Pointers and Data Structures in ROBOTC
New capabilities enhance a leading programming environment

Tens of thousands of kids are getting their first programming experience through robotics. From an educational software developer’s perspective it is critical that the software has a low entry point, but a very high ceiling. With the release of ROBOTC 3.5, the ROBOTC development team included many new features designed to teach advanced programming concepts like variable pointers and recursive functions allowing students to learn the higher level programming concepts used by professional programmers today.

Programmers are now able to create efficient and effective code while developing complex algorithms for applications such as autonomous path planning and advanced sensor processing using complex data structures. This article is a tutorial on how pointers work in the ROBOTC programming environment specifically with the LEGO MINDSTORMS NXT robotics controller. The tutorial is written with both new and experienced programmers in mind. New programmers will be introduced to the concepts of variables and pointers using diagrams and examples while experienced programmers will be able to see applications of these advanced concepts being used with a robotics-based application.

The release of ROBOTC 3.5 brought a myriad of new features, including the long-awaited implementation of pointers to variables. Having this functionality opens a whole range of new possibilities, such as the implementation of complex data structures.

WHAT ARE VARIABLES?
Prior to version 3.5, ROBOTC only supported normal variables; in other words, a variable, be it an int, float or anything else, only had a value, like 712, 0.383 or “howdy”. For example:

```c
long foo = 76278;
float baz = 2.9121;
string greet= "hello";
```

In effect, when you declare a variable, the compiler puts aside an appropriately sized amount of memory and gives it a user-defined symbolic name, like “I” or “foo” or “baz”. An integer (int) is 4 bytes large, a float point number (float) is also 4 bytes but a string of character (string - in the case of ROBOTC) is usually 20 bytes large. Take a look at the drawing (right—you see the memory blocks, the labels assigned to them, their contents and the memory address for that block (the hex numbers under the blocks).

DISSECTING A VARIABLE
A variable has two parts, an r-value and an l-value. The term r-value refers to the right side of the assignment statement and l-value refers to the left side. Now consider this snippet of code:

```c
// ptr tutorial example 1

// Task main example

task main()
{
  int i, j;
  i = 10;
  j = 51;
  // This should print out - i: 10, j: 51
  writeDebugStreamLine("i: %d, j: %d", i, j);

  // Assign i’s r-value to j
  j = i;
  // This should print out - i: 10, j: 10
  writeDebugStreamLine("i: %d, j: %d", i, j);

  i++;
  // This should print out - i: 11, j: 10
  writeDebugStreamLine("i: %d, j: %d", i, j);
}
```

When this program is executed, the output to ROBOTC’s “Debug Stream” should look like this:

```
i: 10, j: 51
i: 10, j: 10
i: 11, j: 10
```

(Note: To Open the ROBOTC “Debug Stream”, make sure your ROBOTC interface is in “Expert” or “Super User” mode and open the Debug Stream from the normal debugger windows menu.)

When looking at the memory locations and their contents after each operation, it would look something like this:

<table>
<thead>
<tr>
<th>Initial state</th>
<th>j = i</th>
<th>i++</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>j</td>
<td>51</td>
<td>10</td>
</tr>
<tr>
<td>foo</td>
<td>76278</td>
<td>0x2532</td>
</tr>
<tr>
<td>greet</td>
<td>&quot;hello&quot;</td>
<td>0x3754</td>
</tr>
<tr>
<td>baz</td>
<td>2.9121</td>
<td>0x8354</td>
</tr>
</tbody>
</table>

On the first assignment at the beginning of the program, variable i’s r-value is given the value 10. Right below this assignment, there’s a second assignment where variable j’s r-value is given the value 51. This is considered setting an initial value.

Later in the program, what will happen with the line “j = i”? Simple, the r-value of i is copied and assigned to j’s r-value. Now both j and i’s r-values are 10. What happens when we increment i? Does it change j’s r-value? In short, no, the two r-values are completely sepa-
So now that you know how what pointers are, let’s have some fun with them. Consider the program below:

```c
#include <RobotC v3>

struct{ // struct that we’ll use to store the data looks as follows:
    // foobar: float foobar = 1.0; // we’ll use this data around your program in a simple efficient way.
    // struct pointer
    float *fooPtr;
}

// struct main

// ptr tutorial example 2

// main task

int main()
{
    int i = 10;
    int *iPtr;

    iPtr = &i; // iPtr now points at the address of i

    // This should print out
    // i: 10, iPtr: 656F6968, *iPtr: 10
    // (value for iPtr may vary)
    writeDebugStreamLine("i: %d, iPtr: %p, *iPtr: %d", i, iPtr, *iPtr);

    // ptr tutorial example 2

    i++;

    // This should print out
    // i: 11, iPtr: 656F6968, *iPtr: 11
    // (value for iPtr may vary)
    writeDebugStreamLine("i: %d, iPtr: %p, *iPtr: %d", i, iPtr, *iPtr);
}
```

That should print out something like this in the Debug Stream:

```
arr[0]: 1, arrPtr: 656F6968, *arrPtr: 1
arr[1]: 2, arrPtr: 656F6969, *arrPtr: 2
```

Please note that the address arrPtr may be different in your case. It is important that on the second line, the address arrPtr points to is one higher than the first in the first one. When you look at what it looks like in memory, you can get a good idea of what’s going on:

```
Initial state

i: 10, iPtr: 656F6968, *iPtr: 10
```

```c
arr
1 2 3
```

```c
arrPtr
NULL
```

```c
*arrPtr
undef
```

**INTERESTING FACT ABOUT POINTERS**

When you use sizeof() on a normal variable, like aubyte, int or long, you get the number of bytes that variable takes up. In the case of a byte, that’s one byte, int takes up two and a long uses four. Doing a “sizeof” of a pointer is, well, pointless. You won’t be getting the size of the item you’re pointing at, it’s always 4 in the case of ROBOTC. That’s because the address stored in the pointer is a 4 byte number, a long. Keep in that mind when you want to use sizeof() to get the number of bytes you want to write with a memmove(), for example.

**PUTTING IT INTO PRACTICE**

To get started with a simple example of datalogging, we’ll need the following parts:
- An NXT running ROBOTC 3.5 firmware or higher
- A LEGO Colour Sensor in Sensor Port 1 (S1)
- A LEGO Touch Sensor in Sensor Port 2 (S2)
- A standard LEGO NXT Motor in Motor Port A
- 3 NXT cables to connect the sensors and motor

**PROJECT: DATALOGGING**

If you’ve ever built a robot, may have found that it doesn’t always do what you think it ought to do. Reality has a nasty habit of messing with your perfectly programmed robot. To deal with this, we have two choices either change reality (not an easy task) or find out what’s going on and change the behaviour of our robot. Sounds easy, but how will you know what change if you don’t know what’s going on? This is where datalogging comes in. Datalogging is the practice of constantly taking snapshots of the current state and saving that information in safe place so you can look at it later. This is useful because as your robot disappears under the sofa, it’s a little hard to keep an eye on the screen!

```c
#include <RobotC v3>

// define the data point structures in a struct:
enum{ TColors, TTouch, TMotor, TSpeed, TTime };

struct{ // struct for the data point structure:
    tColors colourNum;
    bool touchSensorPressed;
    long motorEncoder;
    long timeMS;
    jDataEntry dataEntries[MAX_DATAPOINTS];
}
```

```c
// Pointer to jDataEntry struct

// get the data from the sensors and motor

// Wait a little bit

// Our function to read the data from the sensors and put it into a struct

```

As you can see, the individual members of the struct are accessed through the “->” operator. Why not the “.”? Take a look at the function “readData” parameters. We’re not passing “readData” a struct, but rather a pointer to a struct. This allows us to keep a single set of data, but pass it around so other functions can process the actual data rather than dealing with copies of it. When working with members of a pointer to a struct, you’ll have to use the “.” to access or modify the member variables of the struct. To recap:

- Use “->” to access members of a struct via pointers
- Use “.”

As you can see, ROBOTC has made great strides in recent updates to bring advanced functionality to the end user. The inclusion of pointers opens up a whole new level of flexibility and functionality that creative and advanced users can tap into to achieve more efficient code. Pairing pointers and structs together allows the programs to manage data much more efficiently and effectively. In addition to pointer and data structure support, ROBOTC 3.5 also includes new recursive and re-entrant functions. Now that programmers have access to all of these industry standard tools, they will truly be able to test the limits of your robot’s capabilities by implementing complex algorithms and other computer science concepts.

To see the complete program, including a function to print all the data after the initial logging, please go to http://botbench.com/botmag.

For more information, please see our source code on page 100.
So now that you know how what pointers are, let’s have some fun with them. Consider the program below:

```c
// ptr tutorial example 2
task main(){
    int i = 10;
    int *iPtr;  // iPtr now points at the address of i
    iPtr = &i; // This should print out
    // i: 10, iPtr: 656F6968, *iPtr: 10
    // (value for iPtr may vary)
    writeDebugStreamLine("i: ", i, iPtr, *iPtr);

    i++;
    // This should print out
    // i: 11, iPtr: 656F6968, *iPtr: 11
    // (value for iPtr may vary)
    writeDebugStreamLine("i: ", i, iPtr, *iPtr);
}
```

That should print out something like this in the Debug Stream:

```
i: 10, iPtr: 656F6968, *iPtr: 10
```

Please note that the address arrPtr may be different in your case. It is important that on the second line, the address arrPtr points to is one higher than the first in the one.

When you look at what it looks like in memory, you can get a good idea of what’s going on:

```
i: 10, iPtr: 656F6968, *iPtr: 10
```

The output in the ROBOTC Debug Stream should look like this:

```
i: 10, iPtr: 656F6968, *iPtr: 11
```

If you look at it from a memory perspective, our program will look like this:

```
i: 10, iPtr: 656F6968, *iPtr: 11
```