Hands-on Lab

Lego Data Logging and Actuation

We've explored the Lego NXT's input port and ADC to measure resistances and voltages. This lab will turn the NXT into a data logger. Also, the lab will explore using the output port as a voltage source.

Concept 1 – NXT File Saving Review:

As a quick review, the NXT's ability to save data will be explored.

The program `displaySquareAndSquareRoot1_0.nxc` displayed an integer, its square and square root on the Brick's LCD. This program used the for-loop to iterate the integer from 1 to 10. Building on this example, a program is written to save the values to a file. The file will then be imported into an Excel worksheet. Once one has a worksheet, the data can be manipulated and/or plotted.

Step 1: Click `File – Open` and load `displaySquareAndSquareRoot1_0.nxc.` Click `File – Save As` with the name "`displaySquareAndSquareRoot2_0.nxc`".

Step 2: Define global variables that serve for file handling. Add the following code to above your `task main` routine.

```
// File: displaySquareAndSquareRoot2_0.nxc
// Date: 10/01/12 15:43
// Desc: Display number, its square and square root save to file
// Vers: 2.0
// Refs: displaySquareAndSquareRoot1_0.nxc

// Global variables (for file writing)
unsigned int result; // flag returned when handling files
byte fileHandle; // handle to the data file
short bytesWritten; // number of bytes written to the file
string fileHeader; // column header for data in the file
int fileNumber, filePart; // integers to split up data file names
string fileName; // name of the file
string strFileNumber; // file number e.g myDataFile 1, 2, 3
string strFilePart; // file part e.g. myDataFile1-1, 1-2, 1-3
string text; // string to be written to file i.e. data values

task main ()
```

Step 3: Compose a function to initiate a file. Add the following code above `task main`:

```
string strFilePart; // file part e.g. myDataFile1-1, 1-2, 1-3
string text; // string to be written to file i.e. data values

// Create and initialize a file
void InitWriteToFile() {
  fileNumber = 0;  // set first data file to be zero
  filePart = 0; // set first part of first data file to zero
  fileName = "squareData.csv" ; // name of data file
  result=CreateFile(fileName, 1024, fileHandle);
  // NXT Guide Section 9.100 pg. 1812 and Section 6.59.2.2 pg. 535
  // returns file handle (unsigned int)

  // check if the file already exists
  while (result==LDR_FILEEXISTS) // LDR_FILEEXISTS returns if file pre-exists
     {
      CloseFile(fileHandle);
      fileNumber = fileNumber + 1; // create new file if already exists
      fileName=NumToStr(fileNumber);
      fileName=StrCat("squareData" , fileName, ".csv");
      result=CreateFile(fileName, 1024, fileHandle);
      } // end while

  // play a tone every time a file is created
  PlayTone(TONE_B7, 5);
  fileHeader = "x, x^2, sqrt(x)" ; // header for myData file
  WriteLnString(fileHandle, fileHeader, bytesWritten);
  // NXT Guide Section 6.59.2.43 pg. 554
  // Write string and new line to a file
  // bytesWritten is an unsigned int.  Its value is # of bytes written

} // end InitWriteToFile

task main ()
```

Step 4: Compose a function to write to file. Add the following code above `task main`:

```
} // end InitWriteToFile

void WriteToFile(string strTempText) {
  // strTempText stores the text (i.e. ticks and motorRpm to be written to file
  // write string to file
  result=WriteLnString(fileHandle, strTempText, bytesWritten);
  // if the end of file is reached, close the file and create a new part
  if (result==LDR_EOFEXPECTED) // LDR_EOFEXPECTED is flagged when end-of-file
    { // close the current file
      CloseFile(fileHandle); // NXT Guide Section 6.59.2.1 pg. 535
    // Closes file associated with file handle

    // create the next file name
    filePart = filePart + 1;
    strFileNumber = NumToStr(fileNumber);
    strFilePart = NumToStr(filePart);
    fileName = StrCat("squareData" , strFileNumber,"-", strFilePart ,".csv");

    // delete the file if it exists
    DeleteFile(fileName); // NXT Guide Section 6.59.2.5 pg. 537
    // Delete the file specified by the string input

    // create a new file
    CreateFile(fileName, 1024, fileHandle);
    // play a tone every time a file is created
    PlayTone(TONE_B7, 5);
    WriteLnString(fileHandle, strTempText, bytesWritten);
    } // end if
} // end WriteToFile

task main ()
```

Step 5: Next, compose a function that closes the file. Add the following code above `task main`:

```
} // end WriteToFile

// Close the file
void StopWriteToFile() {
  // close the file
  CloseFile(fileHandle);
} // end StopWriteToFile

task main ()
```

Add this function

At this point, save your NxC program. To recap, Step 2 declared the variables needed for file handling and Steps 3 to 5 created functions to respectively initialize (i.e. create), write string data and close a file.

Step 6: File data is stored as strings. As such, strings must be declared for each integer and float. Also, to create a file, one must initialize one. Add the following within `task main`:

```
task main ()
{
  int x; // integers from 1 to 10
  int xSquared; // square of x
  float xSquareRoot; // square root of x

  string strX;
  string strXSquared;
  string strXSquareRoot;

  // Create a new file that captures time and motor speed
  InitWriteToFile();

  for (x = 1; x <=10; x++) {
    xSquared = x*x;
    xSquareRoot = sqrt(x);
```

Declare string versions of integers and floats. Also, create a file.

Step 7: In the for-loop, the program iterates from 1 to 10, calculating the square and square root. We can use the FormatNum function to create a string version of numbers (i.e. integers and floats). Add the following within the `for-loop`:

```
    TextOut (10, LCD_LINE4, FormatNum("x = %d" , x));
    TextOut (10, LCD_LINE5, FormatNum("xSquared = %d" , xSquared));
    TextOut (10, LCD_LINE6, FormatNum("sqrt(x) = %3.3f" , xSquareRoot));
    Wait (SEC_2);

    // Create string version of calculated values
    strX = FormatNum("%d" , x);
    strXSquared = FormatNum("%d" , xSquared);
    strXSquareRoot = FormatNum("%3.3f" , xSquareRoot);

 } // end of for loop


} // end of main
```

FormatNum is akin to ANSI-C's sprintf() function. It creates strings from numbers.

Step 8: Finally, one should write the 3 strings (`strX, strXSquared` and `strXSquareRoot`) to the file. To do so efficiently, one can employ the ANSI-C `strcat` function which concatenates multiple strings into a single one. Finally, write the string to file. Add the following code within the `for-loop`

```
// Create string version of calculated values
    strX = FormatNum("%d" , x);
    strXSquared = FormatNum("%d" , xSquared);
    strXSquareRoot = FormatNum("%3.3f" , xSquareRoot);

    // Concatenate the 3 strings into a single one.
    // Write resulting string to file. The text will be end with a EOL
    text=StrCat(strX, "," , strXSquared, "," , strXSquareRoot, "," );
    WriteToFile(text);

  } // end of for loop

} // end of main
```

Use strcat to combine strings. Write resulting string to file

Step 9: After the program has generated the data (i.e. completed the `for-loop`), one terminates the program gracefully by closing the file. One can also add an LCD message and beep to let the user know the program is done. Add the following after the `for-loop` and before the end of `main`.

```
  // Concatenate the 3 strings into a single one.
    // Write resulting string to file. The text will be end with a EOL
    text=StrCat(strX, "," , strXSquared, "," , strXSquareRoot, "," );
    WriteToFile(text);
  } // end of for loop

  // Finished computing square and square root, so clean up and quit
  ClearScreen();
  TextOut(0, LCD_LINE2, "Quitting", false);
  StopWriteToFile();
  PlaySound(SOUND_LOW_BEEP);  // Beep to signal quitting
  Wait(SEC_2);

} // end of main
```

Add this alert user of termination and close file

Step 10: Save, compile and execute the resulting program. The program should iterate from 1 to 10, displaying the integers, its square and square root. Additionally, in the background, the Brick stores the data to file named: `squareData.csv`.

To view this data file, after the program completes, select Tools – NXT Explorer (see **Figure 1A**). A pop-up box should display the files stored within your NXT Brick (as shown **in Figure 1B**). Click-and-drag the file `squareData.csv` from the left pane (i.e. Brick's directory) to the right one (your PC's drive).

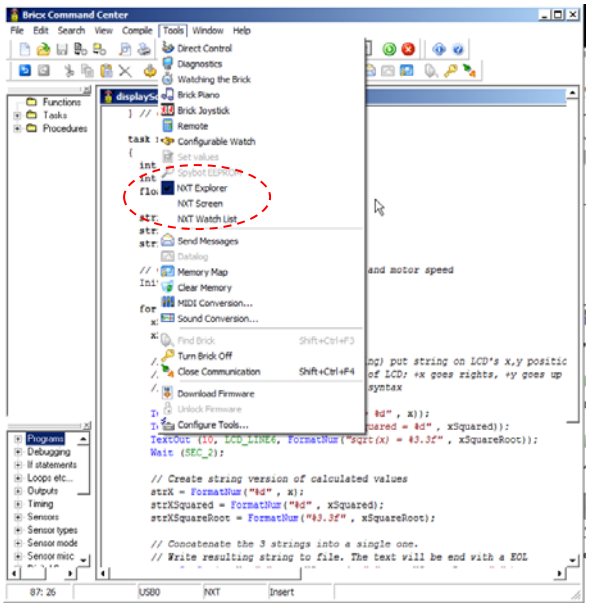


Figure 1A: Launch the NXT Explorer to view Brick's files

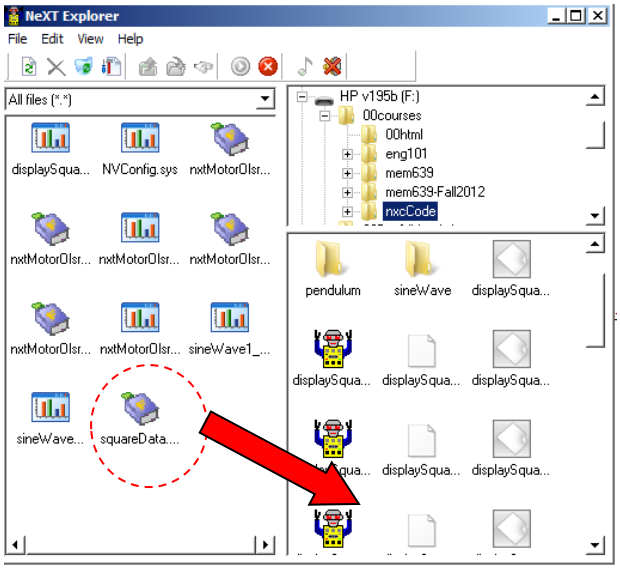


Figure 1B: Click-and-drag the data file `squareData.csv` to your PC.

Step 11: Double-click on the version of `squareData.csv` that is saved on your PC. Excel should already be configured to open CSV (comma-separated files), resulting in **Figure 1C.** **Figure 1D** shows the resulting scatter plot of the first 2 columns.

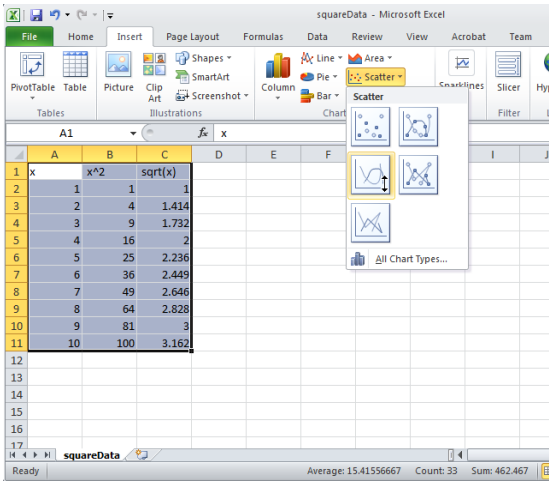


Figure 1C: Excel opens the resulting squareData.csv file. One can then select data for a scatter plot.

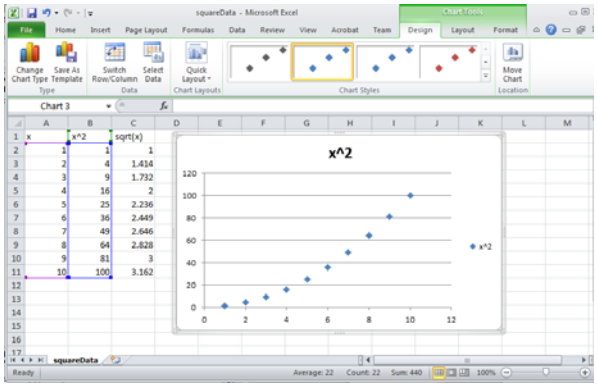


Figure 1D: Scatter plot of first 2 columns of data reveal the expected parabolic curve resulting from computing the square of values.

Code Explanation: `displaySquareAndSquareRoot2_0.nxc` iterates from 1 to 10 using a for-loop. Within this loop, the square and square root is also computed. To save any values to a file, one must first declare (Step 2) and initialize (Step 3) one. File data is stored as strings (i.e. a collection of alphanumeric characters). As such, string versions of any computation are needed and the strcat function is used (Steps 6 and 7) along with the file writing function created in Step 4. After computations are finished (i.e. for-loop terminates), the file should be closed (Step 9) using the function created in Step 5. Steps 10 and 11 show the instructions for using NXT Explorer within the BricxCC IDE to export any files saved on the Brick's memory, to one's PC.

Exercise 1: In NxC create programs for the following:

1-1: Iterate integers from -10 to +10 incrementally by 1. Compute the square and cube and save to a file named "squareAndCube.csv". Export the data file and plot the curves in Excel.

1-2: Capture all file handling functions into a header file named `fileSavingFunctions.h`. Rewrite a new program called `displaySquareAndSquareRoot3_0.nxc` that includes this header file. This new program should run like `displaySquareAndSquareRoot2_0.nxc` – and just serves as a sanity check that file saving works.

1-3: Prove that you know how to name data files and save data in desired formats, and can import these into Excel.

Concept 2 – NXT Timing Review:

As a quick review, the NXT's timer capabilities will be explored. Here, one defines a sampling frequency. A calculation is performed and the result and elapsed time will be recorded to a file.

Step 1: Open BricxCC, click `File – New` and save your program as `timer1_0.nxc`. Write the following code.

```
// FILE: timer1_0.nxc - Works!
// DATE: 08/26/16 12:42
// AUTH: P.Oh
// DESC: Basic timer and file saving demonstration
//       User specifies sampling time and desired maximum elapsed time;
//       calculate and write calculated value to file at desired sampling time

#include "fileSavingFunctionsForTimer1_0.h"

task main() {

  // Boolean related variables - Brick buttons to start/stop execution
  bool orangeButtonPushed, greyButtonPushed;

  // Timing related variables
  long prevTick, curTick, deltaTick; // previous, current and difference in ticks
  string strDeltaTick; // string form of deltaTick for file writing
  float deltaTickInSeconds;  // NB: deltaTick in [msec]
  float elapsedTimeInSeconds; // elapsed time in [sec]
  string strElapsedTimeInSeconds; // string form of elapsed time for file writing
  float maxElapsedTimeInSeconds; // total time in [sec] for data acquisition
  float samplingTimeInSeconds;
  float waitingTimeInSeconds; // for delay loop
  long  waitingTimeInMilliSeconds;
  float epsilonTime = 0.0001; // maxElapsedTimeInSeconds - elapsedTimeInSeconds will be almost zero

  // Calculation related variables
  int x;
  float xSquareRoot;
  string strXSquareRoot; // string form of xSquareRoot for file-writing

  // Create a new file to capture values... call the write-to-file function
  InitWriteToFile();
```

Code description: `task main` begins by declaring button and timer related variables. Also, time related variables are declared. As one will encounter later, the NxC function `CurrentTick()` will be used to poll the Brick's current clock (called a tick counter). Like a stopwatch, variables `curTick` and `prevTick` are used to calculate the time that has elapsed and store the resulting difference in the variable `elapsedTimeInSeconds`. File writing uses alphanumeric values so numeric variables are also declared strings like `strElapsedTimeInSeconds`.

Step 2: Continue adding code to initiate the timer in `timer1_0.nxc`

```
   // Prompt user to begin
   TextOut (0, LCD_LINE1, "Orange Btn starts");
   do {
      orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
   } while(!orangeButtonPushed);

   ClearScreen();
   TextOut(0, LCD_LINE1, "Grey Btn Stops");

   // Initialize timing- and calculation-related variables
   samplingTimeInSeconds = 0.1; // sampling time in [sec]
   maxElapsedTimeInSeconds = 10.0; // total time of data acquisition
   elapsedTimeInSeconds = 0.0; // set elapsed time to zero
   prevTick = CurrentTick();
   x = 0;
```

Code description: This code segment prompts the user to push the NXT's orange button. Timing is defined with sampling time (0.1 sec) with `samplingTimeInSeconds` and the maximum time for the program to run (10 sec) with maxElapsedTimeInSeconds. Before computing (in Step 3), the elapsed time is set to 0.0 sec with `elapsedTimeInSeconds`. The timer is then initiated with a call to `CurrentTick()`.

```
   do {
     greyButtonPushed = ButtonPressed(BTNEXIT, FALSE);
     TextOut(0, LCD_LINE6, FormatNum("Time = %5.3f s" , elapsedTimeInSeconds));

     // perform desired work i.e. square root of values
     x = x + 1;
     xSquareRoot = sqrt(x);
     strXSquareRoot = FormatNum("%5.2f" , xSquareRoot);

     // check how much time to wait for sampling interval to elapse
     curTick = CurrentTick(); // get current tick count
     deltaTick = curTick - prevTick; // measure elapsed ticks [msec]
     deltaTickInSeconds = deltaTick/1000.0; // in [sec]
     waitingTimeInSeconds = samplingTimeInSeconds - deltaTickInSeconds;
     waitingTimeInMilliSeconds = waitingTimeInSeconds * 1000;
     Wait(waitingTimeInMilliSeconds);

     // sampling time interval has past, so capture time and write to file
     curTick = CurrentTick();
     deltaTick = curTick - prevTick; // measure elapsed ticks [msec]
     deltaTickInSeconds = deltaTick/1000.0; // in [sec]
     elapsedTimeInSeconds = elapsedTimeInSeconds + deltaTickInSeconds; // in [sec]
     strElapsedTimeInSeconds = FormatNum("%5.3f" , elapsedTimeInSeconds);
     text=StrCat(strElapsedTimeInSeconds, "," , strXSquareRoot);
     WriteToFile(text);

     // Update current tic value
     prevTick = curTick;

  } while( ((maxElapsedTimeInSeconds - elapsedTimeInSeconds) >= epsilonTime) && !greyButtonPushed);
  PlayTone(400,200); TextOut(0, LCD_LINE4, "Done!");

  StopAllTasks(); StopWriteToFile();

} // end main
```

Code description: The program begins a do-while loop which exits when the user pushed the Brick's grey button or when the total elapsed time has been reached.

A simple square root calculation is performed. The timer is polled again with a curTick = CurrentTick() statement. The difference between ticks is computed between the current and previous ticks using deltaTick= curTick – prevTick statement. This difference is compared to the sampling time to determine how long the Brick should idle. This idle time is given by a Wait(waitingTimeInMilliSeconds) statement.

After the Brick has reached the appropriate sampling time, the elapsed time since the user hit the orange button is computed. This elapsed time and square root computation is then written to file using a WriteToFile(text) statement.

Before looping back, the tick variables are updated using a prevTick = curTick statement.

Once the total elapsed time exceeds the user-defined maximum (i.e. 10 sec), the loop exits, plays a tone, and gracefully exits.

Step 5: Lastly, save your header file `fileSavingFunctions.h` as `fileSavingFunctionsForTimer1_0.h` and make the following modifications.

```
void InitWriteToFile() {
  fileNumber = 0;  // set first data file to be zero
  filePart = 0; // set first part of first data file to zero
  fileName = "timer.csv" ; // name of data file
  result=CreateFile(fileName, 1024, fileHandle);
  // NXT Guide Section 9.100 pg. 1812 and Section 6.59.2.2 pg. 535
  // returns file handle (unsigned int)

// check if the file already exists
  while (result==LDR_FILEEXISTS) // LDR_FILEEXISTS returns if file pre-exists
     {
      CloseFile(fileHandle);
      fileNumber = fileNumber + 1; // create new file if already exists
      fileName=NumToStr(fileNumber);
      fileName=StrCat("timer" , fileName, ".csv");
      result=CreateFile(fileName, 1024, fileHandle);
      } // end while

  // play a tone every time a file is created
  PlayTone(TONE_B7, 5);
  fileHeader = "time [s], Sqrt" ; // header
  WriteLnString(fileHandle, fileHeader, bytesWritten);
  // NXT Guide Section 6.59.2.43 pg. 554
  // Write string and new line to a file
  // bytesWritten is an unsigned int.  Its value is # of bytes written

} // end InitWriteToFile
```

In your `WriteToFile()` function, edit code so that data is saved to the desired file

```
void WriteToFile(string strTempText) {
  // strTempText stores the text (i.e. ticks and motorRpm to be written to file
:
:
:
// create the next file name
    filePart = filePart + 1;
    strFileNumber = NumToStr(fileNumber);
    strFilePart = NumToStr(filePart);
    fileName = StrCat("timer" , strFileNumber,"-", strFilePart ,".csv");
:
   } // end if
} // end WriteToFile
```

Step 6: Save both `timer1_0.nxc` and `fileSavingFunctionsForTimer1_0.h` files. Compile and execute. Once the program runs, hit the orange button to start the program. After 10 seconds, the program should play a tone and exit. Open NXT Explorer to retrieve your data file (called timer.csv) and plot in Excel which should resemble **Figure 2A**.

Figure 2A: Excel plot of `timer.csv`

Exercise 2: In NxC create programs for the following:

2-1: Modify timer1_0.nxc for a maximum run time of 20 seconds and computes the square root at a 0.1 second sampling time. Your saved files may likely look like timer.csv, timer-1.csv, timer-2.csv, etc. Stitch these data files into a single one and plot the curve.

2-2: Prove that you can capture data for desired duration and at a desired sampling time.

Concept 3 – Aliasing

Shannon's sampling theorem puts a limit on the minimum sampling time. Suppose the phenomena you wish to you sample has a frequency f_p. Shannon says that you must sample at least twice as fast. Failing to do so will result in aliasing – or essentially data that does not capture the phenomena.

Step 1: Prepare a 1 Hz sine wave with a function generator. The peak-to-peak voltage should be less than 5 Volts. Also, adjust the sine wave so that voltages are all positive, that is, have a DC offset of 5 Volts. Thus, the minimum voltage would be 0 Volts, and maximum voltage is 5 Volts (see **Figure 3A** top left).

Step 2: Use the oscilloscope to measure the sine wave's frequency. Calculate the minimum sampling time (based on Shannon's Theorem). Connect the function generator to Pins 1 and 2 of the NXT Brick via **Port 1**. Recall, previous lab "Lego Sensing – Analog-to-Digital Basics" Concept 3 – ADC Voltage" and Exercises 3-1 and 3-2.

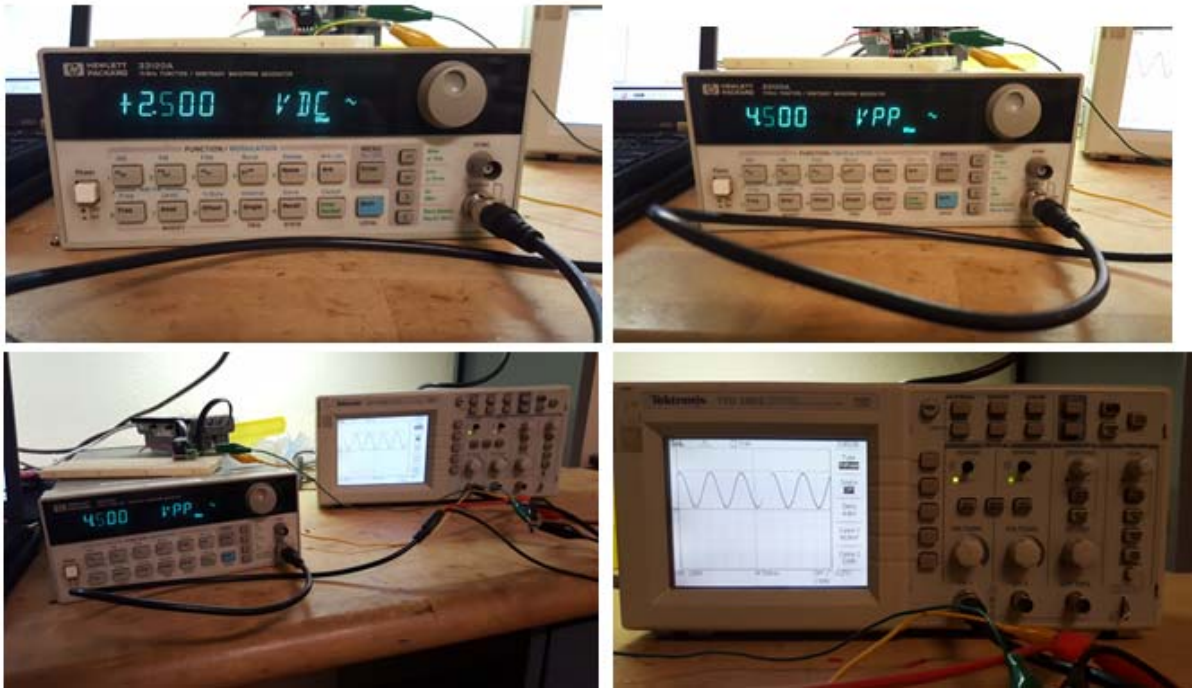


Figure 3A: Function generator set for a 1 Hz sine wave with a 2.5 V DC offset (top left) and 4.5 V peak-to-peak voltage (top right). Oscilloscope hooked up to the function generator (bottom left) and shows resulting sine wave (bottom right)

Step 3: Write an NXC program called **`alias1_0.nxc`** to capture the voltages from the function generator. Use a fast sampling frequency (e.g. 10 times the minimum). Capture data for about 5-seconds. Confirm that a plot of the captured data in Excel, matches that function generator's sine wave.

Step 4: Repeat Step 3 but use a sampling frequency slightly above the minimum (e.g. 1.5 times the minimum). Show the resulting plot

Step 5: Repeat Step 4 using a sampling frequency at the minimum value. Show the resulting plot.

Step 6: Repeat Step 5 using a sampling frequency below the minimum value and show the resulting plot.

Exercise 3:

3-1: Provide Excel plots of the sine wave captured in Steps 3, 4, 5, and 6. See **Figure 3B** for an example. Explain which plots have aliasing and why this was the result.

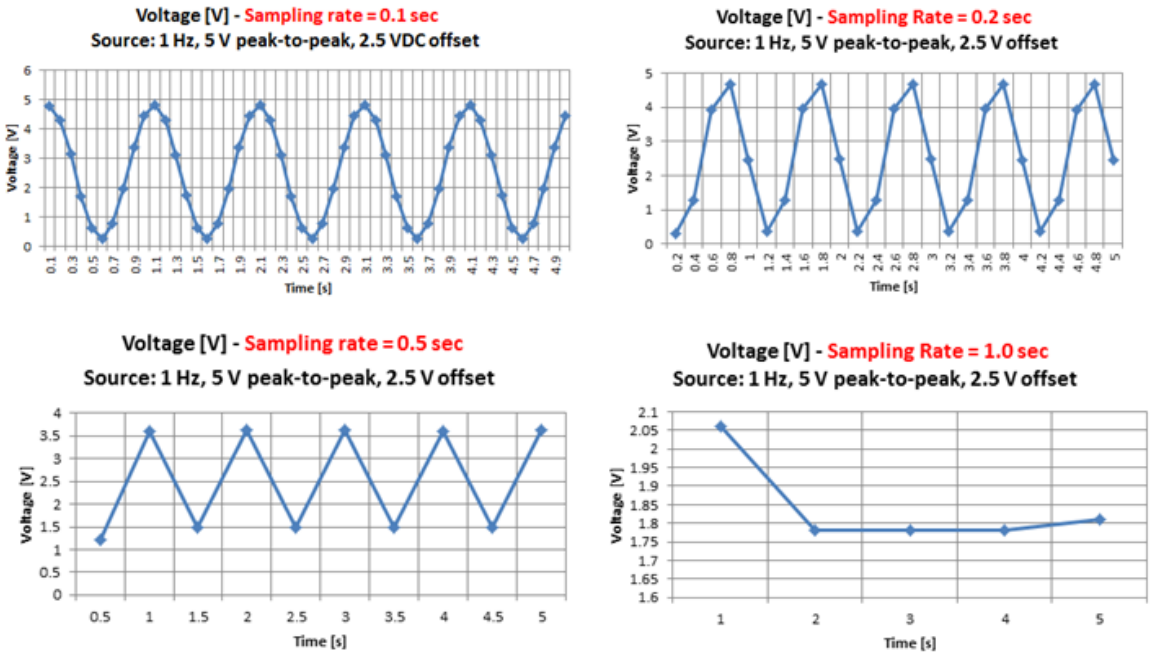
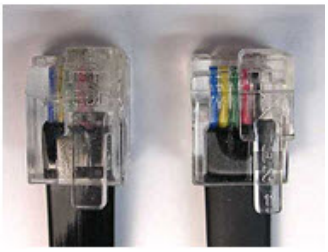


Figure 3B: Excel plots for Exercise 3

Concept 4 – NXT Voltage Source

Motors, relays, and pneumatic valves are examples of actuators. As such, actuators are critical components in any robot. Actuators need a power source to adjust motor speed, relay rates, and valve displacements. The NXT Brick's **Ports A, B, and C** can be programmed as an adjustable power source so that the actuator's state can be controlled. Recall the following pin-out

Pin	Name/Color	Input Role	Output Role
1	ANA (WHITE)	Analog interface	Motor Power 1
2	GND (BLACK)	Ground	Motor Power 2
3	GND (RED)	Ground	Ground
4	PWR (GREEN)	+4.3V supply	+4.3V supply
5	DIGI0 (YELLOW)	I2C clock	Encoder Signal 1
6	DIGI1 (BLUE)	I2c data	Encoder Signal 2



Step 1: Connect an NXT motor to **Port A**. Run the following (sanity check) program

```
// FILE: voltOut1_0.nxc - WORKS!
// DATE: 08/18/16 19:33
// AUTH: P.Oh
// DESC: Port A (Motor port) WHITE (Line 1) is M1.  BLACK (Line 2) is M2
//       Can attach NXT motor (sanity check); piezo buzzer; and E10 7.5V lamp
// VERS: 1.0 - simple program: voltage output (0 to 9V)

task main() {

  // button variables
  bool orangeButtonPushed, rightArrowButtonPushed, leftArrowButtonPushed;
  bool greyButtonPushed;

  int powerLevel; // 0 to 100 will be sent to Port A for corresponding 0V to 9V

  // Prompt user to begin
  TextOut (0, LCD_LINE1, "Orange Btn starts");
  do {
     orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
  } while(!orangeButtonPushed);

  ClearScreen();
  TextOut(0, LCD_LINE1, "Grey Btn Stops");
  TextOut(0, LCD_LINE3, "<- keys ->");
  powerLevel = 50; // middle of range... this is about 3.9V
  // 10 = about 0.6V; 70 = about 5V; 90 = about 7V

  do {
     leftArrowButtonPushed = ButtonPressed(BTNLEFT, FALSE);
     rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
     greyButtonPushed = ButtonPressed(BTNEXIT, FALSE);

     if(leftArrowButtonPushed) powerLevel = powerLevel - 10;
     if(rightArrowButtonPushed) powerLevel = powerLevel + 10;

     if(powerLevel <= 10) powerLevel = 10; // set saturation minimum
     if(powerLevel >= 90) powerLevel = 90; // set saturation maximum

     TextOut(0, LCD_LINE4, FormatNum("Power = %3d", powerLevel));
     OnFwd(OUT_A, powerLevel);
     Wait(250); // need some delay so that buttons can be read properly

   } while(!greyButtonPushed);  // end do-while
```

Code Explanation: The Brick's keys are used to adjust the amount of power (i.e. voltage) being sourced out of Port A. Each time the left arrow button is pushed, the power level decrements by 10 and vice-versa for the right arrow button. This results in controlling the NXT motor's rotational speed.

The power supply comes from the batteries inside the Brick. Thus, the amount of voltage the port can provide is determined by the voltage-levels of the Brick's batteries. The maximum voltage would be 9 Volts (six 1.5 Volt AA batteries). But often, the batteries (especially rechargeable ones) will have lower voltages.

The NXT Brick uses *pulse-width modulation* (PWM) to adjust the voltage coming out of Ports A, B, and C. Brick specs say that the PWM cycle is 128 microseconds (or 7800 Hz). Port A can source 800 mA while Ports B and C can source 500 mA. PWM is the ratio of the times when a signal is on and off. This results in efficiency. Think about a bicyclist. The bike's speed is a ratio of how the bicyclist pedals verses coasts. A bicyclist pedaling 100% of the time will likely get tired quickly. By contrast, pedaling 0% of the time, the bike won't move.

Exercise 4:

4-1: Replace the motor with a voltmeter. Connect the voltmeter's positive cable to Pin 1 and negative cable to Pin 2 of Port A. Rerun your `voltOut1_0.nxc` program. What power level corresponds to +5V? What are the voltages at power levels 10 and 90?

4-2: Replace the voltmeter with a 7.2V lamp and/or a 3-28V piezo buzzer. What happens when you run `voltOut1_0.nxc`?

4-3: Replace the voltmeter with an oscilloscope and run voltOut1_0.nxc. What happens to the wave when you increment/decrement the power level? Sketch the wave forms. What is the frequency of the wave? What are the voltage levels of the waves? What is the ratio of the on to off times when the power level is 10, 50, 90?

Concept 5 – Data Logger – Temperature Sensing

The 3-pin LM35 is a popular temperature sensor. It is calibrated and gives precise Centigrade measurements. It comes in various packages (e.g. TO-92 and TO-220) to meet industrial application needs (see **Fig. 5-1**)

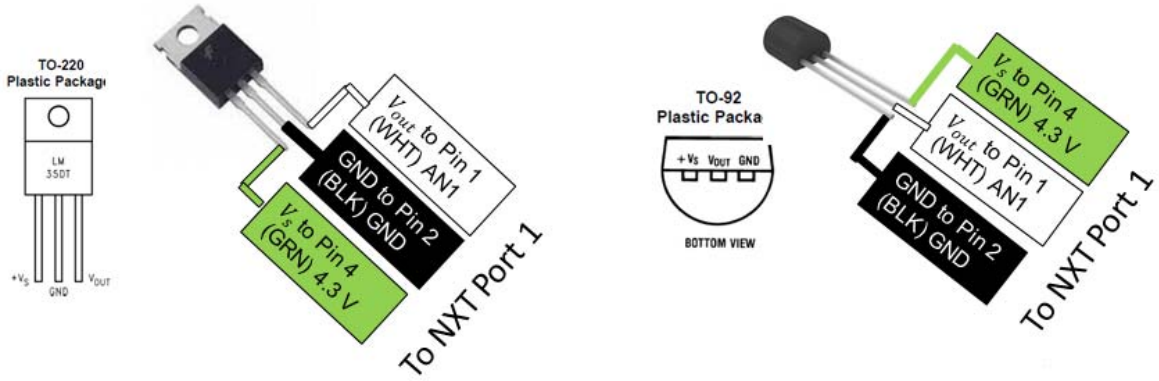


Fig. 5-1: TO-220 package (left) provides heatsinking. The TO-92 package (right) is suitable to create a custom-made sensor probe. NB: The pinouts refer specifically to the LM35 made by National Semiconductor. There are other manufacturers of the LM35 and their version's datasheet should be referenced for pin labels.

The LM35 is linear with a $10.0\,mV/\,^{o}C$. They are rated to operate from -55 to +150 degrees Celsius. They operate with a supply voltage $V_s = 4 \text{ to } 30\,V$. They have a low output impedance.

Naively, one would connect the LM35 directly to the NXT as shown in **Fig. 5-1**. Devices like the LM35, are very low voltage, low current ones. Other sensors that fit this category Hall Effect (for measurement magnetic fields and often used to measure engine RPM) and Infrared Sensors (often used for proximity measurements).

Recall that the inside the NXT Brick is a +5 V supply that's connected to an internal $10k\Omega$ pull-up resistor. This pull-up resistor serves to protect the Brick's internal circuitry. Unknown (and not documented in the Mindstorm's technical documents) is the input impedance of the digital lines that connect to the Brick's microcontroller's ADC. A general rule of thumb is that the impedance is 10 times the pull-up resistor. Thus the Brick's high input impedance could be $100\ k\Omega$.

There are thus two methods to deal with high input impedances. The first is called impedance matching. Here, one identifies a pull-down resistor for the sensor so that enough current can flow into the ADC. The second is to use a voltage-follower which theoretically has zero output impedance and hence all the current from the sensor will flow into the ADC. In reality, the transistor-based voltage-followers have a 0.6 to 0.7 Volt drop across them.

Method 5-1: Impedance Matching

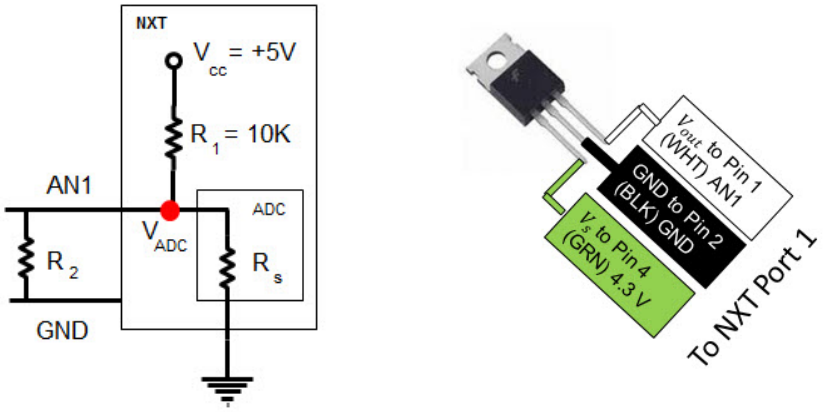


Fig. 5-2: Hook up a LM35 (TO-220 package) to the NXT (left) on lines AN1 and GND. Insert a $R_2 = 220\ \Omega$ pull-down resistor across lines AN1 and GND

Step 1: Attach female jumpers to the pins of the LM35 as shown in **Fig. 5-2** (right). Next, connect a 220 ohm resistor across the NXT Port 1's Lines 1 (AN1) and 4 (+5V).

Step 2: Write a program named `celsius1_0.nxc` that performs ADC and displays the raw sensor value and corresponding voltage. Use the LM35's $10\ mV/\ ^oC$ conversion factor to display the corresponding temperature in Celsius. Also display the temperature in Fahrenheit. Hint (see code snippet below)

```
SetSensorTouch(IN_1);  // homemade touch sensor on Brick Port 1
  do {
  TextOut(0, LCD_LINE1, "Raw value:");
    touchSensorRawValue = SensorRaw(IN_1); // read raw value at port
    TextOut(0, LCD_LINE2, FormatNum("%d", touchSensorRawValue));
    voltageValue = (touchSensorRawValue * 5)/1023; // in [V]
    celsiusValue = voltageValue / volts2Celcius; // in [C]
    fahrenheitValue = (9.0/5.0)*celsiusValue + 32.0; // in [F]
    TextOut(0, LCD_LINE3, "Voltage is:");
    TextOut(0, LCD_LINE4, FormatNum("%3.3f [V]", voltageValue));
    TextOut(0, LCD_LINE5, "Celsius is:");
    TextOut(0, LCD_LINE6, FormatNum("%3.3f [C]", celsiusValue));
    TextOut(0, LCD_LINE7, "Fahrenheit is:");
    TextOut(0, LCD_LINE8, FormatNum("%3.3f [F]", fahrenheitValue));

    Wait(100);
    ClearScreen();
  } while(true); // endless do-while loop
```

Step 3: Prepare a thermal source (e.g. hot pot or ice bath). Tape the LM35 (TO-220) to the thermal source and execute `celsius1_0.nxc`. Use a thermometer to compare results with those reported by the NXT.

Exercise 5:

5-1: Write observed differences between using a thermometer and your NXT LM35 sensor running `celsius1_0.nxc`.

5-2: Save a copy of `celsius1_0.nxc` and name it `celsius1_1.nxc`. This time, include an offset value of 0.11. Write observed differences between using a thermometer and your NXT LM35 sensor running `celsius1_1.nxc`. Hint: code snippet

```
float offsetDueToVcc = 0.11; // in [V].

adjustedVoltageValue = voltageValue- offsetDueToVcc; // in [V]

celsiusValue = adjustedVoltageValue / volts2Celcius; // in [C]

TextOut(0, LCD_LINE3, "Voltage is:");
TextOut(0, LCD_LINE4, FormatNum("%3.3f [V]", adjustedVoltageValue));
```

Method 5-2: Voltage-Follower

Transistors can be configured into a voltage-follower. The 2N3906 PNP transistor is widely available device.

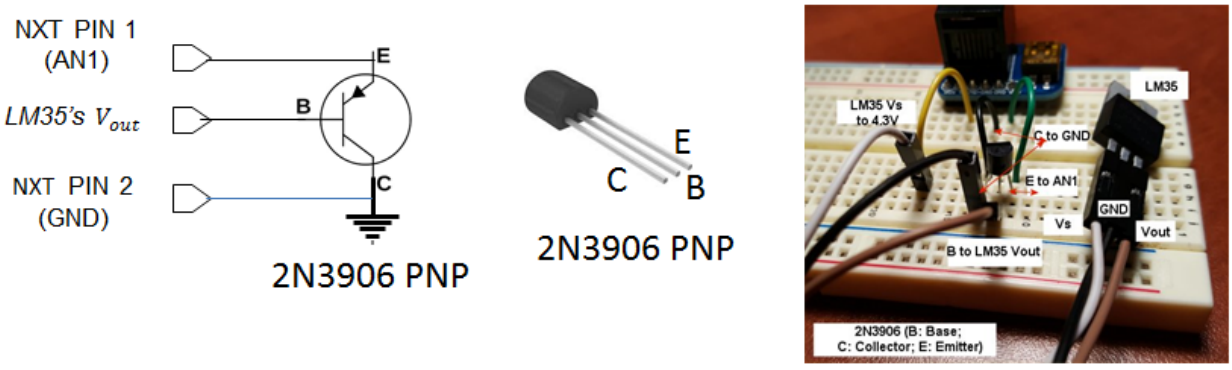


Fig.5-3: Then 2N3906 transistor is also a TO-92 package. This particular version (Radio Shack 276-1604) has the 3-pins labeled as shown (right).

Step 1: As before, attach female jumpers to the pins of the LM35 as shown in **Fig. 5-2** (right). Next, insert the 2N3906 transistor into a solderless breadboard. Connect the opposite end of the (LM35) jumpers into the 2N3906 PNP transistor as shown in Fig. 5-3 above.

Step 2: Write a program called **`celsius1_2.nxc`**. This is essentially the same as `celsius1_1.nxc` as seen in **Exercise 5-2**. In `celsius1_2.nxc` use an offset voltage of 0.6 Volts. Hint: code snippet below

```
float offsetDueToVcc = 0.60; // From 0.6 to 0.7 [V]

  SetSensorTouch(IN_1);  // homemade touch sensor on Brick Port 1
  do {
  TextOut(0, LCD_LINE1, "Raw value:");
    touchSensorRawValue = SensorRaw(IN_1); // read raw value at port
    TextOut(0, LCD_LINE2, FormatNum("%d", touchSensorRawValue));
    voltageValue = (touchSensorRawValue * 5)/1023; // in [V]
    adjustedVoltageValue = voltageValue- offsetDueToVcc; // in [V[
    celsiusValue = adjustedVoltageValue / volts2Celcius; // in [C]
    fahrenheitValue = (9.0/5.0)*celsiusValue + 32.0; // in [F]
    TextOut(0, LCD_LINE3, "Voltage is:");
    TextOut(0, LCD_LINE4, FormatNum("%3.3f [V]", adjustedVoltageValue));
    TextOut(0, LCD_LINE5, "Celsius is:");
    TextOut(0, LCD_LINE6, FormatNum("%3.3f [C]", celsiusValue));
    TextOut(0, LCD_LINE7, "Fahrenheit is:");
    TextOut(0, LCD_LINE8, FormatNum("%3.3f [F]", fahrenheitValue));

    Wait(100);
    ClearScreen();
  } while(true); // endless do-while loop
```

Step 3: Prepare a thermal source (e.g. hot pot or ice bath). Tape the LM35 (TO-220) to the thermal source and execute `celsius1_2.nxc`. Use a thermometer to compare results with those reported by the NXT.

Exercise 5 continued:

5-3: Write observed differences between using a thermometer and your NXT LM35 sensor running `celsius1_2.nxc`.