Robot C Study Guide

Table of Contents

Introduction

Robot C is not actually all that different from the C++ that you're already learning. However, it differs in how it is applied. In a standard C++ program we write code that follows a relatively strict procedure to get to an end result. In Robot C we are writing code that **responds to events**. Event-driven programming is a fundamentally different concept from what you've been learning so far, and this is why Robot C might appear to be difficult. This guide will follow a similar pattern to the drill problems, but will also contain additional information that will help you through the difficult parts.

The guide is organized into three sections:

1. **Concepts**: Concepts are fundamental things you need to be able to do before writing more complex Robot C programs. For example, turning the motors on, or initializing the encoders.
2. **Skills**: Skills use multiple concepts to achieve something useful. Once you master a given skill, you will be ready to combine it with other skills to achieve a complex task. For example, a very important skill might be waiting for a button to be pressed and released, then taking action based on which button was pressed.
3. **Applications**: Applications are the final piece of the puzzle. They are the "big questions" that we ask on exams and assignments. They use multiple skills, usually do not follow a linear programming path, and require you to make assumptions. For example, assuming that the robot has a lawnmower attachment but is otherwise in the standard lab configuration, how would you go about mowing a lawn?

If you are not comfortable with Robot C, I suggest you begin with the concepts section. However, feel free to skip ahead as you wish. If you feel that you aren't progressing, go back a level and re-enforce what you need before coming back.

One of the major challenges of programming Robot C is that you may not have access to the robots, so you can't tell if you are right until you either test it out or see a solution. Although this is a big challenge, it mimics what you will be doing on the exam. I suggest that if you are going to study Robot C for the exam, begin by writing out the code by hand, then check it against the solutions. However, you may also choose to run Robot C and see if the code at least compiles before checking the solution.

In order to keep the length of this document manageable, I will be supplying solutions in a separate document.

Concepts

Cheat sheet ideas: what is the standard lab configuration? Which motors and sensors are available? How are they organized on the robot?

NOTE: Unless otherwise specified, the robot is in standard lab configuration.

How do you learn Robot C?

Your first step should be to get a really good handle on how the display works, since it allows you to write information to the screen to better understand what your program is doing. It is great for debugging. Your next step should be to learn how the robot can interact with the world, and to learn each subsystem on its own. Learn to use each of the sensors and the motors, so that your robot can actually do things. Now, if that were all, Robot C would be a lot easier. But there is another aspect to programming a robot that interacts in the real world that makes things a lot more challenging.

The final step is something that we don't usually consider in C++ - time. In our C++ programs we had only a very limited idea of time. Either we waited on the user to do something, or we computed as fast as we possibly could. However, the robot is a physical thing, and it doesn't react "as quickly as possible". If you try to turn, you have to wait until the turn is finished, even if the computer on board is fast enough to run your code in a few microseconds! In fact, your robot will spend most of its time waiting. It might be waiting on you to push a button, or waiting on its sensors to indicate that a certain motion is complete. A Robot C program can be thought of as tiny bursts of furious computation followed by agonizingly long waits. In order to really understand Robot C, you need to really understand the different timescales on which the robot operates - the super fast timescale of computation, and the lazy timescale of physical action.

In this section, your goal will be to learn each subsystem individually. Later, we'll think about how they all fit together.

The display

Review

The display is written to using the function `displayString`. This function takes at least two inputs:

- line number (an integer from 0 to 7)
- The output string itself

In addition to these two things, you may wish to print variables to the screen. To do so, you specify in the output string that these variables will be coming using placeholders that begin with a percent sign %, and that have a **format specifier** following them. Finally, after the output string, you add the variables themselves in the order you want them to appear. For example, to display an integer, x, to the screen, without any additional information:

```
displayString(0,"%d",x);
```

Notice that I still had to include the output string, even though it only contained the placeholder. This is a very important distinction between Robot C and C++! You **always** need an output string. The format specifiers that you have access to are:

- **%d**: an integer (if you're wondering, d stands for "decimal", which is meant to distinguish it from "binary", and **not** to imply that you can store numbers that contain decimal points. It's a confusing and unfortunate term that comes from the ancient times when computers didn't do floating point comutations very well)
- **%f**: floating point. There are additional specifiers you can have for floating point numbers. In particular, you can specify the number of digits of precision you want to display as follows:
  - **%.pf**, where **p** is some whole number. For example, %.2f means "display a floating point number with 2 digits after the decimal point". This number must be a constant, it can't be a variable.
- **%s**: a string. It might sound strange to display a string inside of another string, but if you are building up a string based on user input, for example, it's useful. In this course we rarely use this option.

Practice

1. Write "Hello, world!" to the display on line 3.
2. Write your first and last name to the screen on different lines.
3. If you have written "Hi!" on line 1, what happens if you write "There" to line 1 after the first string has been displayed?
4. Declare an integer and set it to have a value of 15. Display it to the screen.
5. Declare a float and an integer. Set their values to whatever you'd like. Display both the float and the integer on the same line.
6. Declare two integers, A and B, and set their values to 5 and 7, respectively. Using the variables and anything else you feel you need, display the string `"A = 5, B = 7"` to the screen.
7. Declare a float and set it equal to 1.234567. Display it to the screen with only two decimal places.

Cheat sheet ideas: how to display multiple variables, especially floating point variables, to the screen.

Initializing and reading sensors

Review

Working with sensors is done in two parts. First, you must initialize the sensor. Next, you need to actually read from it.

Sensor initialization is done to tell the robot two things - which sensor is on which point, and which mode is the sensor in. For simple sensors like the touch sensor there is only one mode. However, the

color sensor, and some of the other sensors you may be using for your project, have multiple modes. A very common mistake in the beginning is to accidentally initialize the sensors to the wrong mode. The modes we will be using for most of this course are:

- **Ultrasonic**: `sensorSONAR`
- **Touch**: `sensorTOUCH`
- **Color**: `sensorColorNxtFULL`

In order to initialize, you use the `sensorType` command. For instance, if the color sensor is on port S1, you write

```
sensorType[S1] = sensorColorNxtFULL;
```

You only need to initialize the sensors once in your program, and typically you do this at the very top. There is very little advantage to initializing your sensors anywhere else.

Reading from the sensor is done via the `sensorValue` command. This command is supplied with the sensor you want to read from, and it **always** returns an integer value. For instance, if you want to read the color sensor that we set up previously, you can do this:

```
int colorValue = sensorValue[S1];
```

You cannot read a sensor that you haven't initialized (although give it a shot! You'll see that the code will compile, you'll just get strange readings).

Practice

Sensor initialization:

1. Initialize the ultrasonic sensor, on port S2.
2. Initialize the touch sensor, on port S1.
3. Initialize the color sensor to read the full range of colors, on port S3.

Sensor reading:

Assume that the sensors have been initialized as above.

1. What is the datatype of all data returned by all sensors? (Hint: all sensors commonly used in class return the same type of data)
2. Read the value from the color sensor and print it to the screen.
3. Read the value from the color sensor and print the color it represents to the screen.
4. What does the ultrasonic sensor read when it is not pointing at anything?
5. What is the range of the ultrasonic sensor, in meters? What does the ultrasonic sensor read if

there is an object just inside of this distance?
6. Read the ultrasonic sensor and convert its reading to a distance in meters. Print that distance to the screen.
7. What are the possible values that the touch sensor can read?

Cheat sheet ideas: the return values of all of the sensors, and their setup codes.

Buttons

Review

There are three usable buttons on the robot, and one, the grey button, which stops the program. The buttons are numbered 1, 2, and 3, and are accessed through the variable `nNxtButtonPressed`, which is created for you by Robot C. This variable **always** stores the button number that is **currently** being pressed, or -1 if nothing is being pressed. It does not retain any information about the last button to be pressed if that button has been released. Also, when you press more than one button, the behaviour is completely undefined and depends on how the robot measured its button presses - so just avoid pressing more than one button at a time!

Practice

1. What is the value of `nNxtButtonPressed` when nothing is being pushed?
2. What are the possible values that `nNxtButtonPressed` can be when a button is being pushed?
3. Is 0 a valid button value? Why or why not?
4. Which button gives which value?
5. What is the value of `nNxtButtonPressed` once a button has been released?

Cheat sheet ideas: which button is which, and how they are organized on the robot's chassis.

The motors

Review

Unlike sensors, you do not need to initialize a motor. This is because the NXT's motor ports are **only** for motors. As a result, as long as you have motors attached to a given port, you can set them. Setting motors is done through the `motor` command, and typically we give that command a motor name. For instance, to set motor A to have 50% power, we say:

```
motor[motorA] = 50;
```

Motor power is an integer between -100 and 100. Internally, Robot C prevents you from setting motors beyond 100% power, positive or negative. So even if you do this:

```
motor[motorA] = 100000;
```

The robot will just set its motor power to 100.

Motors can be set via a variable. For example, the following code is valid:

```
int power = 50;
motor[motorB] = power;
```

And motorB will be set to have a power of 50%.

In the standard lab configuration, if you run both motors at the same positive power, the robot moves forward. If you run them at the same negative power, the robot moves backward. If you run them at different powers the robot will turn.

Practice

1. In the standard lab configuration, which motor ports are used, and where are those motors located?
2. Set motor A and motor C to have two different values.
3. Stop the motors.
4. What happens, in the standard lab configuration, if the motors are set to different values and the robot is put on the floor?

Motor encoders

Review

Motor encoders are special sensors that detect rotation of the motors themselves. When the motors are attached the wheels, the encoders can be used to determine distance travelled. On their own, they only sense the total number of degrees that the **output** of the motor has turned since the encoders were last reset. The encoders are accessed through the `nMotorEncoder` variable like this:

```
int MAE = nMotorEncoder[motorA];
```

The variable MAE now contains the current value of motor A's encoder.

Encoders measure rotation in terms of "ticks". In the case of the NXT's motors, there are exactly 360

ticks per revolution of the motor's output shaft, so one tick equals one degree of rotation.

Encoders cannot be reset to an arbitrary value. For instance, the following code does not work:

```
nMotorEncoder[motorA] = 50;
```

What this actually might do is just reset the motor to 0. The only way to change the value of an encoder without moving the motor itself is to reset its value to zero:

```
nMotorEncoder[motorA] = 0;
```

Finally, to determine distance travelled by the robot in standard lab configuration, we need to know the circumference of the wheel and how many revolutions that the wheel has performed. The radius of the wheel is 1.5cm, and therefore the circumference is `2*pi*1.5`. The number of rotations that the wheel has performed is `nMotorEncoder[motorA]/360.0` (or `motorB` or `motorC`). Therefore, for the standard lab configuration, the total distance travelled is:

```
float dist = 2*PI*1.5*nMotorEncoder[motorA]/360.0;
```

This distance will be in centimeters.

Practice

1. Reset motor C's encoder to 0
2. Is it possible to set an encoder to hold the value 1850 without moving the motors?
3. If you run the motors backwards, do the encoders count up or down?
4. Assuming that the robot was moving forward and that its encoders were initialized to zero before the move, use motor A's encoder to determine the distance travelled.

Cheat sheet ideas: how to reset and read encoders. How to determine distance travelled with encoders. Which direction the encoders count when moving forward/back.

Time, waiting, and timers

Review

In essence, a Robot C program can be broken into stages: set-up and wait. The programmer first sets up the robot to perform a certain action (or to do nothing), then waits until some event takes place. The robot continues to perform the action, even if it is to do nothing at all, until the event occurs. How to handle complicated events based on the sensors is part of the next section. In this section, we'll talk

only about timing.

The first way we'll talk about to wait is the `wait` functions. These functions completely halt the robot's computer from taking any additional actions until the time is up. By far the most common one is `wait1Msec`, which works like this:

```
    wait1Msec(1000);
```

As the name suggests, this will wait for 1000x1ms, or 1 second. Other options you have are:

- `wait10Msec(x)` - waits for 10*x ms
- `wait100Msec(x)` - waits for 100*x ms
- `wait1Sec(x)` - waits for 1*x seconds

Usually we use wait1Msec because it tends to give us greater control, and because we can give that function a fairly large value to wait for. However, one of the huge disadvantages of waiting for a specified time, as stated above, is that we can't do anything else while we are waiting. For example, what if we were making a game where the user has 5 seconds to catch the robot and push the orange button? In this case, we can't use wait, since we have to have the robot do something during the wait (run away from the user, say). In order to handle this, we use the timers.

There are 4 timers available on the robot, named `T1, T2, T3, and T4`. They fundamentally give us time in 1mS intervals. They are accessed through an array, and they return to us an integer (actually, a long integer, but for almost all purposes this doesn't matter). They start counting as soon as the program starts, and they can **only** be reset to 0, just like with encoders. In order to reset them, we can use the `clearTimer` function:

```
    ClearTimer(T1); //resets timer T1 to 0
```

In order to read the timers, you first have to decide the unit of time you're interested in. Your options are milliseconds, 10 millisecond increments, and 100 millisecond increments. For the next few examples, we'll be accessing timer 2, but the procedure is the same for all of the timers.

```
    int timeInMs = time1[T2];
    int timeIn10Ms = time10[T2];
    int timeIn100Ms = time100[T2];
```

One very important thing to note is that `time1[T2]`, `time10[T2]`, and `time100[T2]` all come from the **same timer**, T2. It's the same time, just different units.

Practice

1. How many separate timers are there on the robot?
2. Do the variables time1[0] and time100[0] represent separate timers?
3. If time1[0] is reading 3400, how much time has passed since it was initialized? If time100[0] is reading 3400, how much time has passed since it was initialized?
4. How do you reset a timer?
5. Can you set the timer to an arbitrary value, like 1500?

Cheat sheet ideas: how to initialize a timer, how to choose a timer to use, how many different timer ticks can be measured.

Skills

Waiting and the concept of an event

An **event** is key to event-driven programming. Typically events are defined using the sensors. For example, we might wait until the color sensor is reading red, or until the touch sensor has been pressed. In order to define an event for your program, go through the following steps:

1. Determine the conditions that will trigger the event
2. Determine if there is anything you need to do while waiting for the event
3. Determine what setup needs to be done before the event, and what actions are taken after the event

The actual code for writing an event tends to follow a basic pattern. In pseudocode:

```
//setup, especially your sensors if needed
while(sensor value is NOT equal to the condition for the event)
{
  //do what you need to do while waiting
}
//take the post-event actions
```

As a concrete example, let's apply the above steps to the following problem:

- Drive the robot forward until the color sensor senses RED (assume that the color sensor is on port S1).
- The condition that will trigger the event is `sensorValue[S1]` returns the value `colorRed` (numerical value of 5).
- We don't need to do anything while waiting for the event.
- We need to start the motors to move forward before we start waiting.

- Although it's not obvious, we need to **stop** the motors after the event takes place (we said that we will drive forward **until** the color sensor reads red, which implies we need to stop)

The code to do this is:

```
sensorType[S1] = sensorColorFULL;
motor[motorA] = 50; //we can choose an abitrary power
motor[motorC] = 50; //the setup
while(sensorValue[S1] != 5){} //remember, the condition is that the event
    //has NOT happened yet
motor[motorA] = 0;
motor[motorC] = 0; //stop after the event
```

Starting a program with a button press and release, then responding to the button

A very common problem involves waiting on a button press and release, then taking action depending on which button was pressed. In this problem, you are tasked with waiting on a button press and release, then printing the button number to the screen.

1. How many events are you waiting on?
2. What are they?
3. Go through the design process above for each of the events **before writing code**
4. Write the code to solve the problem.

Waiting for sensor events

For these questions, follow the design process for events before you code.

1. Wait for the touch sensor to be pressed and released, then move forward.
2. Drive forward at 100% power until the ultrasonic sensor reads less than 100. Then slow down to 20% power until the sensor reads less than 20. Stop after that.

Waiting for simultaneous and multiple events

Note: This is a **big** topic, and requires careful study.

One of the bigger challenges in Robot C programming is the ability to react to events that might occur at the same time. This is especially challenging if the events do radically different things. For example:

Drive forward at 50% power until the ultrasonic sensor reads less than 100, then slow down to 20% power until the ultrasonic sensor reads less than 20, then stop. If, during the 20% power phase, the color sensor reads red, the robot should increase its speed to 30% power and then keep it there until it

stops, even it no longer sees red. Also stop immediately if the touch sensor is pressed.

The challenge is that last part - no matter what the robot is doing, and no matter what the robot is waiting for, it must **also** wait for the touch sensor to be pressed. However, we can't treat that as a separate event to wait for. We have to wait for both events at the same time, and respond appropriately to whichever one actually occurred.

The key here is to expand the design process above slightly. First, rather than defining "the condition" that triggers the event, we identify **all** conditions that identify the event. This typically means that you will also start to define a lot more events. Let's take the example above and unpack all of the events that might happen:

1. The ultrasonic is reading greater than 100 and the touch sensor is not pressed
2. The ultrasonic is reading greater than 100 and the touch sensor is pressed
3. The ultrasonic is reading less than 100 but greater than 20, the color sensor does not detect red, and the touch sensor is not pressed
4. The ultrasonic is reading less than 100 but greater than 20, the color sensor detects red, and the touch sensor is not pressed.
5. The ultrasonic is reading less than 100 but greater than 20, and the touch sensor is pressed
6. The ultrasonic is reading less than 20

Did you notice how event 5 ignores the color sensor, and how event 6 ignores the touch sensor? That's because they were irrelevant, and would lead to considerably more events. It doesn't matter if the color sensor detects red if the touch sensor is pressed, for instance.

The next step is to find a way to efficiently wait for any of the events, and respond appropriately. To do so, I recommend keeping track of which event has triggered at a given time. Let's do this the simple way and keep an integer called `state` that holds the following values:

- 0 = travel at 50% speed
- 1 = travel at 20% speed
- 2 = travel at 30% speed
- 3 = stop

Now, let's see which events change the state, and how they change it.

The robot begins in state 0. Event 1 changes it to state 1. Event 4 changes it to state 2. Events 2, 5, and 6 change it to state 3.

Now that we are speaking about the robot's behaviour in terms of state, it's a lot more clear what we need to do:

1. Set up the sensors and get the robot moving at 50% speed. Set the state to 0.
2. Loop **while state is not 3**.
3. Inside of that loop, first check the sensors to see what the state should be.

4. Once the state is known, change the speed appropriately.
5. On exiting the loop, stop.

In code, this is:

```
sensorType[S1] = sensorSONAR;
sensorType[S2] = sensorTOUCH;
sensorType[S3] = sensorColorFULL;
int state = 0;
motor[motorA] = 50;
motor[motorC] = 50;
while(state != 3)
{
  if(sensorValue[S1] < 100 && state != 2)
  {
    state = 1;
    motor[motorA] = 20;
    motor[motorC] = 20;
  }
  if(sensorValue[S3] == 5 && state == 1) //red
  {
    state = 2;
    motor[motorA] = 30;
    motor[motorC] = 30;
  }
  if(sensorValue[S1] <= 20 || sensorValue[S2] == 1)
  {
    state = 3;
    motor[motorA] = 0;
    motor[motorC] = 0;
  }
}
```

There are a few things to notice here. First, I had to make sure that we weren't in state 2 when transitioning into state 1 - during the while loop, I might see red for a short time and then never see it again. The behaviour, as written, requires me to increase motor power once I see red I increase the power and never decrease it again. Second, I had to make sure that I was already in state 1 before increasing that power - I might have seen red while travelling far from the wall, at which point the appropriate action would be to remain in state 0. Third, notice how I changed the motor power inside the if statements. This is purely for a slight efficiency increase - I should only change the motor power when the state changes.

Let's try it out.

1. Write a program in which the robot drives forward for a minimum of 5m. After 5m, if the robot detects the color red, stop for 1 second. If it detects the color blue, turn clockwise for 3 seconds. If the touch sensor is pressed at any time, even during the first 5m, it should stop.

2. The robot begins stationary. If the ultrasonic reads less than 100, it should back away as fast as it can for 3 seconds no matter what it is currently doing. If at any time the touch sensor is pressed, it should spin in place for 2 seconds. If at any time the orange button is pressed, it should move forward as fast as it can. The behaviour of the robot is up to you if somehow the touch sensor and the orange button are pressed at the same time. At the end of any of these actions, it should stop. The robot should continue doing this for at most 3 minutes.

Keeping track of things using sensors in a long program

1. Write a program in which the robot drives forward for 5 seconds, turns clockwise for 1 second using tank steering, and drives backward for 5 seconds. Determine the total distance travelled forward and backward while ignoring the turns, and print that distance to the screen.
2. A robot is set up to count the number of people passing through a doorway using the ultrasonic sensor. When a person passes through the door, the ultrasonic sensor reads less than 100. Write a program that enables the robot to count the number of people going through the doorway. Output the number to screen as it changes. Allow the program to run for 10 minutes.

Design pattern: responding to events in a loop

1. The robot will begin on white paper. Drive at 100% power. If the robot encounters red, drive at 50% power. If the robot encounters blue, drive at 20% power. For any other colors, keep going at the current power setting.

2. The robot is driving on a large square of white paper, 10m x 10m, bordered by a thick black border (30cm) on all sides. On the square, distributed randomly, are small, 10cm x 10cm squares of different colors. Complete the following actions depending on what the color sensor reads:

3. When black is encountered, back up for 500ms, then turn 90 degrees clockwise. You may assume it takes 750ms to turn 90 degrees at 50% power.

4. When red is encountered, increase the motor power by 5% up to a maximum of 100%.
5. When blue is encountered, decrease the motor power by 5% down to a minimum of 30%
6. If yellow is encountered, stop the program.

Applications

Mowing a rectangular lawn

Hey, this looks familiar... Assume that the robot has a lawn mower attachment that is 30 centimeters wide, and that the robot is pulling the attachment but is otherwise in the standard lab configuration. Assuming that the lawn you need to mow is rectangular and surrounded by a black path, mow the lawn.

Detecting holes in a fence

The robot is in standard lab configuration with one modification - the ultrasonic sensor has been rotated 90 degrees to the right so it is viewing the distance to the side of the robot. On the right hand side is a fence. This fence has holes in it. At the end of the fence is a wall that the robot will run into. Count how many holes are in the fence, stopping at the wall once the robot hits it. Display the number of holes on the screen.