Design Competition 2013, Milestone 3 – Sensors and Actuators

Demonstrate to Nick by Wed. 2/6/2013 (see Assignment section below)

Now that you have written a basic program for the NU32 (see Milestone 2), let's look at interfacing to real-world hardware like sensors and actuators.

Debugging is not specifically sensing or actuating, but it is your most powerful tool when writing code. Often you cannot write code that works the first time, nor should you try to. Instead, write only a few lines, test them, and then continue. If you write your whole program at once, and it fails at the first line, you might be stuck re-writing most of your code, so don't write too much without testing.

To debug code on the NU32, send messages containing text back to the NU32_Utility. You can also send text from the NU32_Utility to the NU32. Good debug statements print specific strings such as "got to loop", "exited loop", "got data". You can also send the values in variables or registers: e.g. "i = 10", "LATA = 0xFFFF".

To do this, use the sprintf() function to put the text into a character array, and use NU32_WriteUART1() to send the character array to the computer:

```
#define MAX_MESSAGE_LENGTH 200
…
char message[MAX_MESSAGE_LENGTH];
…
sprintf(message, "Hello! \r\n");
NU32_WriteUART1(message);
```

The '\r\n' is the equivalent of sending a press of the 'enter' key, to advance the cursor to the next line.

To send the value of a variable:

```
int i = 45;
sprint(message, "i = %d \r\n", i);
NU32_WriteUART1(message);
```

Google the sprintf() function to learn how to send other types of variables (like floats).

If you format the character array as numbers with spaces in between, you can use the "plot" ability in the NU32_Utility (the numbers need to be in the range of 0 to 512).

```
int i1 = read_analog(0)/2; // read_analog() returns 0-1023,
int i2 = read_analog(1)/2; // and the plot wants 0-512, so divide by 2
sprint(message, "%d %d\r\n", i1, i2);
NU32_WriteUART1(message);
```

When you aren't hooked up to the computer, printing to an **LCD** is handy. There are many sizes of LCDs - some use several pins, others fewer. Think about how much data you need to display when choosing an LCD.

Image of 16x2 and spi lcds

Sensors

A **pushbutton** is one of the most simple sensors - you basically just need two conductors to make contact. As we saw in the last milestone, you need a pull-up or pull-down resistor to limit the current drawn through the button and provide a consistently defined voltage from the button.

Circuit diagram, pull up or pull down

You can read the signal with a digital input. Set the bit in the "TRIS" register to one to make a pin an input. The value of the bit in "PORT" is the state of the pin.

```
TRISAbits.TRISA1 = 1; // make pin A1 an input pin
if (PORTAbits.RA1 == 1) { // if the pin is at 3.3V
    // do some stuff
}
```

A **potentiometer** is a resistor with a center tap, physically attached to a knob, and electrically attached to a "wiper" pin. Between one end of the resistor and the wiper is a resistance proportional to the angle of the knob. This resistance can be adjusted from zero to the full resistance and sums with the resistance between the wiper and the other end to the full resistance. Usually a pot is used by putting one end at zero volts and the other at supply voltage (e.g. 3.3V), so that the wiper is a voltage proportional to the knob angle. Pots typically only travel zero to 180 degrees. Some change by a log scale (e.g. for audio equipment), but we will use linear ones.

Circuit diagram

You can only read analog signals with analog input pins, which are located on Port B. There are 16 analog inputs, B0-B15. The **analog to digital converter**, or ADC, can burn out if the voltage applied is outside the range of 0-3.3V, so be careful what you connect the input to. The ADC is 10-bit - it stores the voltage as an integer between 0 and 1023; 0 for 0V, and 1023 for 3.3V

```
init_analog();
…
int i = analog_read(2); // read the voltage at pin B2, number could be 0 to 15
                        // i will be in the range of 0 to 1023
```

A **phototransistor** is semiconductor that allows current to flow when it is hit by light in the visible (the lens is clear) or IR (the lens is black) frequencies. Phototransistors are put in series with a resistor and the junction is read by an analog input pin. The bigger the resistance, the bigger the output voltage, but also the greater the sensitivity to ambient light. Many DC robots have been blinded by over-sensitive phototransistor circuits, so take care!

Circuit diagram

Putting tape around the phototransistor lens can help block light from the sides, and designing a cone around the lens, so that only light from the LED you intend to use can hit the lens, is a good idea.

Phototransistors are often used in two ways, as **optoreflectors** or **optointerruptors**.

Optoreflectors contain a light source in parallel with the phototransistor. The light must reflect off of a surface for the sensor to pick it up. The emitting angle and receiving angle are important here; you want narrow angles to pinpoint objects, like in the case of a laser targeting retroreflective tape, and wide angles for color-sensing. Retroreflective tape is a special reflector used in road signs. It contains tiny spheres. Light that enters the spheres bounces around and a large amount of it reflects back at the same angle that it came in, right back at the light source. The Prey robots will be covered in retroreflective tape! This strong reflection is great for a "yes" or "no" signal that tells you if your sensor sees the tape.

An optointerruptor has the light source pointed directly at the sensor. If something gets in the way, the light is blocked and you get a clear indication that an object is there.

Phototransistor circuits create analog voltage outputs. You can read them and interpret the value in code, like you do with any analog signal, or you can build a circuit to turn the signal into a digital one that is easier to interpret in code.

One way to change an analog signal to a digital one with circuitry is to ensure that the signal changes from close to zero to close to 3.3V, and then put the signal on the input of an inverter like the SN… The inverter will square up the signal to make it look digital, and you could read it the same way you read a push button.

Another way is to put the analog signal into a comparator, and build a potentiometer circuit for the other input of the comparator. You can threshold the phototransistor voltage this way - the comparator will tell you when one value is greater than the other, and you can read the digital output of the comparator the same way you read a push button.

Image of circuit

An **encoder** has two optointerruptors arranged around a disc with slits, and the whole assembly is mounted on a rotating shaft. As the disc rotates, you see the following output from the sensors:

Quadrature image

This signal is called quadrature encoding. By looking at how often the signal pulses, you can calculate speed. By counting the pulses, you can calculate displacement using:

Disp = number_of_pulses / pulses_per_revolution * circumference_of_wheel

If the disc is on the motor shaft, your reading will be amplified by the gear ratio of the motor, because the motor shaft spins faster than the output gear shaft, but the signal has some inaccuracy due to backlash of the gearhead. If you get enough slits on the disc, you can put the disc on the wheel and calculate the position of your robot relative to the starting position, but only if the wheel doesn't slip! Navigation by encoders is a type of dead- reckoning. This is more accurate than calculating distance based on time and motor speeds (as in Milestone 2), but the errors integrate with time. Using encoders to calculate turn angle is fine to do, but if you keep adding up the encoder counts with time in order to calculate your position, soon you will have large errors in your estimation of position.

An **ultrasonic range finder** is a prebuilt sensor that can tell you distance from the sensor to an object in front of it. Sometimes you get lucky and a prebuilt sensor works as

advertised - you ask “what is the distance”, and it replies with an accurate measurement. But with most sensors this is not so easy. To use the **HC-SR04 ultrasonic** sensor, raise the “TRIG” pin high very briefly. This will make the “ECHO” pin go high. Note the time this happens. The sensor will emit several 40kHz pulses. These ultrasonic waves bounc around the room, and some echos hit the sensor, which causes the echo pin to go low. When the echo pin goes low, note the timer again. Hopefully, an object directly in front of the sensor bounced a wave directly back to the sensor. This shortest-path time is linearly proportional to the distance, based on the speed of sound in air. But rather than rely on that number, do some calibration. Measure the time for several known distances, and calculate the multiplication factor you should use to obtain distance.

```
init_ultrasonic();
…
double dist = read_ultrasonic(); // returns the distance in meters
```

Note that sometimes the sensor misses the reflection, or picks up other reflections that are further away, giving you a bad reading; thus thorough testing is necessary. The function in the sample code attempts to average out bad readings – but doesn’t do a good job. Can you think of a way to do it better?

There are lots of sensors out there - look for breakout boards on www.sparkfun.com, www.adafruit.com, www.seeedstduio.com, www.parallax.com, and many more.

Actuators

DC motors are (electrically) like a resistor and inductor in series. The angular velocity is proportional to voltage, and the torque is proportional to current. We have a single voltage supply, so it is inefficient to create different voltages for the motor to get different speeds. Instead, we will turn the motor on and off at a high frequency. The on and off voltage values will average out, so the duration spent in the on state determines speed – the longer you stay on, the faster you go. This is called **pulse width modulation** (PWM).

DC motors are very fast with very low torque. We often add a gearhead (or buy a motor with a gearhead already attached) to reduce the speed and increase the torque. A gearhead motor is usually specified by its voltage and the speed at that voltage. Use half the voltage and you get about half the speed. If you run the motor at higher voltages, you risk wearing out the motor earlier than it is rated for - the brushes can wear out, the motor windings can melt or short, and the bearings can seize up.

The gearhead adds some friction, so it may take a few volts to get the motor to start turning. The gearhead friction results in the loss of transmission of torque and speed, and since changing the direction of the motor makes the gears separate, backlash could be as much as a few degrees.

RC servomotors are cheap, provide high torque and low speeds, and are position controlled. You supply power (6V), ground, and a signal proportional to the angle you want

it to go to, and you can assume it will go there, unless it jams on something. Servos have a potentiometer at the last gear stage to measure position. They have a physical stop, so they can only go +/-90 degrees. The servos we have use 6V on the red wire, ground on the brown wire, and signal on the orange wire.

```
init_rcservo();
…
Set_rcservo(45); // set the servo to 45 degrees, or a number between 0 and 180
```

Different servos use different signals to set the angle. The sample code uses a function that should work for most servos, but if you notice your servo "talking" (making a lot of noise or vibration) near 0 degrees or 180 degrees, back off, and only set your servo to angles in a smaller range, like 10-170.

Sometimes you can cut out the physical stop, remove the potentiometer from the loop, reinsert the potentiometer somewhere else in your mechanism, and get the same closed-loop behavior (control the angle of your mechanism). You can also cut out the hard stop, remove the potentiometer from the loop and set it to the middle of the range, changing the position command into a velocity command. Altering a servo will shorten its lifespan - the motors are not designed to continuously run and will over heat.

A **stepper motor** is very different from a DC motor. When you turn on the electromagnets in the motor, the core (and output shaft) will align with the electromagnets. When you cycle through which magnets are on, the motor shaft will snap to specific angles. Stepper motors have low torque and are good for setting a position without feedback from a sensor, but the motor can stutter and cause you to lose track of its position. Stepper motors always use power, even to hold a position.

Brushless motors are similar to steppers, but contain a built-in sensor and a controller. They can run at very high speeds, can provide high torques, and are thus a good choice for drive motors in RC planes, helicopters, or high speed cars.

Solenoids create a push or pull action and require a spring or other mechanism to return it to its default position. They need to be continuously powered, similar to the stepper motor. They are not used often (in DC robots) because you can usually design a mechanism with a different actuator to perform the same function better. We will talk about mechanism design soon.

Assignment:

For Milestone 3, do the following, testing after each step and building up the code as you go:

1. Build a potentiometer circuit, read the value, and print it with the plotting format that NU32_Utility requires. Verify that plotting the graph of voltage works as expected when you turn the pot.

2. Build a phototransistor circuit as an optoreflector. Read the voltage value and add it to the output string you created in the last step, and plot both pot voltage and optoreflector voltage.

3. Add the ultrasonic sensor and read the distance value. Convert the value into the range of 0 to 512, and add that to the plot (remember it has to be an int, not a double, when you print it to the plotting function in NU32_Utility).

4. Add a small servo to your circuit. Make the angle of the servo shaft proportional to the potentiometer angle.

Demo this to Nick, and email him the main C file for reference.