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!

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

What we want to do is the following:

- Log, for a short period of time:
- The LEGO Colour Sensor
- The encoder count of the motor
- Whether or not the LEGO Touch Sensor is pressed
- A timestamp for the above measurements

At the end of the run, it should be possible to review the data. So just how do we intend to use pointers for this? Well, they are great for passing variables around in a program. In this example, we'll use pointers to allow various functions in the program access to the data. Suppose we wish to log sensor and motor encoder data at specific intervals and inspect them later. There are two ways to go about it:

- Pass around all of the data
- Pass around only a data entry struct

If you don't have pointers, the first option may be your only way to go about this. However, ROBOTC is not impeded by this, so we'll use the second way. To achieve this, we'll use a "struct" to hold the data. A "struct" (short for structure) is a way to package multiple variables of different types into a single object. You can then use this object to pass a lot of data around your program in a simple efficient way. The struct that we'll use to store the data looks as follows:

```
struct
{
  TColors colourNum;
  bool touchSensorPressed;
  long motorEncoder;
  long timeStamp;
} tDataEntry;
```

Since this would only hold one data point, we must create an array of them:

```
tDataEntry dataEntries[MAX_DATAPOINTS];
```

If you're familiar with structs, then you'll know that you can access the individual members through the "." operator. For example, to access the motorEncoder member, you would type something like this:

```
tDataEntry entry;
entry.timestamp = 42;
```

To get back to what we were working on, we need a loop to go through the array and read the data:

```
// Pointer to tDataEntry struct
tDataEntry *dataPtr;
for (index = 0; index < MAX_DATAPOINTS; index++)
{
  // Point dataPtr to a fresh new data entry struct
  dataPtr = &dataEntries[index];

  // Get the data from the sensors and motor
  readData(dataPtr);

  // Wait a little bit
  wait1Msec(100);
}
```

Our function to read the data from the sensors and put it into a tDataEntry struct looks as follows:

```
void readData(tDataEntry *data)
{
  data->colourNum = (TColors)Sensor-value[COLOUR];
  data->touchSensorPressed =
(bool)SensorBoolean[TOUCH];
  data->motorEncoder = nMotorEncoder[MOTOR_A];
  data->timeStamp = nPgmTime;
}
```

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 structure, but rather a pointer to a structure. 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 structure, you'll have to use the "->" to access or modify the member variables of the structure. To recap:

- Use "->" to access members of a pointer to a struct: structPtr->member
- Use "." To access member of a regular struct variable: struct.member

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 pointer and structures together allow 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 these 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. ◎

Links
Carnegie Mellon Robotics Academy,
www.education.rec.ri.cmu.edu,
(412) 681-7160

For more information, please see our source guide on page 80.

See More Online!
Scan bar code or type in botmag.com/051304