Hands-on Lab

Lego Communications – I2C Basics

The Lego NXT has 4 sensor ports. Each port is capable of I2C communications. The PCF8574 is an I2C chip that provides 8-bit digital lines. These lines can be configured to serve as outputs or inputs. The net effect is that I2C and the PCF8574 increase the NXT's capabilities to interface and communicate with devices.

Preamble – NXT and the PCF8574

A high or low signal represents a binary system. Electronically, these signal states are represented by voltages. In TTL (transistor-to-transistor logic) chips, +5V is HI and GND is LO. The ability to control HI and LO states is important; when interfaced to devices, these states allow a computer to turn on/off actuators (like lights, relays and transistors) or read the closed/open state of sensors (like switches).

The wires that connect a computer to a device are called digital lines. These lines are often bundled together and called a port. Quite common is an 8-bit port where eight lines form a port. In part, this is historical because 8-bits (called a byte) yield $2^8 - 1 = 255$ unique states. In ASCII, which is the standard to represent alphanumeric characters, 255 states could capture all English letters, digits and symbols.

PCF8574 Motivation: Suppose one wants their NXT Brick to turn on an off-the-shelf relay or read a standard 12-key keypad. Since such devices are rarely I2C-compatiable, one has to resort to using digital lines. The PCF8574 provides such lines. It is I2C-compatiable and yields eight digital lines. Each line can be configured to be either an output or input one. Conceivably, the PCF8574 can control up to eight relays or other on/off actuator (like a DC motor or lamp). It can also read up to eight separate switches. The eight lines can be configured into a single 8-bit port which is useful if one wants to interface the Brick to a separate LCD display.

Concept 1 – NXT and the PCF8574 Digital Outputs

Step 1: Using the following schematic (**Fig. 1A**), construct the PCF8574 circuit on a solderless breadboard. Advice: Insert DIP devices (e.g. PCF8574, Resistor network and Bar Graph) into sockets. Then insert sockets into the breadboard (**Fig. 1B**). Use (rainbow ribbon) jumpers between PCF8574 and the Resistor network. Reduce wiring; take advantage of common lines e.g. +5V bus on the breadboard (**Fig.1B right**).

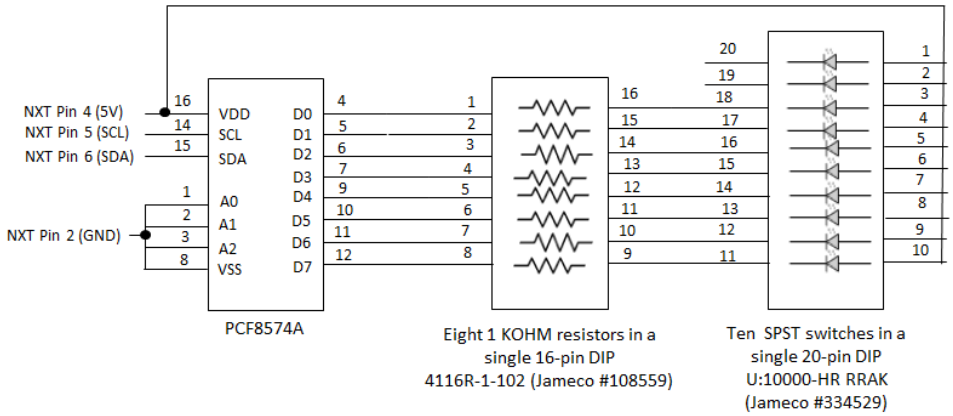

Fig. 1A: NXT-to-PCF8574 circuit diagram. Takes advantage of DIP resistor and LED DIP packages. PCF8574A's digital lines (D0-D7) are configured for output and sink current.

NB: The pin labels and numbers in schematics often do not reflect their physical position in the real DIP device. Real DIP devices enumerate counter-clockwise. Pin 1 on the real DIP device is the first top left pin. Pin 2 is the next pin and so forth. A physical marking (e.g. notch) on the DIP device denotes which side is top.

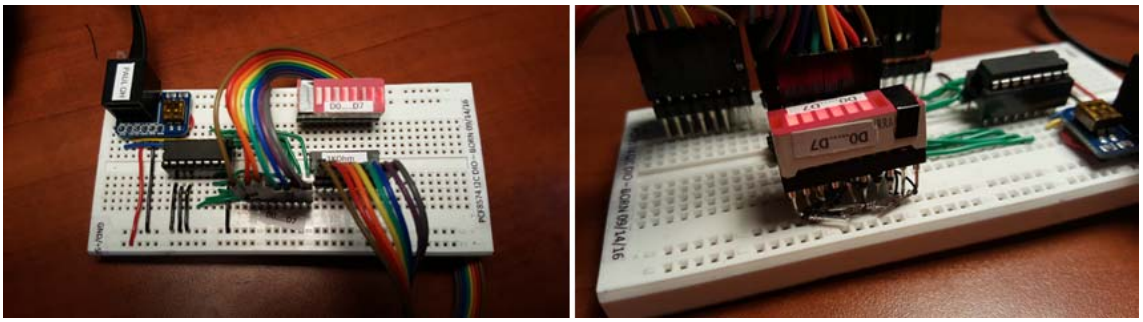

Fig. 1B: An 8-wire ribbon cable connects the PCF8574 chip to a 10-segment DIP display (left). Since only 8 LEDs are needed, black electrical tape covers two of them (right). To reduce wiring, one side of the 20-pin socket is used to connect the +5V bus. Some pins on this side are bend and wirewrapped to the other pins. Demo video: https://youtu.be/AJlvYtODjlg

Step 2: Compose, compile and run the following NXC program called `dioOutput2_0.nxc` (code listing follows on next page).

Code Explanation: The NXC statement `SetSensorLowSpeed(I2Cport)`sets Port S1 for I2C communications with an I2C device. The PCF8574 is the I2C device. The NXT needs to know the device's address. **Fig.1A** ties the PCF8574's address pins (A0, A1, and A2) to GND.

The circuit (and code) uses the *PCF8574A* specifically. The datasheet for this particular version of the PCF8574 states that A2-A1-A0 (i.e. 0-0-0) corresponds to address 0x70 (i.e. 70 hexadecimal). Hence, the statement `WriteBuf[0] = I2CAddr8574.`

The PCF8574A datasheet states that to set the digital lines as outputs, one must write 0x00 to the address. Hence, the statement `WriteBuf[1] = 0x00.`

Next, the statement `I2CBytes(S1, WriteBuf, RdCnt, ReadBuf)`is used to send the array (containing the address and setting) through the NXT Brick.

Before the code enters a `do-while` loop, all LEDs are turned on by setting the eight digital lines (D0-D7) LO. The `do-while` loop then iterates `decimalNumber` from 0 to 255. The statements send the decimal number across the digital lines and lights up the corresponding LEDs. The pattern of LEDs reflects the binary equivalent of the decimal number:

```
WriteBuf[1] = decimalNumber;
WriteBuf[0] = I2CAddr8574;
I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);
```

The `if`-statements just add some fun to the program. Major decimal numbers are just powers of two such as 0, 2, 4, 8…128. In these instances, only one of the LEDs in the bar-graph display will be dark while the remaining seven will be lit. The Brick beeps when this happens.

The NXT is quite fast, hence a `Wait(250)` statement is inserted in the do-while loop. This allows one to actually see the individual LEDs light up or go dark.

```
// FILE: dioOutput2_0.nxc - Works!
// AUTH: P.Oh
// DATE: 07/20/16 10:32
// VERS: 2.0 Uses I2CBytes and my understanding of registers
// DESC: Connect to Port S1.  LEDs configured to sink i.e. when bit is "O" then LED lights up
// NOTE: Uses PCF8574A chip (hence address A2-A1-A0 set to 0-0-0 hence 0x70

#define I2Cport S1 // Port number
#define I2CAddr8574 0x70 // I2C address x040 8574 or 0x70 for 8574A

task main() {

 // array variables (since NXC's I2C functions take array variables)
 byte WriteBuf[2]; // data written to PCF8574A.  Declares a two one-byte variables
 byte ReadBuf[]; // data received from PCF8574A.  We won't be reading any data but we need this for I2CBytes
 int RdCnt = 1; // number of bytes to read

 // button and counting variables
 bool orangeButtonPushed, rightArrowButtonPushed, overflowFlag;
 int decimalNumber; // values from 0 to 255

 SetSensorLowspeed (I2Cport); // PCF8574A connect to NXT on S1
 // Prompt user to begin
 // First, set address with first I2CWrite.  Recall, WriteBuf[1] has address 0xF0 0x00
 WriteBuf[1] = 0x00; // i.e. write zeros to port sets up PCF8574A for writing
 WriteBuf[0] = I2CAddr8574; // i.e. address is 0x70
 I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);

 // Lets start with all LEDs on.  This means making the port LO
 WriteBuf[1] = 0x00; // Port lines are LO; LEDs should be on
 WriteBuf[0] = I2CAddr8574; // i.e. address is 0x70
 I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);

 TextOut (0, LCD_LINE1, "Right Btn starts");
 do {
    rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
 } while(!rightArrowButtonPushed);

 TextOut(0, LCD_LINE1, "Orange BTN quits");
 decimalNumber = 0;
 do {
   orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
   // If pressed, then orange button becomes TRUE.  If not pressed, then orange button is FALSE
   WriteBuf[1] = decimalNumber;
   WriteBuf[0] = I2CAddr8574;
   I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);

   TextOut (0, LCD_LINE3, FormatNum("Value Out: %3d" , decimalNumber));
   // Play beep for major bits being lit up
    if( (decimalNumber == 0) || (decimalNumber == 2) || (decimalNumber == 4) ||
        (decimalNumber == 8) || (decimalNumber == 16) || (decimalNumber == 32) ||
        (decimalNumber == 64) || (decimalNumber == 128) ){
      PlaySound(SOUND_LOW_BEEP);
      Wait(1000);
    }; // end if
    if(decimalNumber == 255) {
      overflowFlag = TRUE;
    } else {
      overflowFlag = FALSE;
      decimalNumber++;
      WriteBuf[1] = decimalNumber;
    }
    Wait(250); // wait 250 millsec
  } while(!orangeButtonPushed && !overflowFlag);

  TextOut(0, LCD_LINE5, "Finished!");
  PlaySound(SOUND_DOUBLE_BEEP);
} // end main
```

Fig. 1A also shows that +5V drives into the eight LEDs, through their corresponding resistors, and into the PCF8574A's eight digital lines (D0-D7). In other words, the PCF8574A is *sinking* current. This means that when a digital line is LO (i.e. 0V), the LED lights up. This is because the current from the NXT Brick's Pin 4 (+5V) can flow through the LED and its resistor. If the digital line is HI (i.e. 5V), then current cannot flow, and the LED remains dark. Sinking current is often preferred (albeit somewhat counter-intuitive). This is because devices can often sink more current than it can source. The PCF8574 datasheet says it can sink about 25 mA, but only source 20 mA.

Exercise 1: In NxC create programs for the following:

1-1 Modify your circuit and modify `dioOutput2_0.nxt` (call the new program `dioOutput2_1.nxc`) so that the PCF8574A sources current. Here, when the digital lines D0-D7 are HI, then the corresponding LED should light.

Concept 2 – NXT and the PCF8574 Digital Inputs

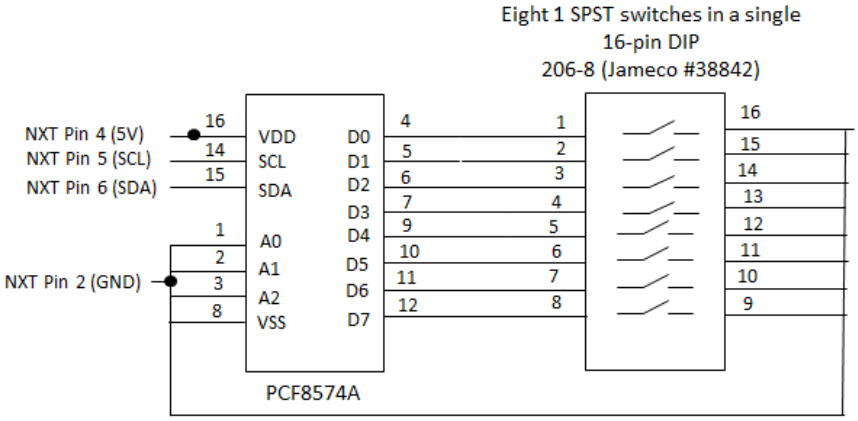

Fig. 2A: NXT-to-PCF8574 circuit diagram. Takes advantage of DIP switch package. PCF8574A's digital lines (D0-D7) are configured for input.

Step 1: Add the DIP switch component to your original circuit (**Fig. 1A**). Use a socket for the DIP switch and take advantage of the GND line that runs along your solderless breadboard (**Fig. 2B**).

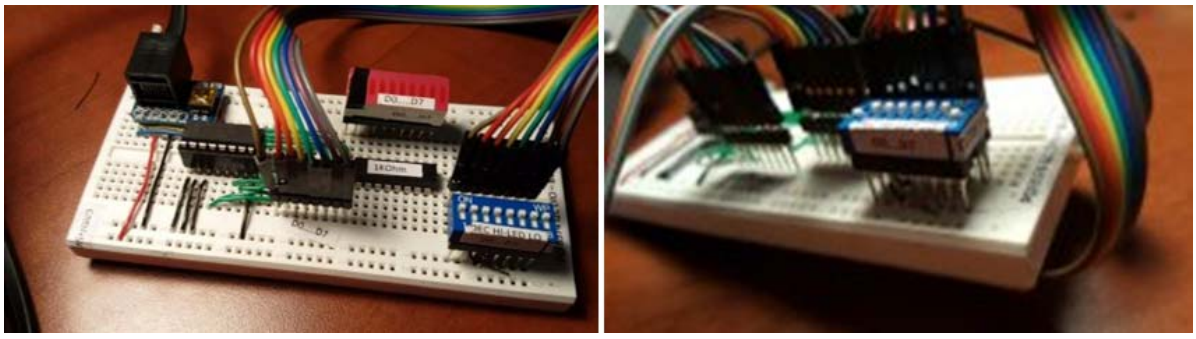

Fig. 2B: 8-wire ribbon cable connects 8-position DIP switch to PCF8574A's digital input lines (left). To reduce wiring, one side of the 16-pin socket is connected to GND (right). Demo Video: https://youtu.be/rHXL8an3Kfs

Step 2: Write and execute an NxC program called `dioInput2_0.nxc`

```
// FILE: dioInput2_0.nxc - Works!
// AUTH: P.Oh
// DATE: 07/20/16 17:32
// VERS: 2.0 - PCF8574A digital input.  Read DIP switch and display decimal value
// DESC: DIP switch configured with pull-ups.  Closed switch (i.e. ON) means 0 Volts going into bit
// NOTE: Uses PCF8574A chip (hence address A2-A1-A0 set to 0-0-0 hence 0x70

#define I2CAddr8574 0x70 // 0x40 8574 or 0x70 for 8574A.  NB: 0x70 = 0111 0000

task main(){

  // PCF8574 read/write variables
  byte WriteBuf[2]; // set up two one-byte variables
  byte ReadBuf[]; // Byte received from PCF8574
  int RdCnt = 1; // number of bytes to read

  // button variables
  bool orangeButtonPushed, rightArrowButtonPushed;
  // Counting variables
  int decimalNumber; // values from 0 to 255

  SetSensorLowspeed(S1); // PCF8574A connect to NXT on S1
  // (1) First, set up PCF8574A for reading.
  WriteBuf[0] = I2CAddr8574; // this is the address 0x70
  WriteBuf[1] = 0xFF; // writing ones to the port sets up chip for reading
  I2CBytes(S1, WriteBuf, RdCnt, ReadBuf); // OK, now port set up for reading

  TextOut (0, LCD_LINE1, "Right Btn starts");
  do {
     rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
  } while(!rightArrowButtonPushed);
  TextOut(0, LCD_LINE1, "Orange BTN quits");

  do {
    orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
    // If pressed, then orange button becomes TRUE.  If not pressed, then orange button is FALSE

    // (2) Read the port
    I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);
    decimalNumber = ReadBuf[0]; // value read from PCF8574
    TextOut (0, LCD_LINE3, FormatNum("Value Read: %3d" , decimalNumber));
    // Play beep for major switch being closed
    if( (decimalNumber == 0) || (decimalNumber == 2) || (decimalNumber == 4) ||
        (decimalNumber == 8) || (decimalNumber == 16) || (decimalNumber == 32) ||
        (decimalNumber ==64) || (decimalNumber == 128) ){
      PlaySound(SOUND_LOW_BEEP);
      Wait(1000);
    }; // end if

    Wait(10);

  } while(!orangeButtonPushed); // end do

  TextOut(0, LCD_LINE5, "Finished!");
  PlaySound(SOUND_DOUBLE_BEEP);
} // end main
```

Code Explanation: Like in Concept 1, the Brick establishes I2C communications using the PCF8574's address. As seen in **Fig. 2A** (as well as **Fig. 1A**), the PCF8574A's address (A2-A1-A0) are tied to GND and hence has a decimal address of 0x70 (70 Hexadecimal). The PCF8574's datasheet says to write 0xFF to the port so that the digital lines (D0-D7) are configured for input. The address is established and lines are configured by this code snippet:

```
SetSensorLowspeed(S1); // PCF8574A connect to NXT on S1
// (1) First, set up PCF8574A for reading.
WriteBuf[0] = I2CAddr8574; // this is the address 0x70
WriteBuf[1] = 0xFF; // writing ones to the port sets up chip for reading
I2CBytes(S1, WriteBuf, RdCnt, ReadBuf); // OK, now port set up for reading
```

The first `do-while` loop just waits for the user to press the Brick's right arrow button. Once pressed the second `do-while` loop reads the digital lines until the user quits the program by pushing the orange button. Inside this loop is the snippet:

```
// (2) Read the port
I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);
decimalNumber = ReadBuf[0]; // value read from PCF8574
TextOut (0, LCD_LINE3, FormatNum("Value Read: %3d" , decimalNumber));
```

Since each digital line is connected to a switch, the above simply reads the state (HI or LO) of the 8 digital lines and displays the resulting decimal number. One uses the DIP switch to configure the state of each digital input. The next effect is binary (from the DIP switch) to decimal (displayed on the Brick) conversion.

Exercise 2:

2-1: When a DIP switch in Fig. 2A is closed, then the corresponding digital line is pulled to GND. What happens to the digital line when the switch is open?

2-2: Modify Fig. 2A with a schematic so that when a DIP switch is closed, the corresponding digital line is pulled to GND, but when the switch is open, is forced to read +5V?

Concept 3 – NXT and the PCF8574 Digital Input and Digital Output

The PCF8574's digital lines are bi-directional; they can be used for both input and output (but not at the same time). However, the lines roles can be switched between the two as needed.

Step 1: Use **Fig. 3A** to connect two 8-wire ribbon cables to DIP switch and DIP LED

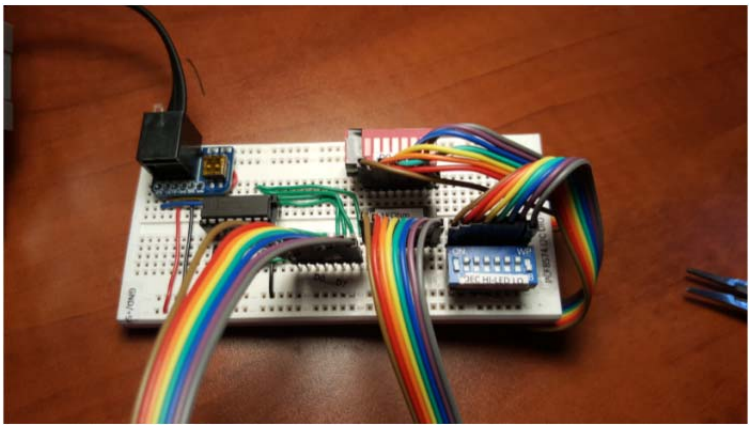

Fig. 3A: Two eight-wire ribbon cables connect the circuits from Concepts 1 and 2. Demo video: https://youtu.be/5GbvA6NnL7M

Step 2: Compose an NXC program called `dioDipLed2_0.nxc` that reads the DIP switch and displays the corresponding LED

```
// FILE: dioDipLed2_0.nxc - Works! Hardware (DIP read, LED glows) and Software (Brick displays number)
// AUTH: P.Oh
// DATE: 07/20/16 17:47
// VERS: 1.1a - PCF8574A read DIP switch and light LED.  Uses bitwise shift
//       1.1b - same but doesn't use bitwise shift; I want to confirm my understanding
//       2.0 - same but refer to dioOutput2_0 and dioInput2_0
// DESC: LEDs configured to sink i.e. when bit is "O" then LED lights up
//       DIP switch configured with pull-up resistors
// NOTE: Uses PCF8574A chip (hence address A2-A1-A0 set to 0-0-0 hence 0x70

#define I2CAddr8574 0x70 // 0x40 8574 or 0x70 for 8574A.  NB: 0x70 = 0111 0000

task main(){

  // array variables (since NXC's I2C functions take array variables
  byte WriteBuf[2]; // Declare two one-byte variables
  byte ReadBuf[]; // Byte received from PCF8574
  int RdCnt = 1; // number of bytes to read

  // button variables
  bool orangeButtonPushed, rightArrowButtonPushed;
  // Counting variables
  int decimalNumber; // values from 0 to 255
  int nbytes;
  byte value = 0;

  SetSensorLowspeed(S1); // PCF8574A connect to NXT on S1
  // (1) Set address of PCF8574A (i.e. 0x70) and memory register
  WriteBuf[0] = I2CAddr8574;
  WriteBuf[1] = 0xFF; // WriteBuf[1] is the register; filling it with ones, sets the port for reading
  I2CBytes(S1, WriteBuf, RdCnt, ReadBuf); // OK, now port is set for reading

  TextOut (0, LCD_LINE1, "Right Btn starts");
  do {
     rightArrowButtonPushed = ButtonPressed(BTNRIGHT, FALSE);
  } while(!rightArrowButtonPushed);
  TextOut(0, LCD_LINE1, "Orange BTN quits");

  do {
    orangeButtonPushed = ButtonPressed(BTNCENTER, FALSE);
    // If pressed, then orange button becomes TRUE.  If not pressed, then orange button is FALSE

    // (2) Set up to read the PCF8574A port.
    WriteBuf[0] = I2CAddr8574;
    WriteBuf[1] = 0xFF; // set up to read port
    I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);
    // (3) ReadBuf[0] should now have data that was read from port
    decimalNumber = ReadBuf[0]; // value read from PCF8574
    TextOut (0, LCD_LINE3, FormatNum("Value Read: %3d" , decimalNumber));
    // (4) Now write to port, thus lighting up corresponding LEDs by writing zeros to port
    WriteBuf[0] = I2CAddr8574;
    WriteBuf[1] = decimalNumber;
    I2CBytes(S1, WriteBuf, RdCnt, ReadBuf);

    // Play beep for major bits being lit up
    if( (decimalNumber == 0) || (decimalNumber == 2) || (decimalNumber == 4) ||
        (decimalNumber == 8) || (decimalNumber == 16) || (decimalNumber == 32) ||
        (decimalNumber == 64) || (decimalNumber == 128) ){
      PlaySound(SOUND_LOW_BEEP);
      Wait(1000);
    };

    Wait(250); // wait 250 millsec
  } while(!orangeButtonPushed);

  TextOut(0, LCD_LINE5, "Finished!");
  PlaySound(SOUND_DOUBLE_BEEP);
} // end main
```

Code Explanation: `dioDipLed2_0.nxc` essentially is a combination of `dioOutput2_0.nxc` (from Concept 1) and `dioInput2_0.nxc` (from Concept 2). The highlighted lines in the code show the sequence. `0xFF` is written to the PCF8574 to set up the digital lines for input. The value read is stored in the byte array `ReadBuf` and stored in `decimalNumber`. After displaying the number on the Brick, `decimalNumber` is stored in byte array `WriteBuf` and outputted to digital lines.

Exercise 3:

3-1: Set the DIP switches to the following positions to complete the table

DIP Position	Brick Value	LEDs ON	LEDs OFF
11110000			
00001111			
01110001			
10000010			

3-2: Capture a video of 3-1 in operation. The Brick should display the value set by the 8-position DIP switch and light up the corresponding LEDs

Concept 4 – NXT and RS-485 Communications

Port S4 of the NXT Brick contains a UART (universal asynchronous receiver/transmitter) which allows for RS-485 communications. This opens up even more possibilities for the NXT Brick. Many devices, like laser range finders and robots contain serial ports. If these ports are USB-based, then USB-to-Serial adapters can be employed. Otherwise, if these ports are traditional serial-based ones (using DB-9 connectors), then cables to connect to the NXT can be made.

Step 1: Compose `485Master1_0.nxc` on the Master NXT

```
// FILE: 485Master1_0.nxc - Works!
// DATE: 09/26/16 12:45
// AUTH: P.Oh
// DESC: Two NXT bricks connected together on their Port S4 (i.e. RS-485 communications)
//       This code runs on Master brick.  485Slave1_0.nxc runs on Slave brick.
//       As long as Slave is on and sending messages, Master iterates and displays number

inline void WaitForMessageToBeSent()
{
  while(RS485SendingData())
  Wait(MS_1);
}

task main() {
  UseRS485(); // (1) Port S4 configured for RS485
  RS485Enable(); // (2) turn on RS485
  RS485Uart(HS_BAUD_DEFAULT, HS_MODE_DEFAULT); // (3) initialize UART to default values
  Wait(MS_1); // (4) wait a bit so all's activated

  int i;
  byte buffer[];
  string msg;
  byte cnt;

  while (true) {
    msg = "Master " + NumToStr(i);
    TextOut(0, LCD_LINE1, msg);
    // send the # of bytes (5 bytes)
    cnt = ArrayLen(msg);
    SendRS485Number(cnt);
    WaitForMessageToBeSent();

    // wait for ACK from recipient
    until(RS485DataAvailable());
    RS485Read(buffer);

    // now send the message
    SendRS485String(msg);
    WaitForMessageToBeSent();

    // wait for ACK from recipient
    until(RS485DataAvailable());
    RS485Read(buffer);

    i++;
  }

  // disable RS485 (not usually needed)
  RS485Disable();
} // end of main
```

Step 2: Edit 485Master1_0.nxc and save as 485Slave1_0.nxc on the Slave NXT. The highlighted statements show the minor edits to make

```
// FILE: 485Slave1_0.nxc - Works!
// DATE: 09/26/16 12:47
// AUTH: P.Oh
// DESC: Two NXT bricks connected together on their Port S4 (i.e. RS-485 communications)
//       This code runs on Slave brick.  485Master1_0.nxc runs on Master brick.
//       When Slave is off, then Master stops.  When Slave is on, the Master iterates

inline void WaitForMessageToBeSent()
{
  while(RS485SendingData())
  Wait(MS_1);
}

task main() {
  UseRS485(); // (1) Port S4 configured for RS485
  RS485Enable(); // (2) turn on RS485
  RS485Uart(HS_BAUD_DEFAULT, HS_MODE_DEFAULT); // (3) initialize UART to default values
  Wait(MS_1); // (4) wait a bit so all's activated

  int i;
  byte buffer[];
  string msg;
  byte cnt;

  while (true) {
    msg = "Slave " + NumToStr(i);
    TextOut(0, LCD_LINE1, msg);
    // send the # of bytes (5 bytes)
    cnt = ArrayLen(msg);
    SendRS485Number(cnt);
    WaitForMessageToBeSent();

    // wait for ACK from recipient
    until(RS485DataAvailable());
    RS485Read(buffer);

    // now send the message
    SendRS485String(msg);
    WaitForMessageToBeSent();

    // wait for ACK from recipient
    until(RS485DataAvailable());
    RS485Read(buffer);

    i++;
  }

  // disable RS485 (not usually needed)
  RS485Disable();
} // end of main
```

Step 3: Connect the Master and Slave Bricks with a standard NXT cable – on Port S4 of each Brick. Run 485Master1_0.nxc and 485Slave1_0.nxc on the Master and Slave respectively. Whenever the Slave stops (e.g. abort) then the Master will stop counting.