `DisplaySmiley` helps with readability. This function accepts two integer-type arguments. The dummy variables `x` and `y` can be replaced with any integer variable in the calling program or with literal integers. The `return;` statement signifies the end of the function and causes the program execution to return to the calling program just after the function call. In this example, the return statement is not necessary, but it is good practice to include it regardless.

```
// computes the mean of its two arguments,
// returns the mean
float Mean(float a, float b) {
 return (a+b)/2;
}
```

Listing 4.7: A function that accepts two float-type arguments, computes their mean, and returns the result to the calling program.

Consider Listing 4.7 which is an example of a function that computes the average of its two arguments and returns the result to the calling program. In the calling program, the function call is literally replaced by the return value. For example, in the calling program, the instruction

```
average = Mean(w1,w2);
```

assigns the mean of the numbers `w1` and `w2` to the variable `average`. During execution, the program recognizes that in order to perform the assignment, it must first determine the value of the right-hand side. All expressions and functions on the right-hand side are evaluated and reduced to a single value *before* the assignment occurs.

4.5.1 Local variables

The body block of a function is just like the main program in that any instructions can be used therein. If new variables are needed within the function's body block to perform the function's task, they may be declared. Variables declared inside a function's body block are called **local** variables. Their scopes follow the ordinary scoping rules of all variables. Local variables pass out of scope when their declaring functions end.

4.5.2 Pass-by-value vs. pass-by-reference

By default, the variables declared in the argument list of a function definition are local variables to that function. When a function is called, its arguments are copied into the argument variables. The original variables, in the calling program, are not affected by any changes made by the function to its argument variables. This style of argument passing is **pass-by-value**.

Alternatively, the programmer may wish for changes made by the function to one or more of its arguments to be reflected in the corresponding variables of the calling program. To accomplish this, when the variable is passed into the function through one of its arguments, the copy process is skipped and the function has access to the actual variable as it exists in the calling program. This style of argument passing is **pass-by-reference**. The syntax to inform the compiler what style of argument passing is desired is to simply precede the argument name with a `'&'` character.

Pass-by-reference is particularly useful when the programmer wishes to have a function return more than one piece of information to the calling program. For example, suppose in our `Mean()` function we wished, not only to have the mean returned to the calling program, but also to have the function give us the sum of the two arguments. A simple return statement is unable to return more than one value. To get around this, consider the function in Listing 4.8.

Compare this to the function in Listing 4.7. The return type is now `void` and there are two pass-by-reference arguments. Observe the use of the `'&'` character in the second two arguments to indicate that these arguments are pass-by-reference. Listing 4.9 shows how to use this function to get both the sum and the mean of the first two arguments.

The main program declares 4 float-type variables. The `x` and `y` variables hold the numbers to be summed and averaged. Notice that we also need variables, `s` and `m`, to hold the results of the `MeanSum()` function. As the instructions of the main program are carried out from top to bottom, *before* the call to `MeanSum()`, the variables `s` and `m` are empty (**uninitialized**). After the call to `MeanSum()`, they contain the sum and mean respectively. Only after `MeanSum()` has been allowed to do its work, can we display the contents of `s` and `m`. If we mistakenly try to display the contents of `s` and `m` before the call to `MeanSum()`, the results are unpredictable– we may just see a zero, or we may see some random number.

```
// computes the sum and the mean of its
// first two arguments, the sum is placed
// in the third argument, the mean is
// placed in the fourth argument
void MeanSum(float a, float b, float &sum,
             float &mean) {
 sum = a+b;
 mean = (a+b)/2;
 return;
}
```

Listing 4.8: A function that accepts two pass-by-value float-type arguments and two pass-by-reference arguments, computes the sum and mean, and puts the results into the pass-by-reference arguments.

```
task main() {
 float x, y, s, m;
 x=3.4; y=2.8;
 MeanSum(x,y,s,m);
 nxtDisplayString(0, "␣Sum␣=␣%5.2f", s);
 nxtDisplayString(1, "Mean␣=␣%5.2f", m);
 wait10Msec(200);
}
```

Listing 4.9: A main program that uses the `MeanSum()` function and displays the results.

More on RobotC function syntax can be found here[2].

[2]http://carrot.whitman.edu/Robots/PDF/Functions.pdf

4.6 Exercises

1. Which of the following variable names are valid?

   `n`, `2n`, `tax2`, `SpeciesType`, `sales tax`, `dog#`, `?89`

   If invalid, why?

2. Suppose a programmer wrote a program that, when executed, displayed a message on the screen indicating whether the touch sensor was depressed or not. What is the indeterminate in the program?

3. Suppose we have an integer-type variable, `count`. Give two different RobotC expressions that decrement `count` by 2.

4. Do some experiments (run little test programs), to determine what happens when a character-type variable is decremented. How about incremented? Give your observations and speculate about what might be going on.

5. Write a program that reads a value from the sonar sensor and displays the value graphically as a horizontal line across the center of the screen that starts at `(0,31)` and ends at `(x,31)` where `x` is the value of the sonar sensor. The idea is that the length of the line indicates the magnitude of the sonar reading.

6. Write a program that reads a value from the light sensor and displays the value graphically as a filled circle with center `(50,32)` and radius `x`, where `x` is the value of the light sensor. The idea is that the size of the circle indicates the magnitude of the light reading.

7. Write a program that starts, waits 1 second, reads the light sensor, displays the value for 1 second, and exits.

8. Write a program that starts, waits 1 second, reads the sonar sensor, displays the value for 1 second, and exits.

9. Write a program that implements a function called

   ```
   displayLightSensor()
   ```

   that accepts one argument called `wait`. The function will read the light sensor and display its value for `wait` seconds before exiting. Use the function to write a program that reads the light sensor and displays its value three separate times, pausing 1.5 second between reads.

10. Write a program that displays an uninitialized variable. What happens when you run the program?

11. Modify the `DisplaySmiley()` function in Listing 4.6 so that it shows a frown instead. Use it to animate a bouncing frowny face.

12. Write a function that displays an equilateral triangle with center (x, y), side length a, and orientation θ. You may assume that the side length is in pixels and that orientation is in degrees.

Chapter 5

Decisions

In Section 4.1, we discussed the notion of indeterminacy–values in programs that are not known to the programmer when they write the program. When dealing with indeterminate information, we must have a method of making decisions. In real life, decisions are complicated, based on incomplete information and thousands of variables. In computer science, decisions are more straightforward and ultimately result in one of two possible choices. Because of this clear dichotomy, programming languages base decision making on Boolean algebra–the study of true and false statements. This explains why one of the basic datatypes summarized in Table 4.1 is Boolean.

5.1 Boolean Algebra

In ordinary algebra, we study variables that take on any real value. We study the behavior of these variables as they interact using the ordinary operations of arithmetic, namely addition, subtraction, multiplication and division. In Boolean algebra, the variables can have only two values, true or false. Further, the operations are no longer those of arithmetic, but rather those of symbolic logic and are called conjunction, disjunction and negation.

In RobotC, we can declare a pair of Boolean variables as

```
bool p,q;
```

and we can assign values to them as

```
p=false; q=true;
```

In much the same way that we can combine integer- and float-type variables using the ordinary arithmetic operations, we can combine boolean-type variables with Boolean operations. The Boolean operations are defined as follows.

5.1.1 Conjunction

Also known as the "and operator", **conjunction** in RobotC is represented by `'&&'`. Two Boolean values are combined syntactically as

```
p && q;
```

&&	true	false
true	true	false
false	false	false

Table 5.1: Conjunction, the "and operator".

||	true	false
true	true	true
false	true	false

Table 5.2: Disjunction, the "or operator".

and the result is a new Boolean value. The result depends on the values of `p` and `q` and follows the rules outlined in Table 5.1.

5.1.2 Disjunction

Also known as the "or operator", **disjunction** in RobotC is represented by `'||'`. Two Boolean values are combined syntactically as

```
p || q;
```

and the result is a new Boolean value. The result depends on the values of `p` and `q` and follows the rules outlined in Table 5.2.

5.1.3 Negation

Also known as the "not operator", **negation** in RobotC is represented by `'~'`. Unlike the conjunction and disjunction operators, the negation operator only acts on a single boolean value as

```
~p;
```

and the result is a new Boolean value that is simply the opposite of the value of `p`. If `p` is `false`, then `~p` is `true`. If `p` is `true`, then `~p` is `false`. Additional information about Boolean algebra in RobotC is available here[1].

5.1.4 Boolean Expressions

It is surprising how complicated Boolean algebra can get given the simple and limited nature of its variables and operations. A **Boolean expression** is any combination of Boolean variables and operations that can be evaluated to a Boolean value if all the values of the

[1] http://carrot.whitman.edu/Robots/PDF/Boolean%20Algebra.pdf

Syntax	Description
>=	greater than or equal, evaluates to `true` if the left-hand side is greater than or equal to the right-hand side, `false` otherwise.
>	greater than, evaluates to `true` if the left-hand side is greater than the right-hand side, `false` otherwise.
==	equal to, evaluates to `true` if the left-hand side is equal to the right-hand side, `false` otherwise.
<=	less than or equal, evaluates to `true` if the left-hand side is less than or equal to the right-hand side, `false` otherwise.
<	less than, evaluates to `true` if the left-hand side is less than the right-hand side, `false` otherwise.
!=	not equal to, evaluates to `true` if the left-hand side is not equal to the right-hand side, `false` otherwise.

Table 5.3: The comparison operators.

variables are known. For convenience, Boolean expressions may also include parentheses to control the order of evaluation. Negation takes precedence over conjunction and disjunction. In the case of ties, expressions are evaluated from left to right. Consider the snippet of instruction in Listing 5.1.

```
bool p,q,r,s;
p=true; q=false; r=true;
s=~(p||q) && (q||r) && (r&&p);
```

Listing 5.1: A compound Boolean expression. At the end of the block, `s` has the value `false`.

5.2 Comparison Operators

Now that we have an understanding of Boolean algebra, it is important to note that in programming we rarely construct expressions comprised solely of Boolean variables like that of Listing 5.1. Instead, we usually construct Boolean expressions that arise by comparing variables of the other types. In RobotC there are 6 operators designed to compare values and return Boolean values. They are: greater than or equal, greater than, equal, less than, less than or equal, and not equal. Each has its own syntax summarized in Table 5.3

Be wary of the == operator! A common programming error is to use the assignment operator, =, to compare values. This error is exacerbated by the fact that, because the mistaken syntax actually makes sense to the compiler (a fact that we will discuss later), it will not cause a compiler error.

Listing 5.2 shows how to use comparison operators. It shows the common task of testing whether a variable lies inside a certain range. The expressions in parentheses evaluate to either `true` or `false` depending on the values of the variables `x`, `y`, and `z`.

```
float x=5.2, y=0.0, z=10.0;
bool s;
s = (x>=y) && (x<=z);
```

Listing 5.2: An example of using comparison operators. The value of `s` at the end of the snippet is `true`. This shows how a programmer would test if $y \leq x \leq z$.

It is interesting to note that the comparison operators also work on string and character values using alphabetical order. When comparing two strings/characters, whichever comes first in the dictionary is the smaller of the two. String/character comparisons are case-sensitive with the rule that capital letters are less than their corresponding lower-case letters.

5.3 Conditional Statements

Now that we have the ability to create Boolean expressions, we introduce conditional statements. A **conditional statement** allows a block of instruction to be executed depending on the value of a Boolean expression.

5.3.1 If-statements

An if-statement is a block of instruction that is executed only if its predicate is true. The **predicate** is the Boolean expression that controls whether or not the block of an if-statement is executed. If the predicate is true, then the block will be executed, if it is false then the block will be skipped and program execution will resume after the closing brace of the block. The syntax, given in Listing 5.3 is simple and quite readable.

```
if( [predicate] ) {
 // conditional block
}
```

Listing 5.3: The syntax of an if-statement. The predicate, a Boolean expression, determines whether the succeeding block of instruction is executed.

5.3.2 If-else statements

A straightforward extension of the if-statement is the if-else-statement. In an if-else-statement the value of the predicate determines which of two blocks of instructions is executed. The syntax is summarized in Listing 5.4.

```
if( [predicate] ) {
 // conditional block executed if
 // the predicate is true
}
else {
 // conditional block executed if
 // the predicate is false
}
```

Listing 5.4: The syntax of an if-else-statement. If the predicate is true, the first block is executed, otherwise the second block is executed.

More on the if-else statement can be found here[2].

In Listing 5.5, we test the value of the sonar sensor and display different messages depending on the distance measured by the sensor at run time.

```
#pragma config(Sensor, S1, Sonar, sensorSONAR)

task main() {
 int distance = 0;
 distance = SensorValue[Sonar];
 nxtDisplayString(3,"Sonar:␣%d", distance);
 if (distance > 50) {
  nxtDisplayString(4,"␣Come␣closer,");
  nxtDisplayString(5,"␣I␣can't␣see␣you!");
 }
 else {
  nxtDisplayString(4,"␣Back␣off␣man!,");
 }
 wait10Msec(300);
}
```

Listing 5.5: An example of an if-else-statement. If the distance measured by the sonar is greater than 50cm, then the first message is displayed. If it is less than or equal to 50cm, then the second message is displayed. The instruction at the top "declares" the sonar sensor and was inserted by RobotC.

The current value of the sensor is an integer and is always available in `SensorValue[Sonar]` (this is actually an array element, we will discuss arrays a little later). For convenience and readability, we copy the current value of the sonar sensor into the integer-type variable, `distance`. For the sake of the example, we arbitrarily decide that if the sonar reading is more than 50cm, then the target is too far away and if it is less than or equal to 50cm, then it is too close. We use the if-else-statement to display different messages depending on this

[2]http://carrot.whitman.edu/Robots/PDF/Decision%20Making.pdf

condition. The predicate in this case is `(distance>50)`. The value of the predicate depends on the value of the indeterminate, `distance`.

5.4 Mathematical Expressions

Numeric variables and literals can be combined using the infix style of mathematical expressions. Infix is the method of expression in which the mathematical operation is placed between the values upon which it acts (as opposed to prefix or postfix). This is the style common in most TI calculators.

5.4.1 Basic arithmetic

RobotC recognizes the usual mathematical symbols: `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). In addition, RobotC recognizes the use of parentheses for grouping in mathematical expressions. RobotC also recognizes a number of more advanced mathematical functions like sine, cosine, logarithms and the exponential function. Some of those additional functions are summarized in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Math` section.

5.4.2 Integer arithmetic

The basic arithmetic operations are straightforward and intuitive in most cases. However, when working with integer datatypes there is an exception. The `/` operator (division) automatically detects when it is operating on a pair of integers and, in that case, switches to whole number division. **Whole number division** returns an integer value that represents the number of times the denominator goes into the numerator, dropping any remainder. For example, the expression `3/2` in RobotC evaluates to `1` not `1.5`. The expression `-1/2` evaluates to `0`. The expression `33/10` evaluates to `3`.

The `/` (division) operator only performs whole number division if *both* the numerator and denominator are integer type variables or literals. In all other cases, ordinary division is used. To force ordinary division of two integers either include a decimal point if it is a literal value, e.g. `3/2.0` instead of just `3/2`, or convert the integer variable to a float, e.g. `((float)n)/2` instead of `n/2`.

If we want the remainder after whole number division, there is separate operator for integers, `%` (modulus) that returns an integer value that represents the remainder after dividing the left-hand side by the right-hand side. For example, `3%2` evaluates to 1, `33%10` evaluates to 3.

Together these operators provide powerful methods of manipulating integer values.

5.4.3 Exponentiation

A curious omission from this collection of mathematical functions is the exponentiation function for computing quantities like 2^3 or $10^{0.33}$. However, we have the tools necessary to build our own. The C programming language uses the function, `pow()`, to perform

exponentiation. For example, 2^3 = `pow(2,3)`, and $10^{0.33}$ = `pow(10,0.33)`. If we have need of exponentiation, we simply take advantage of the properties of logarithms base e and the exponential function, e^x. Recall that

$$\log_e(x^a) = a \log_e x.$$

Also, recall that $\log_e(e^x) = x$. Combining these two properties, we have

$$x^a = e^{a \log_e(x)}.$$

In RobotC, e^x = `exp(x)`, and $\log_e(x)$ = `log(x)`. Therefore, the function in Listing 5.6 will give us the standard C exponentiation function.

```
float pow(float x, float a) {
 return exp(a*log(x));
}
```

Listing 5.6: The standard C language exponentiation function built of available RobotC functions.

5.4.4 Randomness

Another important function is the random number generator, `random()`. Randomness is important in computer science for the purpose of running realistic simulations, for security, and for introducing unpredictability in game play.

Each time the `random()` function is called, it returns a positive integer between `0` and its single argument. For example, `random(10)` will return a number between 0 and 10 inclusively each time it is called. If we only wanted a random number between 1 and 10, we would use the expression `random(9) + 1`. The expression `random(100)-50` will return an integer between `-50` and `50` inclusively. The maximum range of the `random()` function is `32767`.

To generate random float type values between 0 and 1 inclusively, we can use the expression `random(32767)/(float)32767`. Here we use the maximum possible range so that we get as many possibilities between `0` and `1` as we can.

Since computers are completely deterministic, getting randomness can be difficult. In many programming environments (not RobotC) careful analysis of the `random()` function will show that it generates the same sequence of random numbers every time you restart your program. To change the sequence, programmers must seed the random number generator with some externally obtained (and hopefully) random number. The **seed** of a random number generator is the number that the generator starts with when applying its randomness formula for computing the next random number. To set the seed, we call the `srand()` function with an integer argument, just once, at the beginning of the program. Afterwards, the `random()` function will generate a sequence of random numbers based on the seed.

Fortunately, robots have lots of external sources for seeds. The programmer could read the sound sensor and use that value as the seed before proceeding. The sequence of random

numbers would then depend upon the sound level at the time the program was started. In a noisy room, this would be a good source of randomness. However, in a quiet room, the reading may be the same each time, which gives the same random sequence each time the program runs. To get around this, there are lots of possibilities. The programmer could read numbers from several sensors and mix the values together in a formula. A very reliable random seed is value of the system clock when the program is executed. With this method, each time the program runs, a different seed, based on the system clock, will be used and, in turn, a different sequence of random numbers will be generated. The reserved variable, `nSysTime`, contains an ever-changing integer that represents the number of elapsed milliseconds since the brick was powered up. More details about system time are available in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Timing` section. This method is so reliable, in fact, that the RobotC developers decided to make it the default behavior, so seeding will not be an issue in RobotC–lucky us. Years ago developers as Silicon Graphics Inc (SGI), desperate for good random seeds, used live images of lava lamps to generate truly random seeds[3].

[3]http://en.wikipedia.org/wiki/Lavarand

5.5 Exercises

1. What is wrong with the expression `s=(y<=x<=z);`? What is the programmer trying to do? How would you fix it?

2. What is wrong with the snippet of instruction below? What is the programmer trying to do? How would you fix it?

```
if(LightValue = 50) {
 nxtDisplayString(4,"The␣light␣is␣just␣right!");
}
```

3. Re-write the program in Listing 5.5 using a different sensor, different condition, and different messages.

4. Write an if-statement with a predicate that is:

   (a) true, if the light sensor value is between 20 and 80 inclusive, and false otherwise.

   (b) true, if the sonar sensor value is strictly less than 10 **and** the touch sensor is depressed, and false if either of these conditions is not satisfied.

   (c) false, if the sound sensor value is greater than or equal to 20 **or** the touch sensor is not depressed, and true in all other cases.

5. Given the available mathematical functions, write a function that implements the tangent of an angle given in radians, `tan()`.

6. Write an if-else statement that executes the if-block when the integer-type variable `n` is even and the else-block otherwise.

7. Write an expression that returns a random integer between -50 and 300 inclusively.

8. Write an expression that returns a random float between 0 and 100.

9. Write a function called `CoinFlip()` that returns either `true` or `false` (return type `bool`) with a 50-50 probability.

Chapter 6

Loops and Arrays

A natural extension of the if-statement, which conditionally executes a block of instruction, is the loop, which repeats a block of instruction until condition is met. There are two kinds of loops available in RobotC, the while-loop and the for-loop. As we will see, they have the same functionality, but for the sake of readability, use different syntaxes.

6.1 While-loops

A while-loop, like an if-statement, has a predicate followed by a block of instruction. Unlike the if-statement, however, at the end of the block, the program execution returns to the top of the loop, the predicate is checked again and if it is true, the block is executed again. The block is executed over and over until, for some reason, the predicate becomes false. When the predicate if false, the block is skipped and the program resumes at the line after the closing brace, `}`, of the block.

A while-loop can run once, 10 times, an infinite number of times or not at all depending on the behavior of the predicate. The syntax is given in Listing 6.1.

```
while( [predicate] ) {
 // block of instruction
}
```

Listing 6.1: The syntax of a while-loop. The block of instruction is executed until the predicate becomes false.

Listing 6.2 shows a program that shows the light sensor reading. The program exits when the touch sensor is depressed. Notice that the touch sensor value, a Boolean-type, can be used as the predicate of the while-loop (in this case negated). Notice also the use of the `wait10Msec()` function to delay the loop execution each time. Without this function call, the display is updated so quickly that the light meter number flickers and is hard to read.

In Listing 6.3, a while-loop checks the touch sensor over and over. When the touch sensor is pressed, the loop exits and the "balloon" is popped.

The most useful aspect of a while-loop is that its duration is indefinite. On the flip side, the most aggravating aspect of a while-loop is that its duration is indefinite (possibly

```
#pragma config(Sensor, S1, trigger, sensorTouch)
#pragma config(Sensor, S2, light, sensorLightInactive)

task main() {
 while(!(SensorValue[trigger])) {
  nxtDisplayCenteredBigTextLine(2,"%d",SensorValue[light]);
  // slows the loop down so that number does not flicker
  wait10Msec(5);
 }
}
```

Listing 6.2: This program displays the light sensor reading. The while-loop repeatedly polls the light sensor value and displays it. The loop exits when the touch sensor is depressed.

```
#pragma config(Sensor, S1, trigger, sensorTouch)

task main() {
 nxtDrawEllipse(18,63,78,0);
 // loops until trigger is pressed
 while(!(SensorValue[trigger])) {}
 eraseDisplay();
 nxtDisplayBigStringAt(28,40,"Pop!");
 wait10Msec(300);
}
```

Listing 6.3: This program displays a “balloon” that pops when the touch sensor is depressed. The while-loop halts the program until the touch sensor is pressed. The block of the while-loop is trivial.

infinite). A common programming error is to build a while-loop that has no logical end. In other words, the programmer's exit strategy is flawed.

Another purpose of while-loops in the context of robotics is the detection of status changes in sensors. Consider the program in Listing 6.4 which counts the number of times the touch sensor has been pressed. The first inner while-loop, with an empty block, simply puts the program into a "holding pattern" until the trigger is pressed. When the trigger is pressed, a counter is incremented and a second while-loop with an empty block puts the program into a holding pattern until the trigger is released. The outer while-loop has a trivial predicate with literal value `true`. This loop will run forever, or until the battery runs down, or until we press the orange stop button on the NXT brick. This program is nearly always in one holding pattern or the other. Between holding patterns, the counter is incremented (notice too that it has been initialized to zero) and the display is updated.

```
#pragma config(Sensor, S1, trigger, sensorTouch)

task main() {
 int count =0;
 nxtDisplayCenteredBigTextLine(3,"%d",count);
 while(1) {
  // hold as long as trigger is not depressed
  while(!(SensorValue[trigger])) {}
  // increment the trigger count by 1
  count++;
  // hold as long as trigger is depressed
  while((SensorValue[trigger])) {}
  // reset display and display the current
  eraseDisplay();
  nxtDisplayCenteredBigTextLine(3,"%d",count);
  }
}
```

Listing 6.4: This program will count the number of times the touch sensor has been depressed and display the count on the screen.

6.2 Break statements

In addition to the predicate becoming false, a second method of ending a while-loop is the `break` statement. If, during the execution of the while-loop body block, a break statement is encountered, the program execution immediately jumps to the first instruction after the while-loop block–effectively terminating the while-loop. Typically, the break statement should be enclosed in some kind of if-statement. Otherwise, the loop will end on its first iteration.

It is important to note, in the case of nested loops, that the break statement only terminates the inner-most enclosing loop, while outer loops may continue. A break statement is useful if one, or more, conditions arise during the execution of the loop making it necessary to terminate the loop immediately. An example of this is in Listing 6.5. Similar to Listing 6.4, the main loop in Listing 6.5 terminates when the count reaches 20. We accomplish this by inserting an if-statement immediately following the increment of the `count` variable. Only when the value of `count` is 20 will the break statement be executed. This would be useful if we were designing a devices that measured how quickly someone could tap the trigger 20 times.

```
#pragma config(Sensor, S1, trigger, sensorTouch)

task main() {
 int count =0;
 nxtDisplayCenteredBigTextLine(3,"%d",count);
 while(1) {
  // hold as long as trigger is not depressed
  while(!(SensorValue[trigger])) {}
  // increment the trigger count by 1
  count++;
  if (count == 20) { break; }
  // hold as long as trigger is depressed
  while((SensorValue[trigger])) {}
  // reset display and display the current
  count eraseDisplay();
  nxtDisplayCenteredBigTextLine(3,"%d",count);
  }
  count eraseDisplay();
  nxtDisplayCenteredBigTextLine(1,"20␣Taps␣Reached");
  nxtDisplayCenteredBigTextLine(5,"Exiting");
  wait10Msec(300);
}
```

Listing 6.5: This program will count the number of times the tough sensor has been depressed and released and display the count on the screen. When the threshold count of 20 is reached the program exits with an exit message.

Notice that we added a few lines of instruction after the while-loop in order to display an exit message indicating that the 20 threshold has been reached.

6.3 Arrays

Loops allow the programmer to execute multitudes of instruction and manipulate large amounts of information in a controlled way. Suppose we wanted to write a program that

collected light measurements once every 15 minutes for a whole day. Suppose further that at the end of the measurement period we wanted to create a bar graph of the light measurements so that we could visualize the variation of the light measurements throughout the day. To create the bar graph, we would need to be able to retrieve each and every light measurement from beginning to end. That is a total of 96 values.

Typically, the way that we store and retrieve values is by placing them into variables. It would be tedious to have to declare 96 variables in a program, so most programming languages, RobotC included, allow the programmer to declare arrays of variables. An **array** is an indexed collection of variables of the same datatype. **Indexed** means that each variable (also called an **element**) of the array can be accessed using an integer **index**.

The syntax for declaring an array of variables is

```
[datatype] [variable name][[SIZE]];
```

This is just like an ordinary variable declaration, but the variable name is followed by an integer in square brackets indicating the number of elements associated with the array name. In the case of our light measurements, which will be of integer-type, we might use the declaration

```
int light_readings[96];
```

This straightforward declaration quickly gives the programmer 96 variables of the form `light_readings[0]`, `light_readings[1]`, `light_readings[2]`, ..., `light_readings[95]` to work with. Notice that the first element of the array has index `0`, that subsequent element indices each increment by one, and that the last element has index `95`. *Array indices always start at `0` and end with an index one less than the declared size of the array.*

Listing 6.6 shows a snippet of instruction that collects light measurements every 15 minutes for 24 hours.

The program initializes an integer-type counter variable, `count`, to 0, and declares an integer-type array of 96 elements for the light readings. The while-loop, with trivial predicate, takes a reading, increments the counter, and pauses for 15 minutes before repeating. When the array is full, the break statement causes the loop to terminate. Notice that we may use the variable `count` as the array index. It starts at 0 and increments by 1 each time through the loop so that each light reading is assigned to a different array element. Notice also that the break statement occurs after the counter increment, but before the `wait10Msec()` calls.

An **array bound overrun** occurs if you attempt to use an out of range index to access an array element. This common error can be difficult to find because the compiler cannot detect it. Such errors can lead to unpredictable behavior in the same way that uninitialized variables can. Correcting array bound overruns involves careful inspection of all array operations and clearly determining the value of any index variables. For example, in the program in Listing 6.6, if we simply swap the order of the counter increment and the break statement, we will introduce an array bound overrun. Checking the index before incrementing will lead to an attempt to use the illegal value `96` as an array index the next time through the loop.

```
#pragma config(Sensor, S1, lightsensor, sensorLightInactive)

task main() {
 int count =0;
 int light_readings[96];
 while(true) {
  light_readings[count] = SensorValue[lightsensor];
  count++;
  if(count == 95) { break; }
  wait10Msec(30000);
  wait10Msec(30000);
  wait10Msec(30000);
 }
 //
}
```

Listing 6.6: This program uses a while-loop to collect light sensor readings every 15 minutes for 24 hours and store the data in the array `light_readings`. The three calls to `wait10Msec(30000)` are required because the `wait10Msec()` function can only accept arguments that are less than `32767`. To wait 15 minutes requires an argument of `90000` which is too large.

6.4 For-loops

For-loops are syntactically designed to be used with arrays. Though the behavior of a for-loop may be duplicated by a while-loop, for readability, programmers typically use for-loops when the intention of the loop is to manage an array.

```
for( [initialization]; [condition]; [increment]  ) {
 // block of instruction
}
```

Listing 6.7: The syntax of a for-loop. The block of instruction is executed as long as the condition is true.

When a for-loop is first encountered, the `[initialization]` instruction is executed and immediately thereafter the `[condition]` statement. If the `[condition]` statement is true, the for-loop body block is executed. At the end of the block, program execution returns to the `[increment]` statement and immediately thereafter the `[condition]` statement. If the `[condition]` statement is true, the block is executed again and the program execution returns to the `[increment]` statement. The `[initialization]` statement is only executed once when the program execution first encounters the for-loop. Thereafter, the `[increment]` and `[condition]` statements are executed until the `[condition]` statement becomes false.

At first, the for-loop operation seems overly complicated, but consider the code in Listing 6.8 which shows a typical application of for-loop syntax.

```
task main() {
 int i;
 int perfect_squares[10];
 for(i=0;i<=9;i++) {
  perfect_squares[i] = i*i;
 }
 //
}
```

Listing 6.8: A simple for-loop that stores the first 10 perfect squares in the array `perfect_squares`.

In this case, the for-loop controls the value of the the array index, `i`, by initializing it to `0` and incrementing it by `1` through the values `0` through `9`. In the loop body, we assign `i*i` to the `i`th array element. After the loop is finished, the `perfect_squares` array contains $(0, 1, 4, 9, \ldots, 81)$.

Listing 6.9 shows a program that will scroll through a list of the lower-case letters of the alphabet with a quarter-second delay between each letter.

```
task main() {
 int i;
 char alphabet[26];
 for(i=0;i<=25;i++) {
   alphabet[i] = 'a' + i;
 }
 for(i=0;i<=25;i++) {
   nxtScrollText("%c",alphabet[i]);
   wait10Msec(25);
 }
}
```

Listing 6.9: Scrolls through the letters of the alphabet. The first for-loop fills an array with the letters of the alphabet. The second displays them to the screen with a quarter-second delay between each new letter.

Notice the unusual addition of an integer and a character assigned to the `alphabet` array. This is a useful method of manipulating characters via their positions in the alphabet. It works via the notion of casting.

6.5 Two-Dimensional Arrays

A two-dimensional (2D) array is a "grid" of elements. While there is very little that a 2D array can do that an ordinary array cannot, sometimes information is conceptually easier to

represent as a 2D array. For example, the NXT display in Figure 3.1 is naturally suited to a 2D array representation. The syntax for declaring a 2D array is

```
[datatype] [variable name][[SIZE1]][[SIZE2]];
```

More specifically, an array that might represent the NXT display would be

```
bool screen[100][64];
```

Elements that are true indicate pixels that that are on. Elements that are false indicate pixels that are off. To represent a blank screen, we set all of the elements to false

```
bool screen[100][64];
for(i=0;i<100;i++) {
 for(j=0;j<64;j++) {
  screen[i][j]=false;
 }
}
```

To represent a display with a horizontal line across the middle of the display

```
bool screen[100][64];
for(i=0;i<100;i++) {
 screen[i][32]=true;
}
```

Notice that indices start at zero, just like with ordinary arrays. We see a nice correspondence between the pixel coordinate, `(i,j)`, and the 2D array element `[i][j]`. A programmer can make a whole assortment of changes to the screen array and then use it to "paint" the screen all at once by "visiting" every array element and, if true, turning on the corresponding pixel.

```
for(i=0;i<100;i++) {
 for(j=0;j<64;j++) {
  if(screen[i][j]) { nxtSetPixel(i,j); }
 }
}
```

6.6 Exercises

1. Under what circumstances might the program in Listing 6.4 miss a trigger count?

2. Consider the alternative below to the trigger count program in Listing 6.4.

```
#pragma config(Sensor, S1, trigger, sensorTouch)

task main() {
 int count =0;
 nxtDisplayCenteredBigTextLine(3,"%d",count);
 while(1) {
  if(SensorValue[trigger]) { count++; }
  eraseDisplay();
  nxtDisplayCenteredBigTextLine(3,"%d",count);
 }
}
```

   Will it give an accurate trigger count? Why or why not?

3. Write a program that uses the sound sensor to briefly display (1 second) a message (or graphic) when a loud noise is detected.

4. Use the `PlayImmediateTone()`, as detailed in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Sounds` section, to play a tone whose frequency is proportional to the intensity of light being measured by the light sensor. (For added information, display the light measurement on the screen. Also, a small loop delay may be required for this to work well.)

5. Write a snippet of code that declares a 10-element integer array and fills it with random integers between -100 and 100 inclusively.

6. In computer science, a *queue* is a data structure that allows data to be added at the end of the array and removed from the front. Queues are usually stored in arrays. To add a value to the queue simply assign it to the next open array element. This assumes that you, the programmer, are keeping track of how many elements are in the queue with a counter variable like `QueueSize` that you increment when an element is added and decrement when an element is removed. To remove an element, copy all the elements forward in the array, e.g. assign element 1 to element 0, element 2 to element 1, etc.

   Write a for-loop that shifts `n` elements in an array forward, effectively deleting element 0 from the queue.

7. Write a snippet of code that swaps the values in the `i`th and `j`th elements of an array.

8. Write a program that uses the sonar sensor to count the number of times a hand is waved before it in close proximity. Include a break statement that terminates the loop after 20 "waves". (Hint: For this program, you will have to do some preliminary testing to see how the values of the sonar change when a hand is waved in front of it. You will then need to use those explicit values to construct predicates for the holding while-loops.)

9. In Listing 6.6, why is it important that the break statement be placed precisely where it is placed? In your response, consider other placements, e.g. before `count++;` or after the calls to `wait10Msec()`.

10. Write a program that displays a scrolling bar graph of the light sensor values sampled at 100 millisecond intervals.

11. Write a program that tests the reaction time of the user. Users are instructed that when the test starts they are to wait for some visual cue to appear (after some random length of time between 0.5s and 10.0s) and to press the trigger as soon as they see it. The program then reports the reaction time in seconds (to 2 significant digits) along with some (perhaps snarky) comment on the time.

Chapter 7

Motors and Motion

Up to this point, we have only discussed the sensors and the display. Sensors provide information to the robot the environment. The screen provides information to the environment. So far everything has been quite passive. Now it is time to get proactive! The NXT kit contains 3 motors. The motors allow the robot to change its environment–to go from a passive observer to an active participant.

7.1 Motors

The NXT motors are sophisticated devices that not only provide a way to apply force, but also act as rotational sensors–able to measure rotation to the nearest of 360°. Most of the motor control is done through the use of a collection of special reserved arrays.

7.1.1 Motor Arrays

Motors are connected to the NXT brick through the A, B, or C ports only (not the numbered ports). Motors are connected and identified to your program in the same way that sensors are using the "Motors and Sensors Setup" window. It is important that the programmer know which motor is connected to which lettered port.

When motors are connected to the NXT brick and identified to the program, the program maintains two important arrays associated with the motors summarized in Table 7.1.

Array	Description
`motor[]`	speed array, each integer element (one for each motor) ranges from -100 to 100, negative values reverse direction
`nMotorEncoder[]`	encoder array, each integer element (one for each motor) indicates the number of degrees of rotation relative to some zero that the programmer sets

Table 7.1: Two important motor arrays and their purposes.

The `motor[]` array is a 3-element integer array with `motor[0]` corresponding to Port A, `motor[1]` to Port B, and `motor[2]` to Port C. The "Motors and Sensors Setup" window, among other things, sets up meaningful *aliases* for the array indices 0, 1, and 2, to make your program more readable.

At the start of the program, the motor array elements are all 0 (motors off). Assigning a non-zero value between -100 and 100 to a motor array element instantly turns on the motor to that power and in the direction specified by the sign. The motor will remain on, at that power, and in that direction for the duration of the program or until the programmer changes the value.

The `nMotorEncoder[]` array elements have the same correspondence to Ports A, B, and C as the `motor[]` array. Each integer element indicates the number of degrees of rotation of the motor since the beginning of the program or since the programmer last set the element to zero. For example, a value of 360° indicates that the motor has completed one full turn, 720° indicates two turns, 765 indicate two and a quarter turns.

Other motor arrays, as described in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Motors` section, allow the programmer to control more subtle aspects of the motors.

7.1.2 Basic Motor Control

The timing and control of motor actions can be tricky. For example, consider the snippet

```
#pragma config(Motor, motorC, Left, tmotorNormal, PIDControl)

task main() {
 motor[Left] = 50;
}
```

This program will exit immediately leaving no time for the motor to turn. Adding a wait command,

```
#pragma config(Motor, motorC, Left, tmotorNormal, PIDControl)

task main() {
 motor[Left] = 50;
 wait1Msec(1000);
}
```

will cause the motor to run for 1 second at half power.

7.2 Turning and Motor Synchronization

A robot with two drive wheels and a trailing coast wheel (a tri-bot) can be made to turn by driving the two drive wheels at different rates. Consider the circular track in Figure

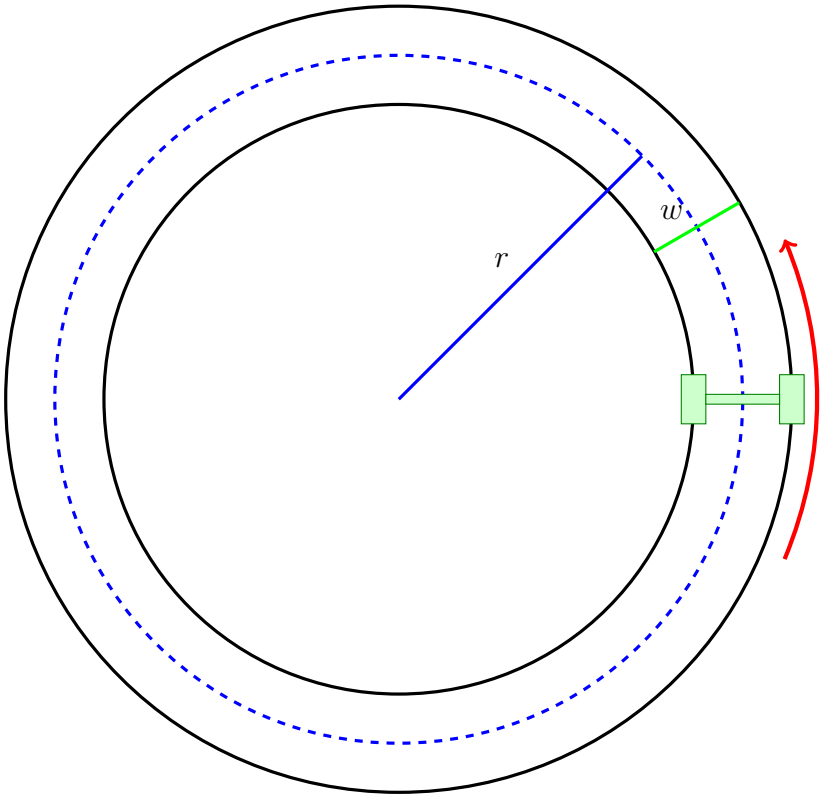


Figure 7.1: A two-wheel drive circular path of radius r and trackwidth w.

7.1 traced out by the two drive wheels of the the tri-bot. A simple geometric calculation determines the relative rates at which the two drive wheels should turn in order to move along this path (the dotted blue line). We assume that the radius of the path is r and the trackwidth (the distance from center of the point of contact of the left wheel with the ground and the right wheel with the ground) is w. Assume that the tri-bot travels around the track in t seconds. The left wheel (inner circle) travels a distance of $2\pi(r-w/2)$–the circumference of the inner circle, while the right wheel travels a distance of $2\pi(r+w/2)$–the circumference of the outer circle. The speeds of the left wheel, s_L, and the right wheel, s_R are given by

$$s_L = \frac{2\pi(r - w/2)}{t}$$

and

$$s_R = \frac{2\pi(r + w/2)}{t}.$$

Consider the ratio of these two speeds

$$\frac{s_L}{s_R} = \frac{r - w/2}{r + w/2}. \tag{7.1}$$

As long as this ratio is preserved, the tri-bot will move along the circular track of radius r.

For example, suppose a tri-bot has a trackwidth of $w = 10$cm and we would like it to move along a circular track of radius $r = 20$cm. Our formula suggests that we run the

motors with a speed ratio of $\frac{15}{25} = \frac{3}{5}$. Of course there are many different power settings for the motors that yield this ratio. In fact, any pair of left and right motor power settings that reduce to this fraction will cause the tri-bot to move along this track of radius 20cm. The only difference will be in how fast the robot moves along the track. A ratio of 15 : 25 will move only half as fast as a ratio of 30 : 50.

RobotC provides convenient commands for controlling a pair of motors. Listing 7.1 shows how to control the relative speeds of the two motors.

```
#pragma config(Motor, motorC, Left, tmotorNormal, PIDControl)
#pragma config(Motor, motorA, Right, tmotorNormal, PIDControl)

task main() {
 nSyncedMotors = synchAC; // Left motor slaved to Right motor
 nSyncedTurnRatio = +60; // Left motor turns 60% of right motor
 motor[Right] = 50; // Right motor moves at 50% power
                    // Left motor automatically moves at 30%
                    // because of synch and synch ratio.
 wait1Msec(1000);
}
```

Listing 7.1: This program will drive a tri-bot with trackwidth 10cm around a circle of radius 20cm.

The left motor is synchronized to the right motor with the instruction, `nSyncedMotors = synchAC;`. The relative speed of the left motor to the right motor is set at +60% (roughly 3 : 5) with the instruction `nSyncedTurnRatio = +60;`. Subsequently, when the right motor is activated with the power 50, the left motor is automatically activated with a power that is 60% of the right's–in this case, roughly 30. With the speed ratio at 3 : 5, the tri-bot will move around a track of radius 20cm for 1 second.

In the previous example, we assumed that the center of the circle about which the robot turned was to the left (or right) of the robot. Suppose the center of the circle is between the drive wheels (under the robot) as in Figure 7.2. This situation occurs when $r < w/2$.

Notice that in this case, the numerator of equation (7.1) is negative indicating that the left wheel turns in the opposite direction of the right wheel. The path of the left wheel is indicated by the inside black circle. The path of the right wheel is indicated by the outside black circle. The arrows show the direction of travel and starting point of each wheel. The dotted blue line shows the path of the midpoint of the distance between the wheels. Additional information on motor synchronization is available here.[1]

7.3 Distance and Motor Encoders

Through synchronization, we now have a method of executing accurate and precise turns, but lack a method of traveling accurate and precise distances. In this section, we continue to

[1] http://carrot.whitman.edu/Robots/PDF/Synching%20Motors.pdf

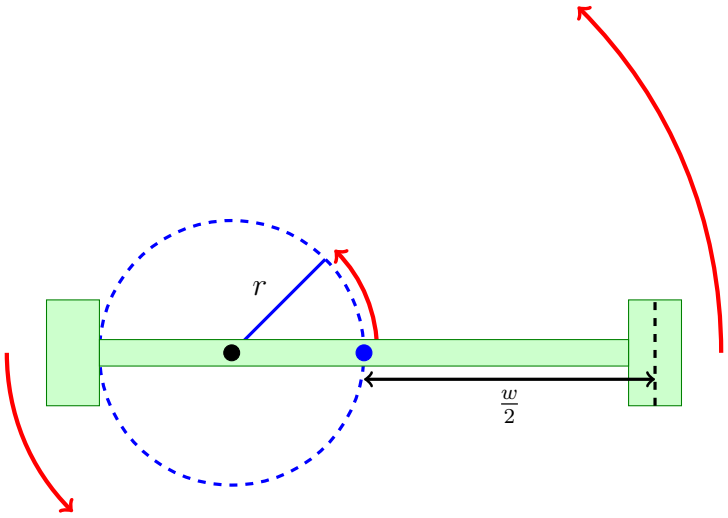


Figure 7.2: A two-wheel drive circular path of radius r and trackwidth w in which $r < w/2$. In this case, the wheels turn at different rates in opposite directions and the blue dot tracks the circle. The center of rotation, the black dot, is *between* the drive wheels.

use the tri-bot model. Controlling distance requires precise information about the effective circumference of the drive wheels and the ability to specify the angle of rotation of the wheels.

7.3.1 Circumference

To determine the circumference of a drive wheel, we could simply remove it, roll it on a piece of scrap paper through one complete rotation, marking the start and end, and measure the distance between them. Or, we could wrap a piece of thread around the wheel and measure its length. Or, we could measure the radius, r, of the wheel and use the circumference formula

$$C = 2\pi r. \tag{7.2}$$

One problem with these approaches is that the wheel has been removed from the context in which it will be used. For example, a weight bearing wheel will not travel as far in one rotation as an unloaded wheel. A more accurate method would be to measure the circumference of the wheel in context. The **effective circumference** is the circumference of the wheel in the context in which it will be used.

To measure the effective circumference, we will use the motor encoders to drive the wheels in a straight line through a set number of rotations and measure the distance traveled. The motor encoders measure the number of degrees of rotation of a motor to the nearest of 360°.

As discussed in Section 7.1.1, the 3-element array, `nMotorEncoder[]`, always contains the current number of degrees the corresponding motor has turned starting since the program started or since the value was last reset to zero. The value is a signed 16-bit integer with the sign indicating the direction of rotation. For example, if the wheel goes forward for one full rotation and then reverses for one full rotation, the motor encoder value for that wheel in the end will be unchanged. Note also that the range of allowable encoder values goes from

-32768 to 32767 degrees. This corresponds to about 91 rotations in either the forward or reverse directions. In long running programs, the programmer should periodically reset the encoder values to zero to avoid an overflow.

Consider the instructions in Listing 7.2. This program will cause the robot to move forward in a straight line through exactly to rotations of the drive wheels. The first two

```
#pragma config(Motor, motorC, Left, tmotorNormal, PIDControl)
#pragma config(Motor, motorA, Right, tmotorNormal, PIDControl)

task main() {
 nSyncedMotors = synchAC; // Left motor slaved to right motor
 nSyncedTurnRatio = +100; // Left right motors same rate
 nMotorEncoder[Right]=0; // Reset right motor encoder to 0
 nMotorEncoderTarget[Right] = 720; // Stops after 720 degs
 motor[Right] = 50; // Right motor moves at 50% power
 while(nMotorRunState[Right]==runStateRunning){} // Hold
}
```

Listing 7.2: This program will cause the tri-bot to move forward in a straight line through exactly two rotations of the drive wheels.

instructions synchronize the motors and cause them to run at the same speed. Next, we set the motor encoder value for the right motor to zero and use the `nMotorEncoderTarget[]` array to set a precise stopping point at 720°. Setting the target does not start the motor, but it does cause the motor to stop when the motor encoder value reaches 720°. We then start the motor at 50% speed and use a while-loop to hold the program execution until the rotations are complete.

It is reasonable to wonder why, in Listing 7.2, we do not omit the encoder target instruction and simply stop the motors by setting the motor power to 0 after the while-loop exits. Indeed, this approach does work. However, particularly at high speeds, stopping in this manner can be quite *rough* causing the robot to rock back and forth and perhaps overshoot (or undershoot) the desired number of rotations.

Setting the `nMotorEncoderTarget[]` causes the motor to slow to a stop at the specified encoder value in a precise way regardless of the motor speed. It should be noted that the target ignores the direction of rotation so that a value of 720° will stop the motors at either ±720°. The sign of the target value does play a role however. Using a negative value disables the precision stopping algorithm and is akin to the rough stopping method alluded to earlier.

There are times when each is appropriate. Using the precise stopping algorithm will drain more power from the batteries in the long run. In the case of attempting to determine the effective circumference of a robot's wheels, the precise stopping algorithm is more appropriate.

7.3.2 Precise Distances

Let us consider Figure 7.1 again. We know how to move along a circular path, but how do we move a specific distance? Suppose we want to travel half-way around the circle. Or, a quarter-way around. Or, 47° around. We can do so using the motor encoder and the effective circumference. Let c be the effective circumference of a tri-bot wheel. Suppose we want to travel through an angle θ (in radians) counterclockwise around the path of radius r as in Figure 7.1. The length, L, of the circular arc of radius r and angle θ is given by the formula

$$L = \theta r.$$

However, to do so the right wheel must travel a distance, L_R, slightly longer

$$L_R = \theta(r + w/2).$$

Hence, the right wheel must rotate L_R/c times in order to move the robot the distance L along the dotted blue path. In terms of degrees, D_R, the encoder target should be

$$D_R = \frac{\theta(r + w/2)}{c} \cdot 360°,$$

where θ is given in radians.

If we are moving clockwise, then we use the left motor to control movement and the same formula applies for computing the encoder target of the left motor.

7.3.3 Named Constants

It is clear that many of the results presented in this chapter will depend upon the exact physical dimensions of the robot. For example, we may have one tri-bot with a trackwidth, $w = 12.5$cm, and second with a trackwidth, $w = 14$cm, and wish to run the same path program on both. To run the program on the second robot, the programmer has to go through the program line-by-line and replace each occurrence of 12.5cm with 14cm. This kind of editing can be tedious and error prone. RobotC, as well as most other languages, provides a way of addressing this issue.

Instead of using the "naked" number, 12cm, throughout the program, a better practice is to declare a global constant at the top of the program file, outside of any program block, and to give the constant a meaningful name. A constant can have any datatype and is declared with the syntax:

```
const [datatype] [CONSTANT NAME] = [value];
```

Putting the `const` keyword at the front of an ordinary declaration causes the variable to be treated as a constant. This means that the compiler will produce an error if the programmer inadvertently tries to change the value of the constant by assignment or passing by reference into a function. Because of this, the value of the constant must be assigned at declaration. The naming rules are the same as for ordinary variables.

In the case of our trackwidth constant, we might use the declaration:

```
const float TRACKWIDTH = 12.5;
```

In this example, we find another good practice, that of capitalizing constant names. This helps to differentiate constants visually from ordinary variables in the program. We see that if the programmer uses a **named constant** throughout a program and places the declaration at the very top of the program file, then it is easy to "port" the program to the second robot with trackwidth 14cm. Changing the `12.5` in the constant declaration to a `14` does the whole job–no searching and replacing throughout the file. Furthermore, using a named constant in place of a naked constant makes a program more readable, e.g. we see `TRACKWIDTH` which means something instead of just `14`.

7.4 Exercises

1. Suppose a tri-bot has a trackwidth of w. Derive a formula for the ratio of motor speeds if the robot is to travel around a circle of radius r in the clockwise direction with its right wheel on the circle.

2. Determine the effective circumference of the drive wheels of your robot. Do so by writing a short program to turn the wheels 2 or 3 rotations and measuring the distance traveled by the robot. Divide the distance by the number of rotations to determine the circumference of the wheels. Do several trials. (Why is it better to do 2 or 3 full rotations and several trials instead of just one?)

   Does the speed at which the motors turn affect your results? What about the coast vs. brake modes?

3. Discuss equation (7.1) in the case when $r = 0$. When $r = w/2$.

4. Use the results from Exercise 2 to write a program that causes your robot to move in a "figure eight" path comprised of two circular paths of radius 20cm placed side-by-side.

Chapter 8

Tasks and Sounds

With the possibility of needing to monitor several input sensors, we introduce the notion of tasks. Each task is an independent program (think `task main()`), and, more importantly, the tasks can run concurrently. Concurrent processes, also known as `threads`, involve forking the program execution into several parallel threads that each run independently.

8.1 Tasks

We have used the task syntax all along when writing our programs. A **task** is an independent program thread designed to be executed simultaneously, or **in parallel**, with the main task. There are many kinds of parallelism in computer science–indeed there are whole books on the topic. Pseudo parallelism, the kind we have in RobotC, is the simplest and oldest form of parallelism. In pseudo parallelism the CPU's computational power, the number of calculations it can do per second, is sliced up according to some prioritization in a process called **time-division multiplexing**.

In a time-division multiplexing scheme, suppose that a CPU can do 100,000 calculations per second and suppose that there are 4 tasks of equal priority and equal computational requirements. Then, each second, the CPU will dedicate 25,000 calculations to each task. Unfortunately, this simple situation rarely occurs.

More often, the tasks, which may include intense loops and leisurely waits, require different amounts of work at different times. When the CPU decides to dedicate some calculations to a particular task it may just so happen that the task is in a wait state and does not require any calculations. A smart allocation of CPU calculations would have the CPU immediately move on to another task rather than wait for the current task to start requiring calculations again. After all, it is possible that the CPU can attend to the other tasks and return the waiting task before it actually requires calculations.

In RobotC, all tasks are given equal priority by default. If two tasks of equal priority require attention, the CPU will work on the task it has been away from longest, but only for a time-slice equal to the task's share. After it receives its allotted calculations, whether it has finished its task or not, the CPU will move to the next task requiring attention. There are advanced prioritization schemes possible with RobotC summarized in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Task Control` section.

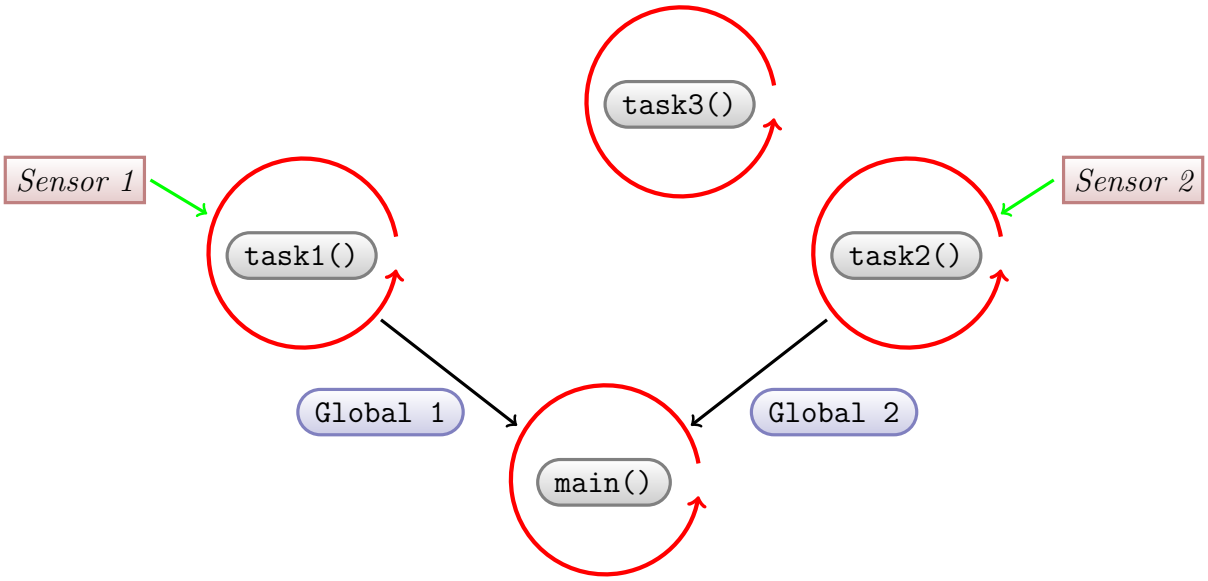


Figure 8.1: Some tasks (`tasks 1` and `2`) might monitor (or poll) sensors independently and rapidly and update global variables. The main program accesses the current sensor information by reading from the global variables. Other tasks (`task 3`), may run completely independently–not communicating with other tasks in any way.

The syntax for writing a new task is

```
task [taskname]() {
 // block of instruction
}
```

and is written outside of the block of any other task, including the main task.

The main task in RobotC is always executed first. Within the main task the programmer may launch a new thread for another task with the `StartTask([taskname])` function. They may end the task with the `StopTask([taskname])` function.

Tasks can communicate with each other through global variables. If a task assigns a value to a global variable, the other tasks all have instant access to the new value. In this way, the programmer can compartmentalize instructions associated with different sensors and motors. Consider the simple program in Listing 8.1.

Here we have two tasks monitoring the light and sonar sensors respectively with while-loops. The tasks continually update the sensor global variables which are then read and displayed by the main task. The main task, executed first, launches the two auxiliary tasks before dropping into a loop to display the light and sonar values.

Listing 8.1 is an example of the more general practice of using tasks to **poll** sensors independently and then using global variables to pass information back to the main program. Figure 8.1 shows the general structure of such a scheme.

```
#pragma config(Sensor, S1, sonar, sensorSONAR)
#pragma config(Sensor, S2, light, sensorLightInactive)

// Global variables.
int LightValue;
int SonarValue;

task MonitorLight() {
  while(true) {
    LightValue = SensorValue[light];
  }
}
task MonitorSonar() {
  while(true) {
    SonarValue = SensorValue[sonar];
  }
}
task main() {
  StartTask(MonitorLight);
  StartTask(MonitorSonar);
  while(true) {
    nxtDisplayString(0,"Light: %3d", LightValue);
    nxtDisplayString(1,"Sonar: %3d", SonarValue);
    wait10Msec(25);
  }
}
```

Listing 8.1: This program has a main task and two auxiliary tasks. Each auxiliary task is committed to monitoring a sensor and updating the sensor value in the associated global variable.

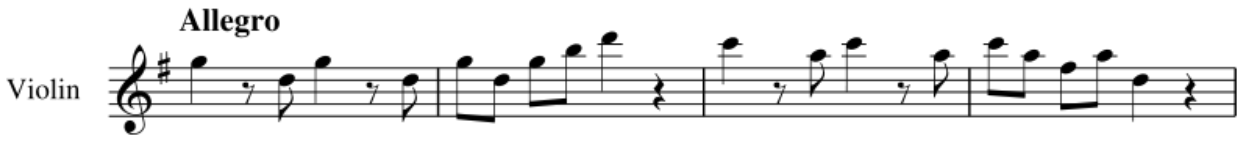


Figure 8.2: *Eine Kleine Nachtmusik*, W.A. Mozart. Using the `PlayTone()` function, the NXT can play this tune.

8.2 Sounds

The NXT brick has a built in speaker to play sounds. Sounds can be both informative and entertaining. Some sounds provide user feedback. As with motor movements, playing a sound takes time. The sound system in RobotC is built around a queue (see Section 6.6, exercise 6). There are several functions, detailed in the RobotC On-line Support on the left side-bar under the `NXT Functions` → `Sound` section, that will cause the NXT to play a sound. If a request for a sound is made before the previous sound is finished playing, the request is put into the sound queue. Each sound in the sound queue is played in the order in which the sound request is made.

8.2.1 Music

Using the `PlayTone()` function and careful timing, it is possible to reproduce simple musical passages on the NXT. The `PlayTone()` function takes two arguments–a frequency followed by a duration in units of 10Msec. The frequencies of musical notes are known constants. Consider, for example, the opening measures of Mozart's *Eine Kleine Nachtmusik* given in Figure 8.2.

The notes of this passage, with octave numbers and R's representing rests, are:

Measure 1:	G_5	R	D_5	G_5	R	D_5
Beats:	2	1	1	2	1	1
Measure 2:	G_5	D_5	G_5	B_5	D_6	R
Beats:	1	1	1	1	2	2
Measure 3:	C_6	R	A_5	C_6	R	A_5
Beats:	2	1	1	2	1	1
Measure 4:	C_6	A_5	$F_5^{\#}$	A_5	D_5	R
Beats:	1	1	1	1	2	2

Since this is originally for violin, we will let the C in the third measure be in the 6th octave. Referring to a list of note frequencies, we arrive at the following list of frequencies and beats for the notes we need.

Measure 1:	783.99	R	587.33	783.99	R	587.33
Beats:	2	1	1	2	1	1
Measure 2:	783.99	587.33	783.99	987.77	1174.66	R
Beats:	1	1	1	1	2	2
Measure 3:	1046.50	R	880.00	1046.50	R	880.00
Beats:	2	1	1	2	1	1
Measure 4:	1046.50	880.00	739.99	880.00	587.33	R
Beats:	1	1	1	1	2	2

In Listing 8.2, we put all of the pieces together.

Notice the use of named constants to represent the notes. Also, notice the use of the named constant `BEAT` to set the duration of notes and rests. Finally, using the reserved boolean variable `bSoundActive`, we can introduce holds to allow notes to finish. The sound queue is limited to just 10 requests, so it is a good idea to wait for the queue to empty at the end of each measure or after 10 requests whichever comes first. While sounds are playing from the sound queue `bSoundActive` is `true`. When all requested sounds are finished playing `bSoundActive` is `false`. Also, `PlayTone()` requests are made instantly, so it is important to wait for tones to finish before rests. Otherwise, the rests will occur while the notes are playing!

8.2.2 Other Sounds

In addition to playing musical notes, it is possible to play other sounds. Using the File Management system, as described in the RobotC On-line Support on the left side-bar under the `ROBOTC Interface` → `NXT Brick Menu` → `File Management` section, it is possible to place sound files onto the NXT brick for subsequent use during programs. Because of memory limitations, these sounds must be short (less than a few seconds). To place your own sounds on the NXT, you must first find a sound file in the `.wav` format. There are many programs available to convert sound files (like `.mp3`) to `.wav` format. A good free program for this purpose is Audacity. You can use Audacity to clip out a section of a larger sound file and save it to a `.wav` format.

RobotC sound files are a special format and have the file extension `.rso`. Once you have created the `.wav` file you are interested in, you must convert it to `.rso` format. A simple, free converter program for Windows is here.

Once you have created your custom sound file, copy it to the NXT using the File Management window, you can play the sound file from within a program using the `PlaySoundFile()` function. Its single argument is the filename of the sound as a string. For example, suppose you have Homer Simpsons famous "Doh!" in a sound file called `doh.rso`, then

```
PlaySoundFile("doh.rso");
```

will cause your NXT to say "Doh!". This is particularly useful if your robot bumps into a wall!

```
// Note Frequencies
const float D5=587.33;
const float G5=783.99;
const float F5Sharp=739.99;
const float A5=880.00;
const float B5=987.77;
const float C6=1046.50;
const float D6=1174.66;
const int BEAT=20; // in 10Msec

task main() {
 // Measure 1
 PlayTone(G5,2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 wait10Msec(BEAT);
 PlayTone(D5,BEAT);
 PlayTone(G5,2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 wait10Msec(BEAT);
 PlayTone(D5,BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 // Measure 2
 PlayTone(G5,BEAT);
 PlayTone(D5,BEAT);
 PlayTone(G5,BEAT);
 PlayTone(B5,BEAT);
 PlayTone(D6,2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 wait10Msec(2*BEAT);
 while(bSoundActive){} //wait for tone(s) to finish
 // Measure 3 and Measure 4 omitted.
}
```

Listing 8.2: This program uses the `PlayTone()` function to play the opening passage of Mozart's *Eine Kleine Nachtmusik*.

8.3 Exercises

1. Write a program that displays a scrolling bar graph of the light sensor values sampled at 100 millisecond intervals. Use the touch sensor to start, pause, and restart the program when the button is depressed.

2. Write a program for a tribot-style robot that will start moving forward at the sound of a loud noise that is detected using the sound sensor and stop after crossing three black lines on the ground that are detected using the light sensor.

3. Write a program for a tribot-style robot that will begin moving forward when the touch sensor is depressed. When the robot gets too close to an obstacle (detected using the sonar sensor), it will emit an exclamatory sound, back up, rotate 90 degrees, and attempt to proceed. It will continue in this fashion until the touch sensor is pressed again.

4. Read the left and right motor encoders to create an Etch A Sketch^{TM}. Implement a way to clear the screen (touch sensor?). Perhaps also implement a way to lift the pen (which you can't do in a real Etch A Sketch^{TM}).

5. Find a well-known piece of music and program your robot to play it using the `PlayTone()` function.

6. Create a custom sound file and write a program that causes your robot to play the sound.

Chapter 9

Files

RobotC on the NXT allows for the creation, reading, writing and manipulation of data files. Data files allow the robot to store data collected during the execution of one program to be used during the execution of a different program.

For example, imagine a simple light sensor program that reads light levels every 15 minutes for a period of one day. We might start the program and leave the robot running for a day in a room with a window. The program would store the light level measurements in a data file and exit after one day has elapsed. Suppose that we return after a few days and would like to see a graph of the light level readings. We turn on the NXT and run a different program that is able to open the data file created by the light sensor program and create a graph of the data. In this way, we can get an idea of what the light level readings were like on the day the measurements were made.

Just like a hard drive or thumb drive, the typical NXT stores a collection of files. The files on the NXT can be viewed from within RobotC using the Robot → NXT Brick → File Management menu described in the RobotC On-line Support on the left side-bar under the `ROBOTC Interface` → `NXT Brick Menu` → `File Management` section.

The File Managment list shows the names of all of the files currently on the NXT. A typical filename has a **name** (up to 15 characters) and an extension (up to 3 characters) with a period separating the two. For example, sound files end with the `rso` extension. Executable program files end with the `rxe` extension.

There are a number of limitations on files other than the 15.3 character limit. The NXT can only hold 64 files in total. If during a project, the 64 file limit is reached, some files will have to be deleted before continuing. During any single program, only 16 files may be open at any one time. Finally, there is a limit to how much information can be stored on the NXT.

9.1 Memory: Bits and Bytes

To understand how much information can be stored in a file on the NXT brick, it is necessary to understand some of the nearly universal units of information storage in computer science. The smallest unit of information is the **bit**–an entity that can be one of two values–typically 0 or 1. A **byte** in our context is defined as a collection of 8 bits. The 8 bits per byte

	←One Byte→							
Places:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Digits:	0	1	1	0	0	0	0	1

Table 9.1: A byte consisting of 8 binary bits. The byte can be interpreted as an 8 digit number expressed in base 2. From right to left there is the ones place, twos place, the fours place etc.

Prefix	Decimal	Binary
kilo-	$10^3 = 1,000$	$2^{10} = 1,024$
mega-	$10^6 = 1$ million	$2^{20} = (1,024)^2$
giga-	$10^9 = 1$ billion	$2^{30} = (1,024)^3$

Table 9.2: The standard prefix multiplier definitions for decimal and binary contexts. Attempts to standardize the use of these prefixes have largely failed. If in doubt, read the documentation carefully.

definition is nearly universal, though in rare instances (usually older computer systems) the byte may be defined as a different number of bits. If in doubt, read your computer system's manual!

A bit can represent one of two different states. A byte can represent one of $2^8 = 256$ different states. Because of this, it is often convenient to interpret the 8 bits in a byte as an integer between 0 and 255. Consider the "byte" in Table 9.1.

To convert this byte to a base 10 integer, we first assign binary place values to each bit position. From right to left, we have the ones position, the twos position, the fours position etc. The corresponding integer is just the sum of the place values that have one digits. In this case,

$$\begin{aligned} 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 &= 64 + 32 + 1 \\ &= 97 \end{aligned}$$

In RobotC, a `char` datatype is represented by a single byte. For the English language character set there is an almost universally agreed upon standard for assigning characters to integers called the ASCII character set. In the case of our example, the byte has a decimal representation of 97. If we look up 97 in the ASCII table, we find that it corresponds to the character `'a'`. Indeed, in the NXT's memory, which is really just an organized collection of bits, the character `'a'` would look just like Table 9.1.

Early computer scientists, in an attempt to use familiar language, adopted the use of the kilo-, mega-, giga- prefixes to describe larger collections of bytes. As a compromise between the traditional meaning of these prefixes and the base two nature of computer science, they extended their definitions as in Table 9.2.

A kilobyte, for example, is not 1,000 bytes, but rather 1,024 bytes. A kilogram, however, is still 1,000 grams. The correct multiplier depends on the context. Typically, in a computer science context, the multiplier is a power of 2. In recent years, marketers of computer

equipment, which one would expect to use the base two multipliers, have instead used the base ten multipliers to make their hardware look better. For example, when purchasing a computer a consumer might wonder: Is a gigabyte of RAM 1,024 megabytes, or just 1,000 megabytes? If the latter, then they may have 24 fewer megabytes than they expected. This abuse of the prefixes has led to widespread ambiguity about the precise meaning of the kilo-, mega-, and giga- prefixes.

There are also some loosely held standards for abbreviating collections of bits and bytes. Typically the word *bit* is abbreviated with a lowercase 'b', whereas *byte* is abbreviated with an uppercase 'B'. The prefix abbreviations are typically capitalized, kilo- (K), mega- (M), and giga- (G). However, it is not uncommon for kilo- to use a lowercase 'k'. A hard disk drive might be marketed as having a capacity of 130GB (130 gigabytes). An internet connection might be marketed as being able to deliver 1.5Mbps (1.5 megabits per second, that's just 0.1875 mega*bytes* per second). And, of course, there is still the question of what giga- and mega- actually mean in these two examples!

The NXT brick has a 32-bit CPU. This means that the CPU can process 32-bits at a time (4 bytes). There are 64KB of RAM and 256KB(0.25MB) of persistent storage. Compared to laptop and desktop computers, this is a small amount of memory. Because the tiny NXT operating system is also stored in the persistent storage, not all of the 256KB is available for data files. As a result, the programmer must be as efficient as possible so as not to exceed the available resources.

Returning to the File Management window, click on one of the sound files and then click the "Info" button. In the text window at the bottom, the size of the file is given in bytes. Indeed, in the file list, the size of every file is given to the nearest tenth of a kilobyte. It is possible to delete files by selecting particular files and clicking the "Delete" button. When trying to free up space, user generated data files, old User Programs (rxe), and Try Me programs (rtm) are good candidates for deletion. All of these types of files can be replaced if you have your original programs saved someplace else.

9.2 Reading and Writing

Files in persistent storage are accessed in a program in one of two modes: read or write. In write mode, data can be written to the file. In read mode, data can be read from the file. A file cannot be in read and write mode simultaneously. The mode is determined at the time the file is opened.

A file in persistent storage is identified by its filename. However, within a program, for the sake of readability, a file is identified by its filehandle. A **filehandle** is a variable that represents the external file. All references to the file are made through the filehandle. A filehandle variable must be declared as type `TFileHandle`. A second special datatype for use with files is the `TFileIOResult` type. A variable of type `TFileIOResult` is capable of holding error messages related to various file operations. For example, if the program attempts to open a file in read mode and the file does not exist, the program will not crash. Instead the error will be reported in the `TFileIOResult` variable. A careful programmer can check the error message and offer the user alternatives to resolve the problem instead of allowing the program to crash.

Function	Description
`OpenRead(TFileHandle, TFileIOResult, string, int)`	opens a file in read mode and attaches it to the first argument, error codes are placed in the second argument, the third argument is the external filename, the fourth argument is the size (bytes) of the file.
`OpenWrite(TFileHandle, TFileIOResult, string, int)`	opens a file in write mode and attaches it to the first argument, error codes are placed in the second argument, the third argument is the external filename, the fourth argument is the size (bytes) of the file, if the file already exists, the function will fail.
`Close(TFileHandle, TFileIOResult)`	closes any file (read or write) and detaches it from the first argument, error codes are placed in the second argument, files not closed may be corrupted and data lost.
`Delete(string, TFileIOResult)`	deletes the file named in the first argument from the NXT brick, error codes are placed in the second argument, useful for erasing a possibly pre-existing file before opening it in write mode.

Table 9.3: A summary of important file handling functions.

A summary of the various file operations is in the RobotC On-line Support on the left sidebar under the `NXT Functions` → `File Access` section. *(Note: this page does not provide documentation for the `TFileHandle` and `TFileIOResult` datatypes described above.)* Table 9.3 shows a list of available RobotC commands for accessing files. Each function is listed along with the expected arguments and their types.

Table 9.4 shows a list of available RobotC commands for accessing files. Each function is listed along with the expected arguments and their types.

As we can see from the tables of commands, it is necessary to know how many bytes are required to store a variable of a specific datatype. Table 9.5 shows the commonly used datatypes and the number of bytes each uses.

Listing 9.1 shows a program that records light levels at 15 minute intervals for one day and stores the readings in a file called `lightreadings.dat`. Notice that we declare the filehandle variable and file error variable but never assign values to them explicitly. Instead they are used by the `OpenWrite()`, `WriteShort()`, and `Close()` functions and we never know their values. To determine how large, in bytes, the file must be, we refer to Table 9.5 and recall that `SensorValue[Light]` is of type `int` which is 2 bytes. We will write 97 readings for a total of 194 bytes. The outer for-loop reads the light sensor, writes the value to the output file, and waits 15 minutes before repeating. When the for-loop exits, the output file is closed. Notice the use of named constants to improve readability.

Listing 9.2 is a companion program to Listing 9.1. This program opens the file of light readings and displays a bar graph of the readings as a simple bar graph. Each sample is read from the file into the temporary variable `n`. The value is then used to create a vertical line to represent the value in the bar graph.

Function	Description
`ReadByte(TFileHandle, TFileIOResult, char)`	reads the next byte from the file and places it into the third argument.
`ReadShort(TFileHandle, TFileIOResult, int)`	reads the next 2 bytes from the file and places it into the third argument.
`ReadFloat(TFileHandle, TFileIOResult, float)`	reads the next 4 bytes from the file and places it into the third argument.
`ReadString(TFileHandle, TFileIOResult, string)`	reads the next several bytes corresponding to string data from the file and places it into the third argument.
`WriteByte(TFileHandle, TFileIOResult, char)`	writes the third argument to the file as 1 byte.
`WriteShort(TFileHandle, TFileIOResult, int)`	writes the third argument to the file as 2 bytes.
`WriteFloat(TFileHandle, TFileIOResult, float)`	writes the third argument to the file as 4 bytes.
`WriteString(TFileHandle, TFileIOResult, string)`	writes the third argument to the file as several bytes.

Table 9.4: A summary of functions for reading and writing data to files.

Datatype	Memory Requirement
`char`, `bool`	1 byte
`int`, `short`	2 bytes
`float`, `long`	4 bytes
`string`	variable size, one byte per character plus one byte for the special (invisible) end-of-string character

Table 9.5: A summary datatypes and their memory requirements.

```
#pragma config(Sensor, S1, Light, sensorLightInactive)

const string FILENAME = "lightreadings.dat";
const int NUMSAMPLES = 97;
const int SAMPLEINTERVAL = 15; // in minutes
task main() {
  TFileIOResult nIOResult;
  TFileHandle LIGHTFILE;
  int FileSize = 2*NUMSAMPLES;
  int i,j;
  OpenWrite(LIGHTFILE, nIOResult,FILENAME,FileSize);
  for(i=1;i<=NUMSAMPLES;i++) {
    WriteShort(LIGHTFILE,nIOResult,SensorValue[Light]);
    for(j=0;j<SAMPLEINTERVAL && i!=NUMSAMPLES;j++) {
      wait10Msec(6000); // one minute
    }
  }
  Close(LIGHTFILE,nIOResult);
}
```

Listing 9.1: This program records light level readings to a file at 15 minute intervals for a full day.

```
#pragma config(Sensor, S1, Light, sensorLightInactive)

const string FILENAME = "lightreadings.dat";
const int NUMSAMPLES = 97;
task main() {
  TFileIOResult nIOResult;
  TFileHandle LIGHTFILE;
  int FileSize = 2*NUMSAMPLES;
  int i,n;
  OpenRead(LIGHTFILE, nIOResult,FILENAME,FileSize);
  for(i=1;i<=NUMSAMPLES;i++) {
    ReadShort(LIGHTFILE,nIOResult,n);
    nxtDrawLine(i,0,i,n);
  }
  Close(LIGHTFILE,nIOResult);
  wait10Msec(1000);
}
```

Listing 9.2: This program displays the light readings found in the file.

9.3 Exercises

1. By referring to the ASCII table, write out the bits that represent the letters of your first name. Make sure you capitalize the first letter and be sure to carefully identify each byte and its corresponding letter.

2. Consider a task in a program that reads the left and right motor encoders every tenth of a second and writes the values to a data file. How large (bytes) will the data file be if the readings are taken over a period of 10 seconds?

3. Suppose your name is stored in two string-type variables, FirstName and LastName. How many bytes will be used if these two variables are written to a file?

4. What will be the result of the following snippet of code?

```
int n = 98;
nxtDisplayString(1,"%c", (char)n);
```

5. Recall the Etch A Sketch™ exercise (Exercise 4 from Section 8.3). Describe a method for storing the finished sketch in a file to be opened later and displayed. How large (bytes) will the sketch file have to be?

6. Write a pair of programs called `Record` and `Playback` for a tri-bot style robot. The `Record` program will record motor encoder values every tenth of a second for 10 seconds while the user pushes the robot around on the floor. The encoder values will be written to a data file. The `Playback` program will open the encoder data file and use the data to reproduce the original movements of the robot.

Chapter 10

Inter-Robot Communication

NXT bricks have the ability to communicate with each other wirelessly via the Bluetooth™ protocol. In this chapter we will learn how to use this capability and, in the process, learn some of the basic features of network communication. Network communication in software is ubiquitous. Mobile phones, iPods, laptops, credit card swipes, and ATM's all have internal software with networking capabilities. There are even refrigerators and vending machines with networking capabilities!

To take advantage of NXT-to-NXT communication, the bricks in question have to be paired up manually. Instructions on how to do so are available in the RobotC On-line Support on the left side-bar under the `ROBOTC Interface` → `Bluetooth and the NXT` → `Connecting Two NXT Bricks` section.

You will notice in this tutorial that the NXT bricks have been given "friendly" names–in this case "`ROBOTC1`" and "`ROBOTC2`". Following this example, it is a good idea to give your robot a friendly name as well. Under the RobotC menu `Robot` → `NXT Brick` → `Link Setup`, choose the "`Rename NXT`" button and rename the brick.

A single NXT brick can be paired with up to 3 other bricks. RobotC provides a very sophisticated set of inter-robot communication tools. However, when paired with more than one other brick, RobotC's inter-robot communication functions are far less efficient. There is also a simplified set of communication tools that are specific to the situation when only two robots are in communication with each other. We will only consider the simplified communication system summarized here[1].

10.1 Simple Communication Functions

All sending of information between bricks is done with the function in Table 10.1.

At the receiving end, there three entities used to manage communication. These entities are summarized in Table 10.2.

As we can see from Tables 10.1 and 10.2, any and all information transmitted between bricks must be represented in triples of type `int`. For example, if we want to send left and right motor encoder values and the light sensor value, in `sendMessageWithParms()` we would make the first argument the left motor encoder value, the second argument the right

[1] http://carrot.whitman.edu/Robots/NXT/Messaging.htm

Function	Description
`sendMessageWithParms(int, int, int)`	transmit 3 ints (2 bytes each) to the paired brick.

Table 10.1: The primary send function.

Entity	Description
`messageParm[]`	a 3-element int array which always contains the next available received message.
`bQueuedMsgAvailable()`	a boolean-valued function that is `true` if a message is waiting, `false` otherwise.
`ClearMessage()`	a `void` function that removes the current message from the message queue. Calling this function will cause the `messageParm[]` array to be updated with the next available received message.

Table 10.2: The primary entities required for managing received messages.

motor encoder value, and the third argument the light sensor value. At the receiving end, we know that the left motor encoder value is in `messageParm[0]`, the right motor encoder value is in `messageParm[1]`, and the light sensor value is in `messageParm[2]`.

If we continue to send updated information about the motors and light sensor in subsequent transmissions, the messages are stored in a queue in the receiving brick. The receive queue can hold at most 10 messages. After that, transmitted messages are lost until the `ClearMessage()` command is used to free up space in the queue.

Another limitation is that all of the information must be sent in the form of a trio of variables of type `int`. Fortunately, most of the data we want to transmit (e.g. encoder and sensor values) are of this type. Non-integer data like characters, strings, booleans and floats must be somehow converted to integers, transmitted, and then converted back. Messages that need more than 6 bytes of storage (3 `int` variables), must be broken down in to several messages by the sender and then reconstructed by the receiver.

10.2 Message Timing

Timing network communication is an important issue in computer science. If information is sent too quickly, it can be lost. Imagine, for example, a brick sending information every tenth of a second to a second brick that is processing received messages every half second. In 1.4 seconds, give or take, the message queue will overflow and messages will be lost. The careful programmer must take this into account and insure that received messages are processed faster than sent messages. Listing 10.1 shows two programs, a send and a receive. The send program transmits the sonar sensor value to the receiving program at a rate of 10 readings per second. The receiving program processes the values at a rate of 20 readings per second.

Notice the use of the three entities summarized in Table 10.2 in the **Receive** program.

Because messages are being processed at a faster rate than they are transmitted, often the message queue will be empty. The **Receive** program uses the `bQueuedMsgAvailable()` function to see if there is a new message. If not, it does nothing. If there is a message, the data is accessed using the `messageParm[]` array and then cleared using the `ClearMessage()` function.

Send

```
#pragma config(Sensor, S1, Sonar, sensorSONAR)
task main() {
  int x1, x2, x3;
  x2=0; // dummy
  x3=0; // dummy
  while(true) {
    x1 = SensorValue[Sonar];
    sendMessageWithParms(x1,x2,x3);
    wait1Msec(100); // wait one-tenth of a second
  }
}
```

Receive

```
task main() {
  while(true) {
    if (bQueuedMsgAvailable() ) {
      eraseDisplay();
      nxtDisplayCenteredTextLine(3,"%d",
        messageParm[0]);
      ClearMessage();
    }
    wait1Msec(50); // wait one-twentieth of a second
  }
}
```

Listing 10.1: This pair of programs demonstrates the periodic transmission of sonar data from one brick to another. The first program transmits data at 10 readings per second. The second program processes data 20 times per second.

Notice also, in the **Send** program, the dummy variables `x2` and `x3`. In this particular situation, only one of the three available integers is needed. We set the other two parameters to zero and ignore them.

10.3 Exercises

1. Describe a method for sending the current touch sensor value to a second brick. The difficulty here is that the touch sensor is a boolean value and transmitted values are integers.

2. Describe a method for sending a single character value to a second brick. The difficulty here is that data is of character type and transmitted values are integers. (Hint: the ASCII table is a good place to start.)

3. Describe a method for sending an array of integers to a second brick. Your method must account for the size of the array and the timing of the transmission.

4. Using a pair of bricks, build a remote light sensor. The remote brick will transmit the light sensor value to the displaying brick every tenth of a second. The displaying brick will show a sliding bar graph (as in Exercise 8.3.1). What is the range of the wireless transmission?

5. Using a pair of bricks, build a remote controlled tri-bot. One brick will have a set of controls that will control the speed and direction of the tri-bot. The second brick, the tri-bot, will receive the commands and execute them.