Amicus18 Companion Shield

Amicus18 Companion Shield for Proton

1

Amicus18 Companion Shield for Proton

Amicus18 Companion Shield

A shield is a PCB that fits over the Amicus18 board and provides extra functionality, such as Ethernet, Motor control, LCD, Smartcard, GPS, GSM etc…

All Arduino shields will physically fit on the Amicus18, however, Arduino source code is not compatible with Amicus18, as they differ in two very crucial aspects. First, the Amicus18 uses a Microchip PIC® MCU for it's microcontroller, while the Arduino uses an Atmel AVR microcontroller. The Arduino uses a subset of the language C, where as the Amicus18's supplied language is BASIC. However, there is no reason why any suitable language cannot be used with Amicus18, in fact, it's encouraged.

The entry level shield, and in the authors opinion, the most useful, is the Companion shield. This is a PCB laid out in the pattern of a solderless breadboard. The holes are single sided, which means that components can easily be removed using solder mop braid, or a solder vacuum tool, if a mistake is made, or components need to be re-used.
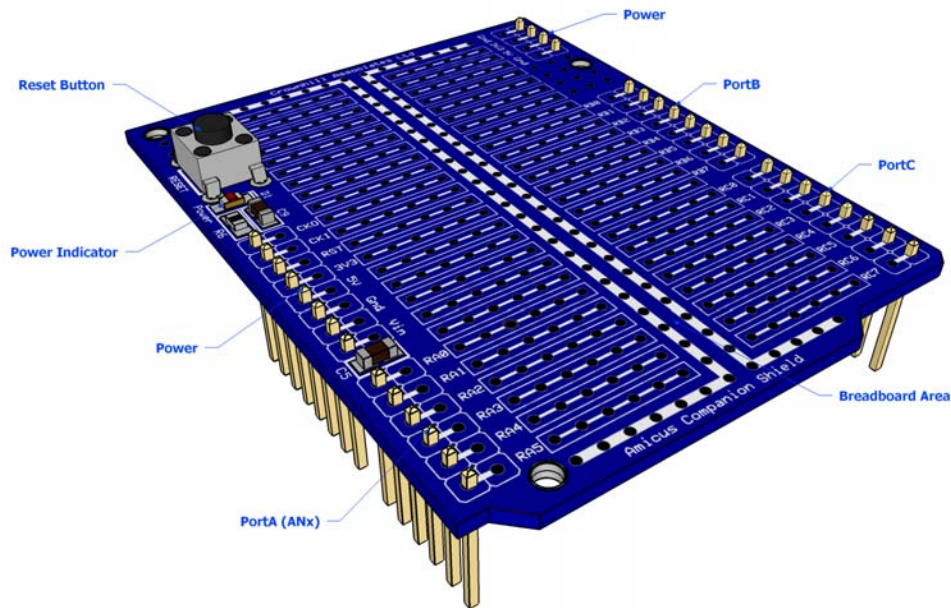
Another solution is to add a solderless breadboard to the Companion shield, thus allowing the full re-use of components without the need for a soldering iron. Notice the use of header sockets instead of header pins.

06-10-2009
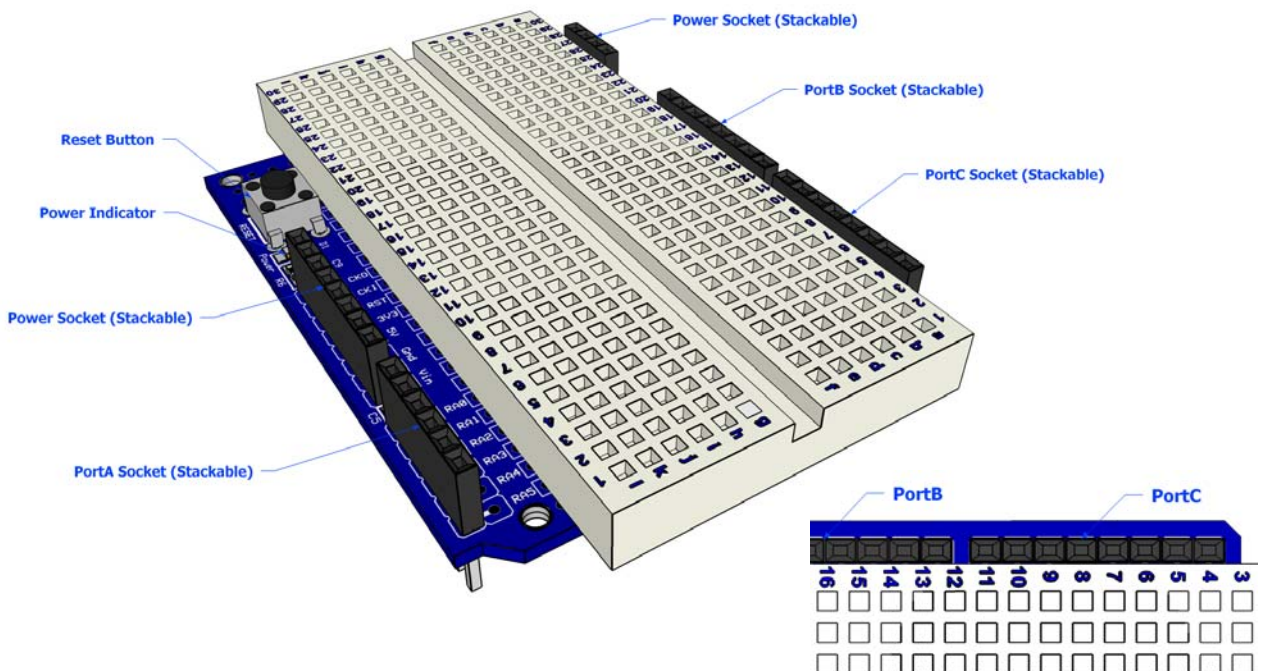
Amicus18 Companion Shield for Proton

Companion Shield Options

The companion shield is available as a blank PCB or ready built. However, there are two flavours of the ready built boards, one with header sockets, and one with header pins. It all depends on what you need to do with the companion shield. The illustrations below show the various flavours:
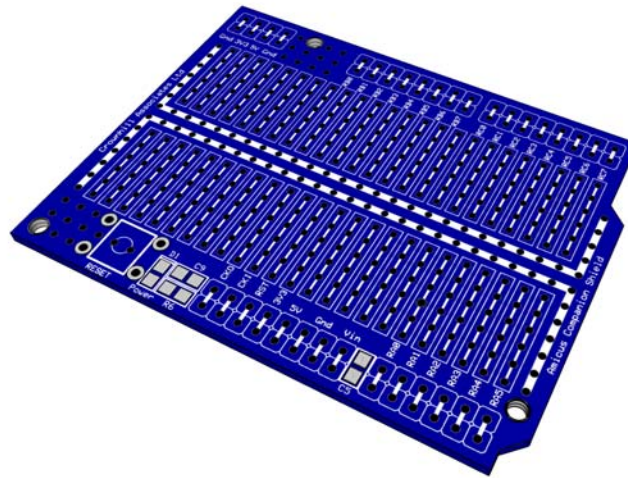
Blank Companion Shield
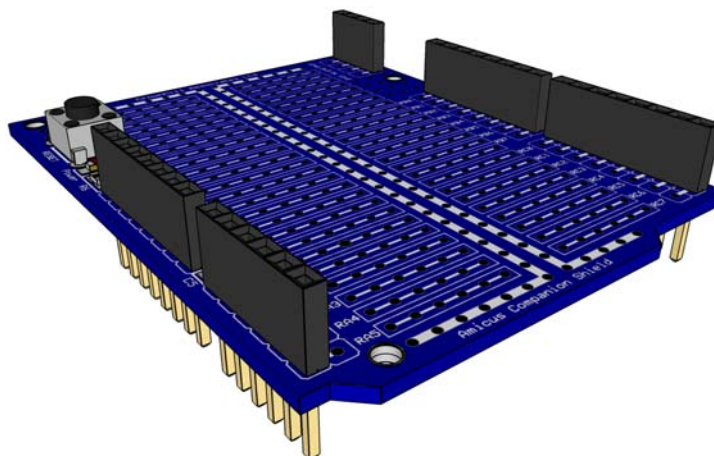
Companion Shield with Header Sockets
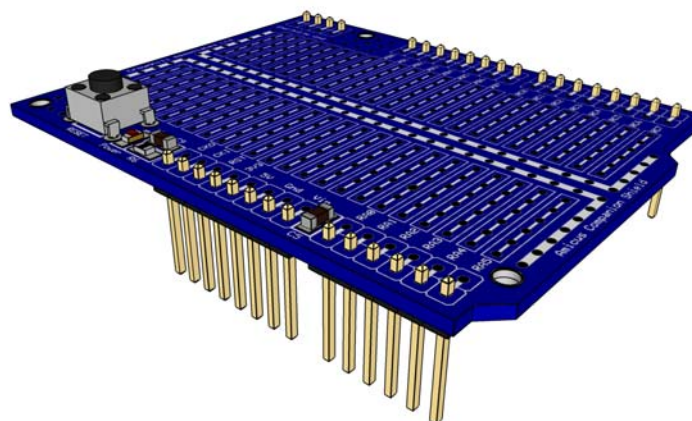
Companion Shield with Header Pins

Amicus18 Companion Shield for Proton

The two flavours of the shield allow the boards to be stackable or at the top of the stack:
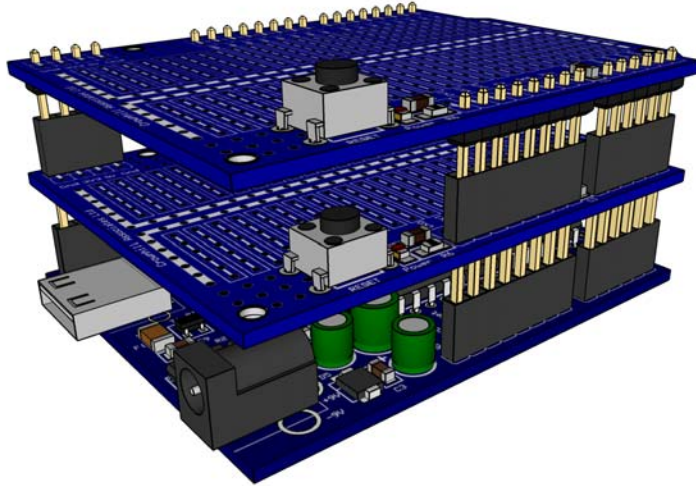
The illustration above shows the Amicus18 board at the bottom of the stack, then a socketed shield, then a pinned shield. A pinned shield could carry an LCD or other user interfacing device that would not suit being stacked between other PCBs. The socket and pin headers used for the companion shield have long legs, thus allowing plenty of clearance between the stacked PCBs, 12mm for the pinned header, and 14mm for the socketed header.

Building the Companion Shield

If you are going to choose the blank companion shield, it must be pointed out that it contains surface mount components (not supplied with it). These components are purely optional, but if you are considering using them, make sure you have the required skills to solder surface mount devices. It's not difficult, and there are plenty of SMT soldering tutorials on the internet.

Start by soldering the decoupling capacitors C5 and C9 on the board, both are 100nF 50 Volt ceramic capacitors with an 0805 casing:
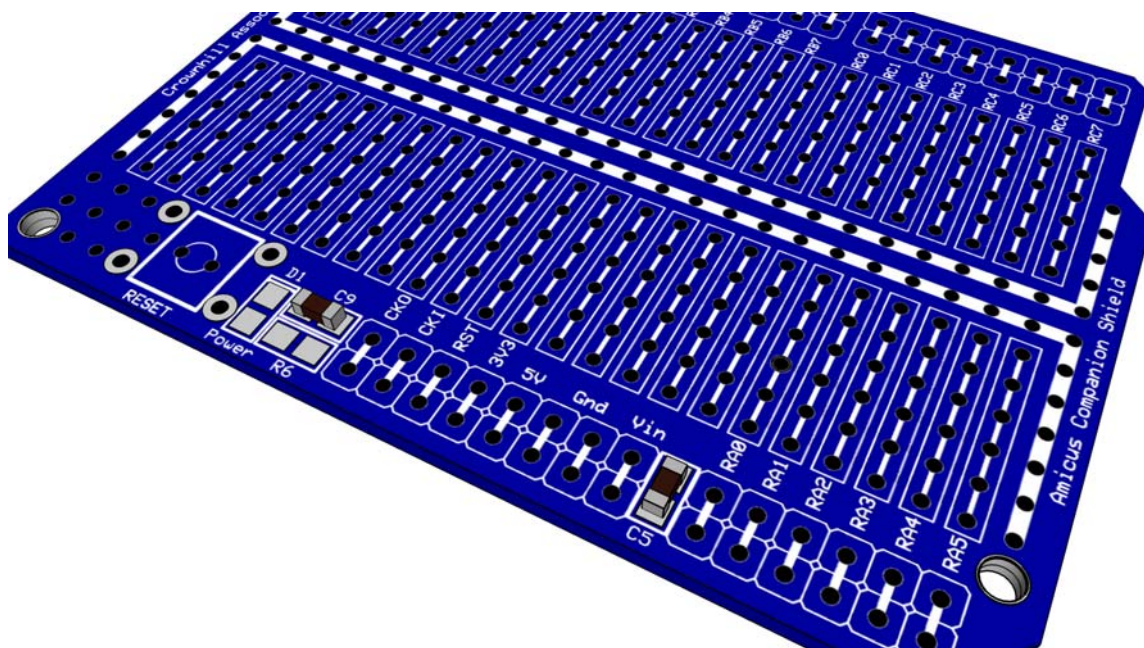
Next solder on the resistor R6 which is a 1KΩ 1% 0805 casing type:
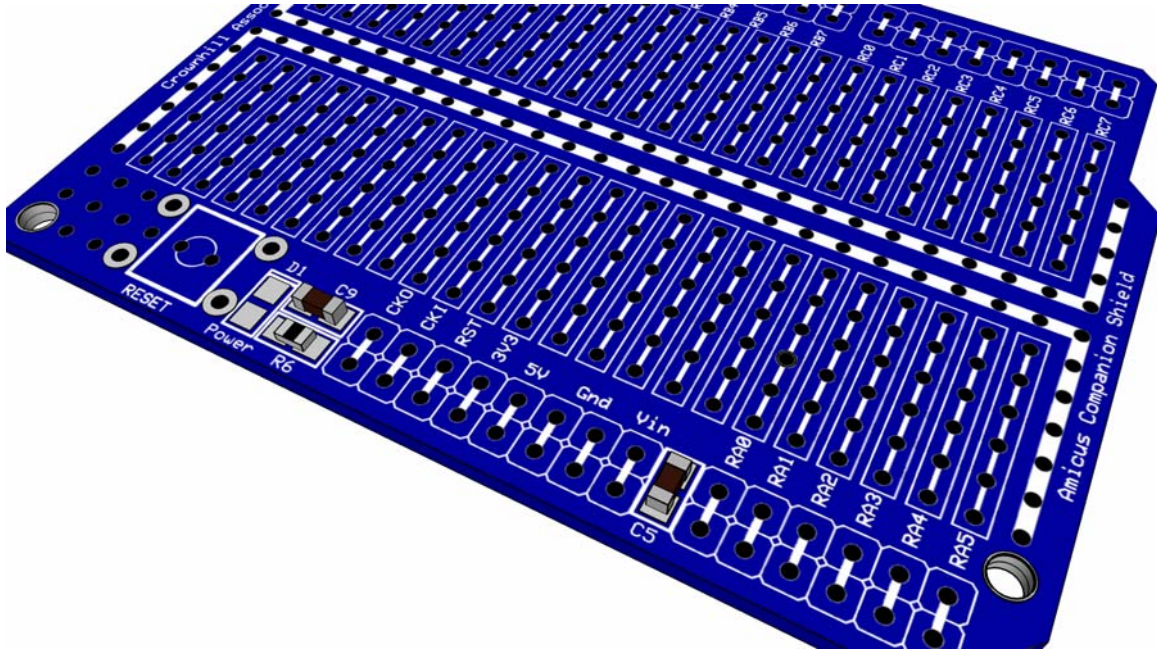
Next to solder is the power indicator LED, this is a red type 0805 casing, but any colour will do. Note that resistor R6 is not required if the LED is omitted:
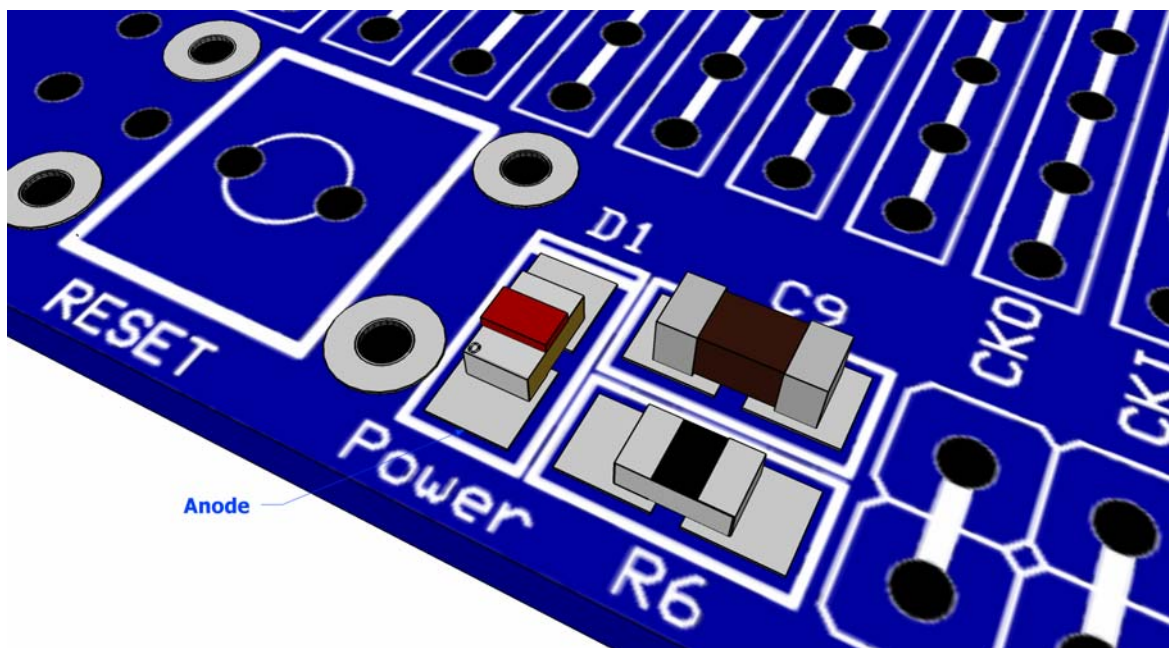
Take note of the orientation of the LED, make sure the Anode is located as in the above diagram. Reversing the LED won't harm it, it just won't illuminate.

5

The next component is the reset button, this is a standard PCB push to make type:
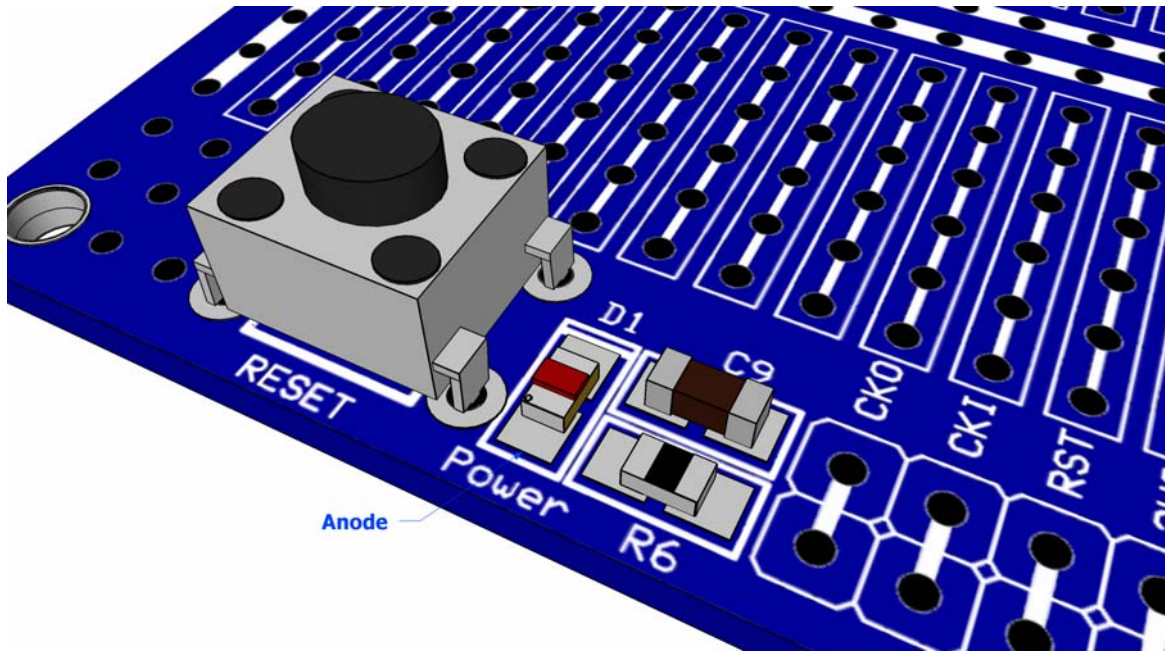
Then place either the header pins or the header sockets as the earlier diagram illustrate. These are standard 2.54 (0.1") spacing Single Inline types (SIL).

You will require 5 of these:

1 x 4 way
1 x 6 way
3 x 8 way

6

First Program

We'll jump straight in at this point and produce our very first program that does something, but not using the companion shield just yet.

Open the ProtonIDE and type in the following code. Note that it is *not* required to type in the commented texts. i.e. *blue texts*:

```
' Flash an LED connected to RB0
' Make sure the Amicus18 board's jumper Q3 is set to the GND position
  Include "Amicus18.inc"  ' Configure the compiler to use the Amicus18 board
  While 1 = 1             ' Create an infinite loop
    High PORTB.0          ' Bring the LED pin high (illuminate the LED)
    DelayMs 500           ' Wait 500ms (half a second)
    Low PORTB.0           ' Pull the LED pin low (Extinguish the LED)
    DelayMs 500           ' Wait 500ms (half a second)
  Wend                    ' Close the loop
```

Move jumper **Q3** to the **Gnd** position, and place an LED into PortB pins RB0 and RB1, with the Cathode connected to RB1, and the Anode connected to RB0. The Cathode is identified by being the shorter of the two wires, and also the body of the LED has a flattened side.
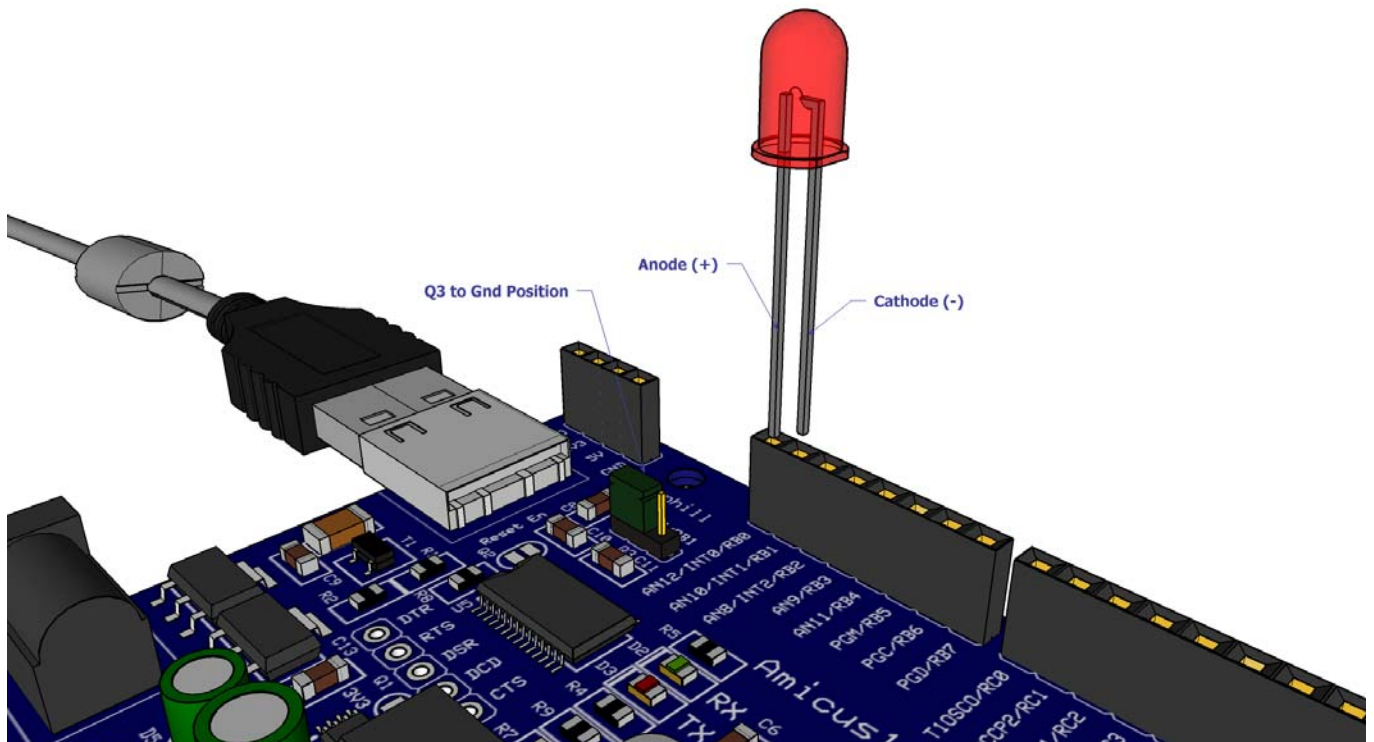
Connect the USB cable to the Amicus18 board, and make sure its red Power LED is illuminated. Press the *Compile and Program* button on the toolbar, or press F10. The code will then be compiled, and the bootloader will open to place the compiled code into the Amicus18's microcontroller. The LED will then start flashing.

The above layout works as expected, however, some rules have been broken in so much as the LED does not have a current limiting resistor in series with it. This means that the LED is seeing the full 3.3 Volts instead of it's working voltage of approx 2 Volts, and is pulling too much current from the microcontroller's IO pin. We can alleviate this situation by using the Companion Shield with a solderless breadboard.

The correct method for connecting an LED is shown overleaf.

Amicus18 Companion Shield for Proton
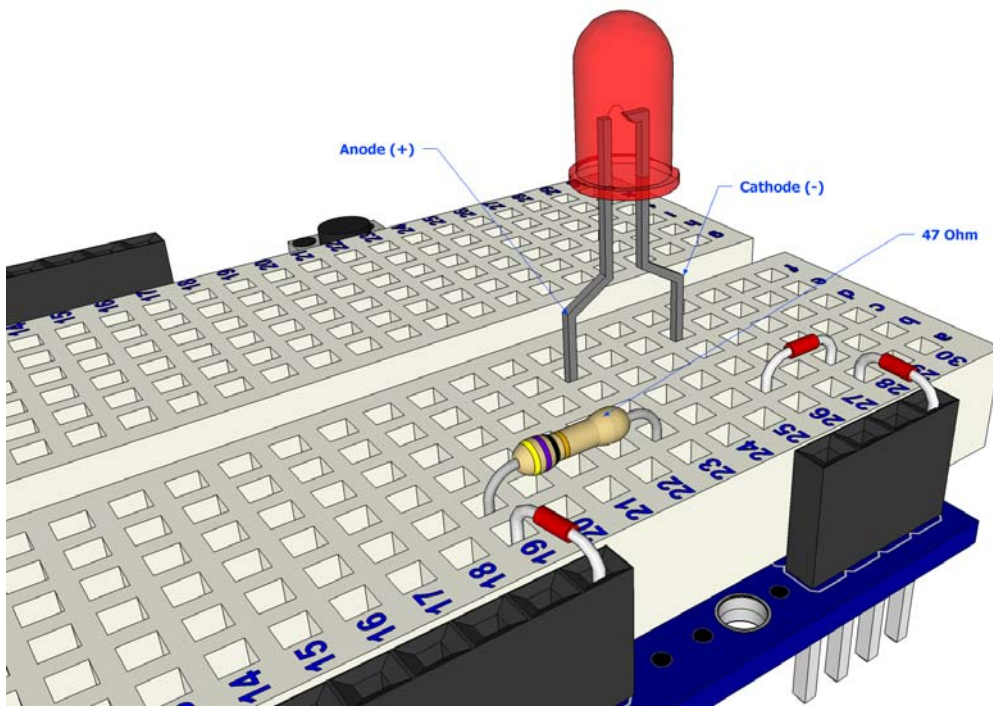
The circuit for the layout above is shown below:
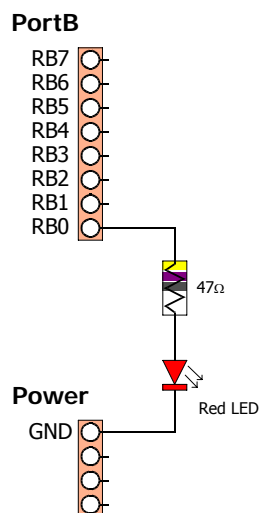
The same program may be used with the layout above, but this time the LED is protected from over voltage and over current.

```
' Flash an LED connected to RB0
' Make sure the Amicus18 board's jumper Q3 is set to the GND position
   Include "Amicus18.inc"   ' Configure the compiler to use the Amicus18 board
   While 1 = 1              ' Create an infinite loop
      High PORTB.0          ' Bring the LED pin high (illuminate the LED)
      DelayMs 500           ' Wait 500ms (half a second)
      Low PORTB.0           ' Pull the LED pin low (Extinguish the LED)
      DelayMs 500           ' Wait 500ms (half a second)
   Wend                     ' Close the loop
```

Remember that you do not need to type in the comments. i.e. the *blue* text following the ' character.

Once the program is typed into the IDE, press the toolbar's *Compile and Program* button to compile the code and place it into the Amicus18's microcontroller. As long as no typing errors have been made, the LED will then begin to flash. If any errors are found the offending line will be highlighted and an error message will be displayed on the bottom of the IDE.

06-10-2009

How to choose the resistor value

A resistor is a device designed to cause resistance to an electric current and therefore cause a drop in voltage across its terminals. If you imagine a resistor to be like a water pipe that is a lot thinner than the pipe connected to it. As the water (the electric current) comes into the resistor, the pipe gets thinner and the current coming out of the other end is therefore reduced. We use resistors to decrease voltage or current to other devices. The value of resistance is known as an Ohm and its symbol is a Greek Omega symbol Ω.

In this case Digital Pin RB0 is outputting 3.3 volts DC at 25mA (milliamps), and our LED requires a voltage of 2v and a current of 20mA. We therefore need to put in a resistor that will reduce the 3.3 volts to 2.2 volts, and the current from 25mA to 20mA if we want to display the LED at its maximum brightness. If we want the LED to be dimmer we could use a higher value of resistance.

To calculate what resistor we need to do this we use what is called "Ohms law" which is I = V/R where I is current, V is voltage and R is resistance. Therefore to work out the resistance we arrange the formula to be R = V/ I which is R = 1.1/0.02 which is 55 Ohms. V is 1.1 because we need the Voltage Drop, which is the supply voltage (3.3 volts) minus the Forward Voltage (2.2 volts) of the LED (found in the LED datasheet) which is 1.1 volts. We therefore need to find a 55Ω resistor. However, 55Ω resistors are not easily found, so we'll find a one close to it, 47 Ohms will do.

A resistor is too small to put writing onto that could be readable by most people so instead resistors use a colour code. Around the resistor you will typically find 4 coloured bands and by using the colour code in the chart on the next page you can find out the value of a resistor or what colour codes a particular resistance will be.

Colour	1st Band	2nd Band	3rd Band (multiplier)	4th Band (tolerance)
Black	0	0	x10^0	
Brown	1	1	x10^1	±1%
Red	2	2	x10^2	±2%
Orange	3	3	x10^3	
Yellow	4	4	x10^4	
Green	5	5	x10^5	±0.5%
Blue	6	6	x10^6	±0.25%
Violet	7	7	x10^7	±0.1%
Grey	8	8	x10^8	±0.05%
White	9	9	x10^9	
Gold			x10^-1	±5%
Silver			x10^-2	±10%
None				±20%

We need a 47Ω resistor, so if we look at the colour table we see that we need 4 in the first band, which is Yellow, followed by a 7 in the next band which is Violet and we then need to multiply this by 100 which is Black in the 3rd band. The final band is irrelevant for our purposes as this is the tolerance. Our resistor has a gold band and therefore has a tolerance of ±5% which means the actual value of the resistor can vary between 46.5Ω and 47.5Ω. We therefore need a resistor with a Yellow, Violet, Black, Gold colour band combination which looks like this:
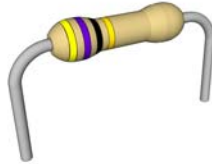
If we needed a 1K (or 1 kilo-ohm) resistor we would need a Brown, Black, Red combination (1, 0, +2 zeros). If we needed a 570K resistor the colours would be Green, Violet and Yellow.

In the same way, if you found a resistor and wanted to know what value it is you would do the same in reverse. So if you found this resistor and wanted to find out what value it was so you could store it away in your nicely labelled resistor storage box, we could look at the table to see it has a value of 220Ω.

The LED

The final component is an LED, which stands for Light Emitting Diode. A Diode is a device that permits current to flow in only one direction. So, it is just like a valve in a water system, but in this case it's letting electrical current to go in one direction, but if the current tried to reverse and go back in the opposite direction the diode would stop it from doing so. Diodes can be useful to prevent accidental connection of a Power supply in a circuit, and damaging the components.

An LED is the same thing, but it also emits light. LEDs come in all kinds of different colours and brightness's and can also emit light in the ultraviolet and infrared part of the spectrum (like in the LEDs within a TV remote control).

If you look carefully at the LED you will notice two things. One is that the legs are of different lengths and also that on one side of the LED, instead of it being cylindrical, it is flattened. These are indicators to show you which leg is the Anode (Positive) and which is the Cathode (Negative). The longer leg gets connected to the Positive Supply (3.3 volts) and the leg with the flattened side goes to Ground (Gnd).

If you connect the LED the wrong way, it will not damage it, but it is essential that you always place a resistor in series with the LED to ensure that the correct current gets to the LED. You can permanently damage the LED if you fail to do this.

As well as single colour LEDs you can also obtain bi-colour and tricolour LEDs. These will have several legs coming out of them with one of them being common (i.e. common anode or common cathode).

06-10-2009
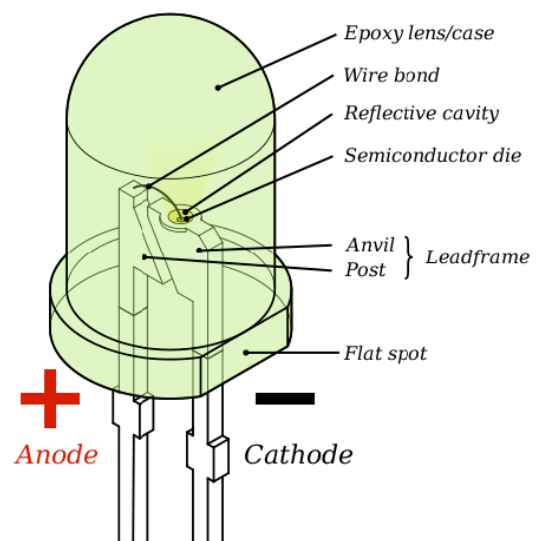
2 LED Flasher

Adding a second LED is simple, and the code for driving them is not too difficult either:
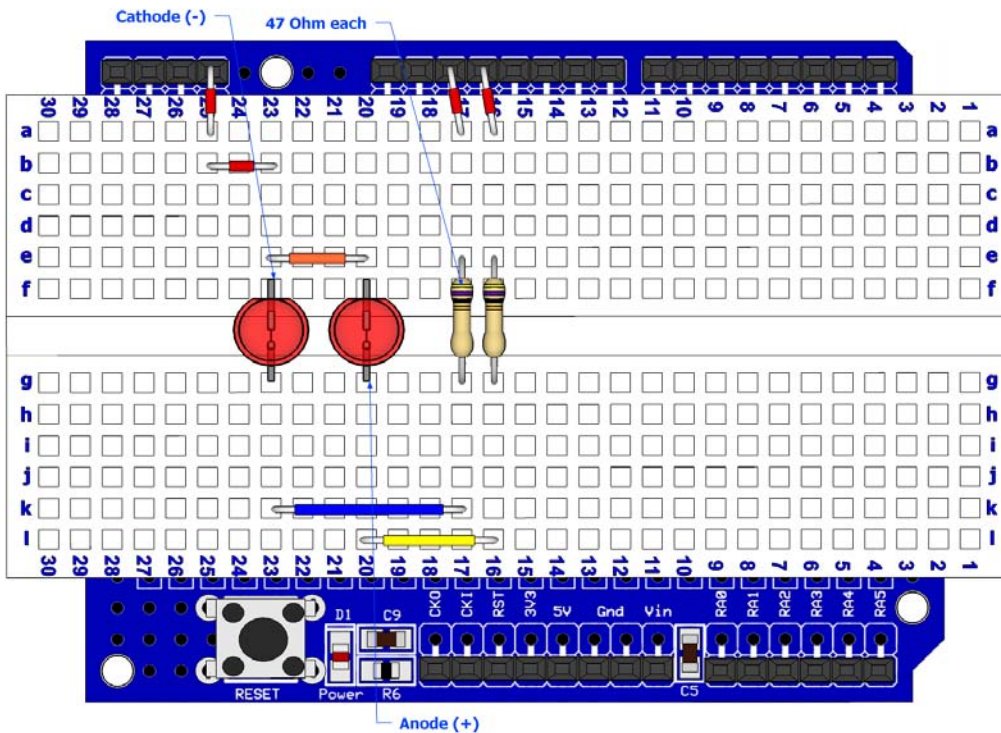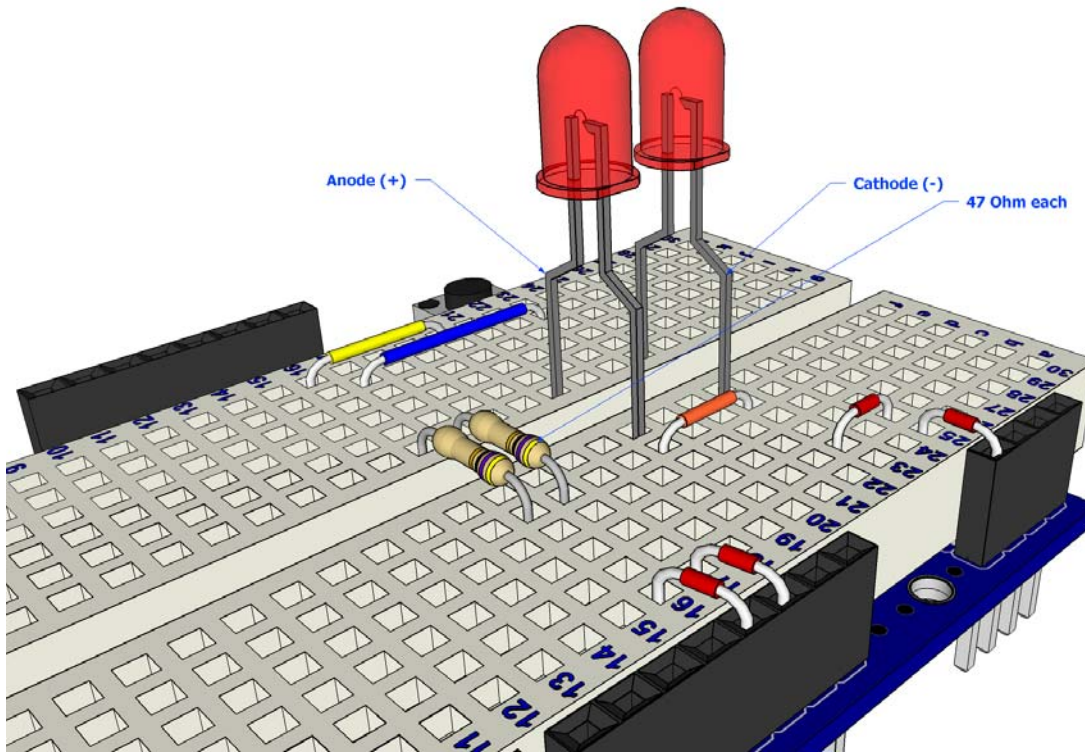
Amicus18 Companion Shield for Proton

The circuit for the two LED flasher layout is shown below:
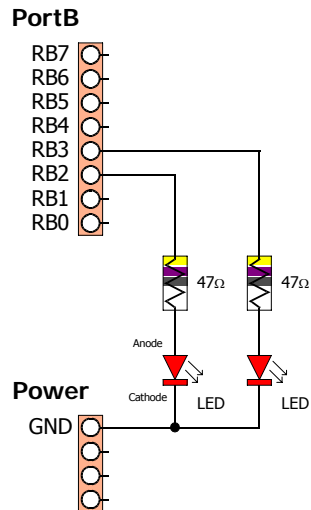
The code for driving the LEDs is shown below:

```
' Flash 2 LEDs connected to RB2 and RB3

  Include "Amicus18.inc"     ' Configure the compiler to use the Amicus18 board
  Symbol LED1 = PORTB.2      ' LED 1 is placed on pin-2 of PortB
  Symbol LED2 = PORTB.3      ' LED 2 is placed on pin-3 of PortB
  While 1 = 1                ' Create an infinite loop
    High LED1                ' Illuminate LED1
    DelayMS 500              ' Wait for half a second
    Low LED1                 ' Extinguish LED1
    High LED2                ' Illuminate LED2
    DelayMS 500              ' Wait for half a second
    Low LED2                 ' Extinguish LED2
  Wend                       ' Do it forever
```

4 LED Sequencer

Adding, and using, extra LEDs is also very simple, as illustrated below:
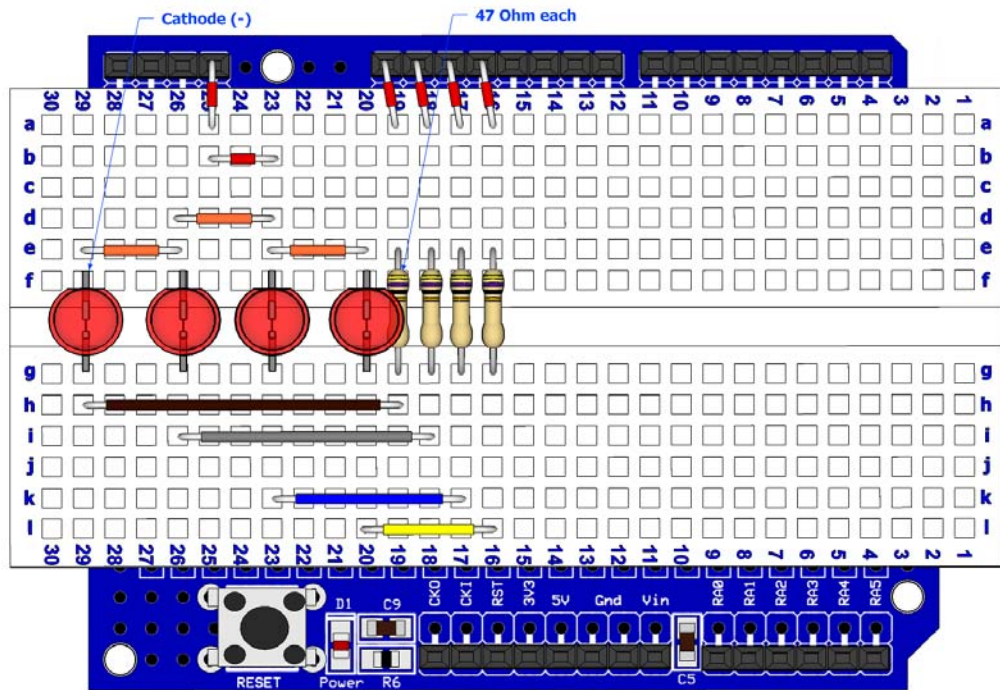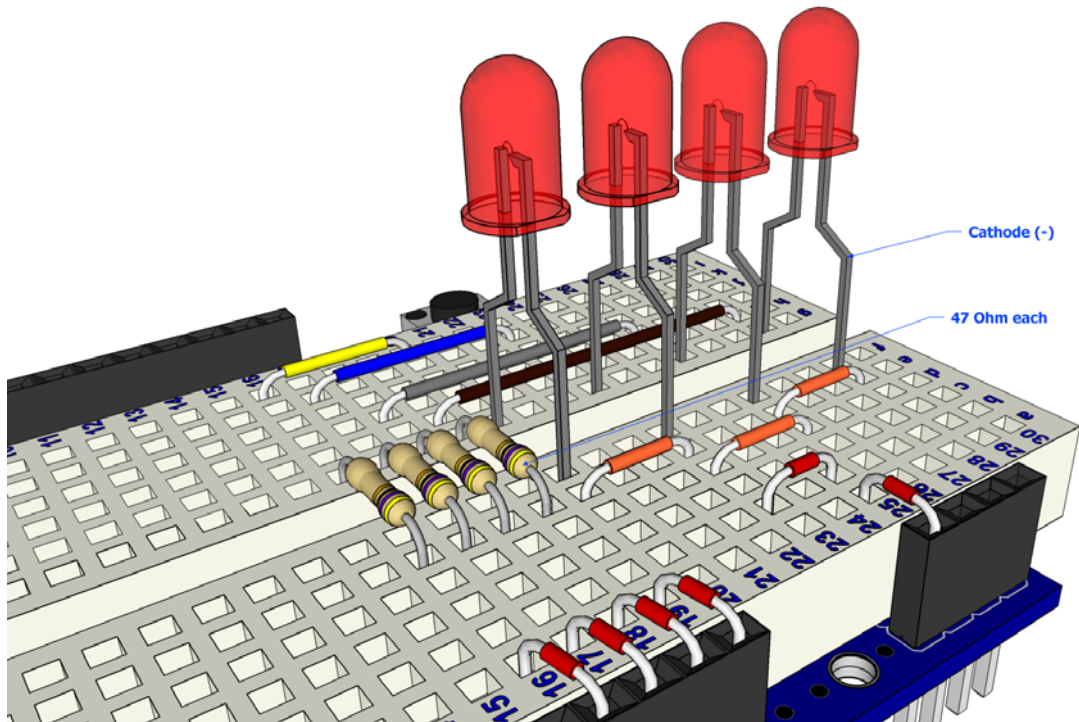
Amicus18 Companion Shield for Proton

The two extra LEDs are connected to RB0 and RB1 of PortB, as the circuit shows below:
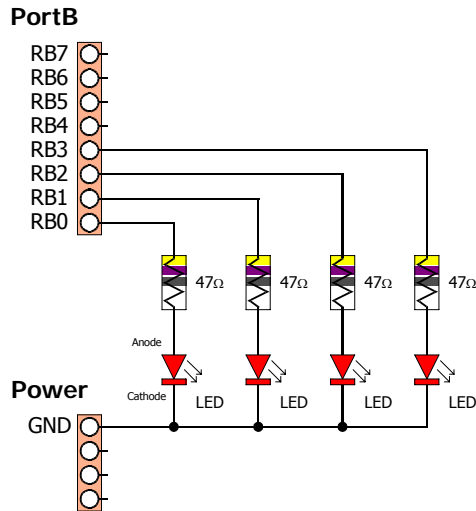
A suitable program for the 4 LED sequencer is shown below:

```
' Illuminate 4 LEDs attached to PortB in sequence
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
    Include "Amicus18.inc"   ' Configure the compiler to use the Amicus18 board
    Low PORTB                ' Make PortB output low (Extinguish all four LEDs)
    While 1 = 1              ' Create an infinite loop
       PORTB = %00000001     ' Illuminate the first LED
       DelayMS 300           ' Delay a pre-determined amount of time
       PORTB = %00000010     ' Illuminate the second LED
       DelayMS 300           ' Delay a pre-determined amount of time
       PORTB = %00000100     ' Illuminate the third LED
       DelayMS 300           ' Delay a pre-determined amount of time
       PORTB = %00001000     ' Illuminate the fourth LED
       DelayMS 300           ' Delay a pre-determined amount of time
    Wend                     ' Do it forever
```

The above program will illuminate each LED in turn.

Amicus18 Companion Shield for Proton

A more advanced program to do the same thing is shown below:

```
' Illuminate 4 LEDs attached to PortB in sequence
' Using a more advanced method
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
  Include "Amicus18.inc"       ' Configure the compiler to use the Amicus18 board

  Dim bPortShadow As Byte      ' Create a variable to hold the state of PortB
  Dim bLoop As Byte            ' Create a variable for the bit counting loop

  Low PORTB                    ' Make PortB output low (Extinguish all four LEDs)
  While 1 = 1                  ' Create an infinite loop
    bPortShadow = 1            ' Set the initial state of PortB
    PORTB = bPortShadow        ' Transfer the shadow variable to PortB
    DelayMS 300                ' Wait a pre-determined amount of time
    For bLoop = 0 To 3         ' Create a loop from 0 to 3
      bPortShadow = bPortShadow << 1 ' Shift a bit left one position
      PORTB = bPortShadow      ' Transfer the shadow variable to PortB
      DelayMS 300              ' Wait a pre-determined amount of time
    Next                       ' Close the loop
  Wend                         ' Do it forever
```

There are many variations of the programs that can be used with the four LED circuit. The program below sequences the LED's up then down the line.

```
' Illuminate 4 LEDs attached to PortB in sequence
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
  Include "Amicus18.inc"       ' Configure the compiler to use the Amicus18 board

  Dim bPortShadow As Byte      ' Create a variable to hold the state of PortB
  Dim bLoop As Byte            ' Create a variable for the bit counting loop

  bPortShadow = 1              ' Set the initial state of PortB
  Low PORTB                    ' Make PortB output low (Extinguish all four LEDs)
  PORTB = bPortShadow          ' Transfer the shadow variable to PortB
  DelayMS 300                  ' Wait a pre-determined amount of time
  While 1 = 1                  ' Create an infinite loop
    For bLoop = 0 To 3         ' Create a loop from 0 to 3
      bPortShadow = bPortShadow << 1 ' Shift a bit left one position
      PORTB = bPortShadow      ' Transfer the shadow variable to PortB
      DelayMS 300              ' Wait a pre-determined amount of time
    Next                       ' Close the loop
    For bLoop = 3 To 0 Step -1 ' Create a loop from 3 to 0
      bPortShadow = bPortShadow >> 1 ' Shift a bit right one position
      PORTB = bPortShadow      ' Transfer the shadow variable to PortB
      DelayMS 300              ' Wait a pre-determined amount of time
    Next                       ' Close the loop
  Wend                         ' Do it forever
```

15

8 LED Sequencer

A more sophisticated layout is shown below, in which eight LEDs are used. Notice how the use of different colour LEDs adds a new twist:
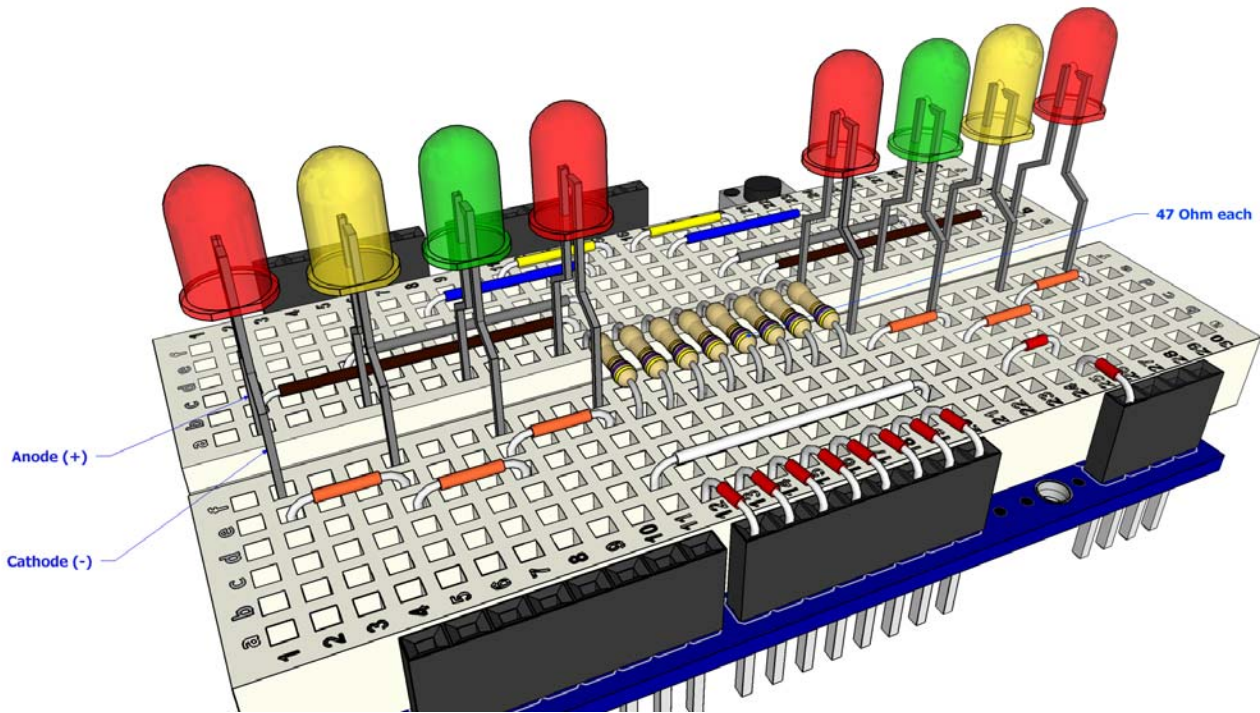
A top down view of the above layout is shown below for extra clarity:
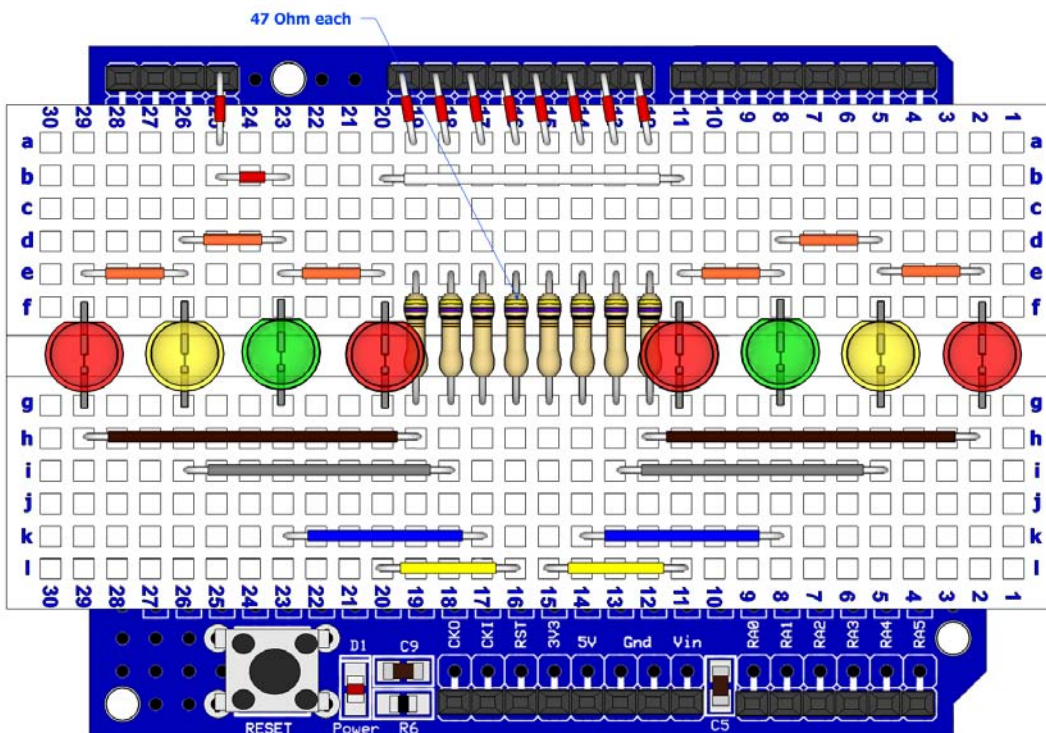
Note. Make sure the Amicus18's *Q3* jumper is set to the **RB1** position.

06-10-2009

Amicus18 Companion Shield for Proton

The circuit for the eight LED layout is shown below:
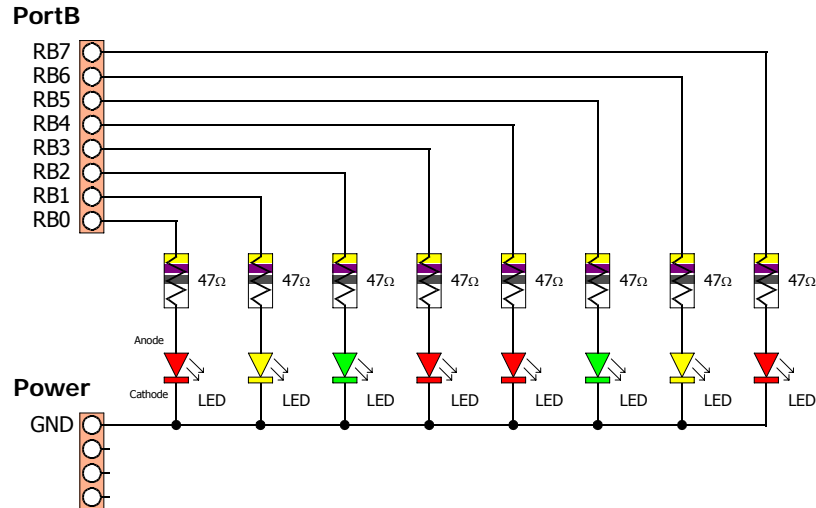
A suitable program for the 8 LED sequencer is shown below:

```
' Illuminate 8 LEDs attached to PortB in sequence
' Using discrete commands
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
  Include "Amicus18.inc"  ' Configure the compiler to use the Amicus18 board

  Low PORTB                ' Make PortB output low (Extinguish all the LEDs)
  While 1 = 1             ' Create an infinite loop
    PORTB = %00000001      ' Illuminate the first LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %00000010      ' Illuminate the second LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %00000100      ' Illuminate the third LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %00001000      ' Illuminate the fourth LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %00010000      ' Illuminate the fifth LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %00100000      ' Illuminate the sixth LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %01000000      ' Illuminate the seventh LED
    DelayMS 300            ' Delay a pre-determined amount of time
    PORTB = %10000000      ' Illuminate the eighth LED
    DelayMS 300            ' Delay a pre-determined amount of time
  Wend                     ' Close the loop
```

The above program will illuminate each LED in turn.

06-10-2009

Amicus18 Companion Shield for Proton

A more advanced program to do the same thing is shown below:

```
' Illuminate 8 LEDs attached to PortB in sequence
' Using a more advanced method
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board

  Dim bPortShadow As Byte       ' Create a variable to hold the state of PortB
  Dim bLoop As Byte             ' Create a variable for the bit counting loop

  Low PORTB                              ' Make PortB output low (Extinguish all the LEDs)
  While 1 = 1                            ' Create an infinite loop
    bPortShadow = 1                      ' Set the initial state of PortB
    PORTB = bPortShadow                  ' Transfer the shadow variable to PortB
    DelayMS 300                          ' Wait a pre-determined amount of time
    For bLoop = 0 To 6                   ' Create a loop from 0 to 6
      bPortShadow = bPortShadow << 1 ' Shift a bit left one position
      PORTB = bPortShadow                ' Transfer the shadow variable to PortB
      DelayMS 300                        ' Wait a pre-determined amount of time
    Next                                 ' Close the loop
  Wend                                   ' Do it forever
```

There are many variations of the programs that can be used with the eight LED circuit. The program below sequences the LED's up then down the line.

```
' Illuminate 8 LEDs attached to PortB in sequence
' Make sure the Amicus18 board's jumper Q3 is set to the RB1 position
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board

  Dim bPortShadow As Byte       ' Create a variable to hold the state of PortB
  Dim bLoop As Byte             ' Create a variable for the bit counting loop

  bPortShadow = 1               ' Set the initial state of PortB
  Low PORTB                     ' Make PortB output low (Extinguish all the LEDs)
  PORTB = bPortShadow           ' Transfer the shadow variable to PortB
  DelayMS 300                   ' Wait a pre-determined amount of time
  While 1 = 1                            ' Create an infinite loop
    For bLoop = 0 To 6                   ' Create a loop from 0 to 6
      bPortShadow = bPortShadow << 1 ' Shift a bit left one position
      PORTB = bPortShadow                ' Transfer the shadow variable to PortB
      DelayMS 300                        ' Wait a pre-determined amount of time
    Next                                 ' Close the loop
    For bLoop = 6 To 0 Step -1           ' Create a loop from 6 to 0
      bPortShadow = bPortShadow >> 1 ' Shift a bit right one position
      PORTB = bPortShadow                ' Transfer the shadow variable to PortB
      DelayMS 300                        ' Wait a pre-determined amount of time
    Next                                 ' Close the loop
  Wend                                   ' Do it forever
```

Traffic Light Sequencer

Using an adaptation of the 8 multi-coloured LED layout, we can create the sequence for a UK traffic light. The layout is shown below, notice that the only difference is the removal of four LEDs and four resistors:
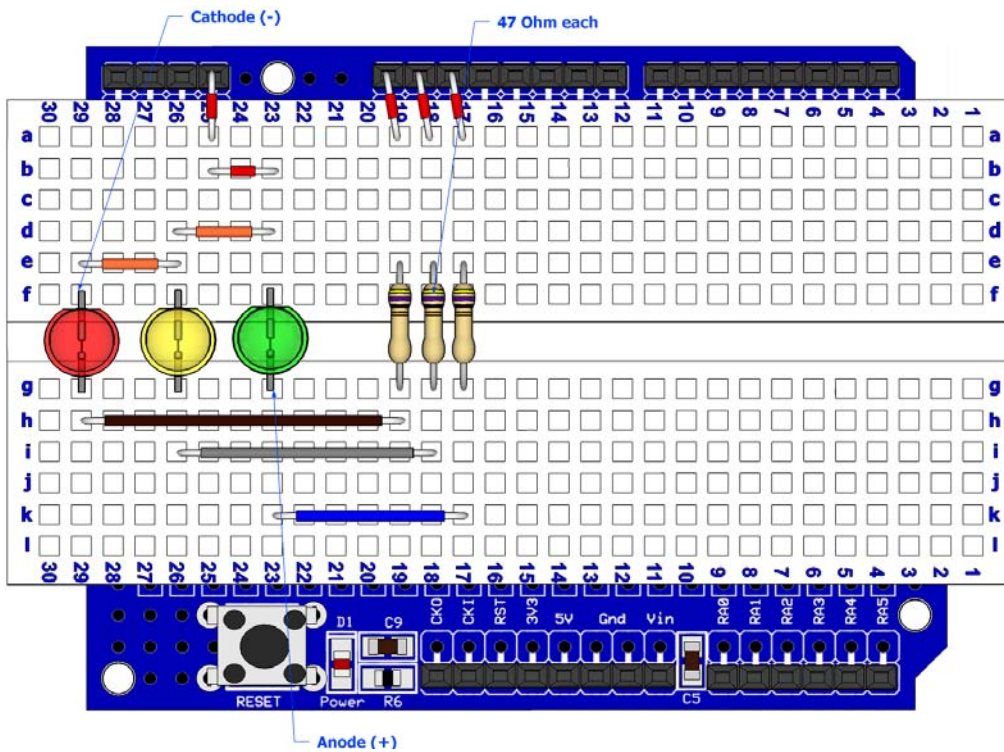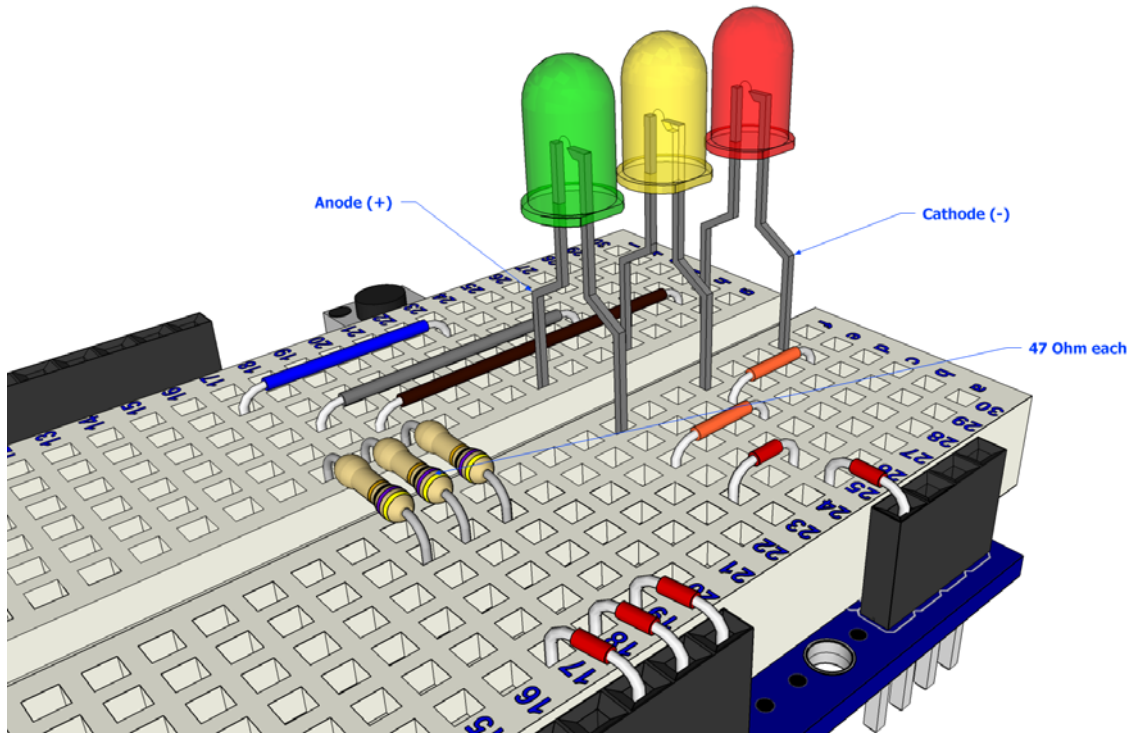
Amicus18 Companion Shield for Proton

The circuit for the traffic light sequencer is shown below:
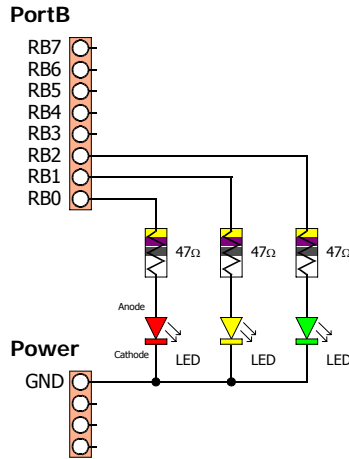
The sequence of traffic lights in the UK is shown below:
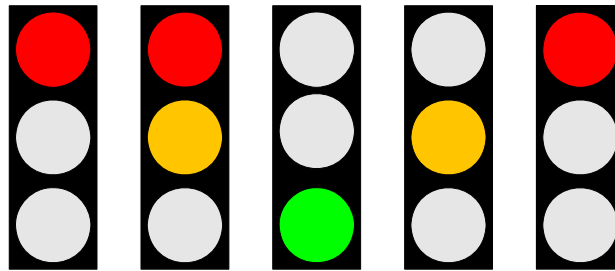
The program below shows the steps required to reproduce the sequence of lights shown above:

```
' Simulate a single traffic light using Red, Yellow, and Green LEDs
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
  Symbol Red = PORTB.0          ' Red LED is attached to RB0
  Symbol Amber = PORTB.1        ' Amber LED is attached to RB1
  Symbol Green = PORTB.2        ' Green LED is attached To RB2
  Symbol RedInterval = 4000     ' Time that the Red light will stay on
' Time that the Red and Amber lights will stay on
  Symbol AmberRedInterval = RedInterval / 4
' Time that the Amber light will stay on
  Symbol AmberInterval = RedInterval - AmberRedInterval
  Symbol GreenInterval = 6000   ' Time that the Green light will stay on
  While 1 = 1                   ' Create an infinite loop
    High Red                    ' Illuminate the Red LED
    DelayMS RedInterval         ' Wait for the appropriate length of time
    High Amber                  ' Illuminate the Amber LED
    DelayMS AmberRedInterval    ' Wait for the appropriate length of time
    Low Red                     ' Extinguish the Red LED
    DelayMS AmberInterval       ' Wait for the appropriate length of time
    High Green                  ' Illuminate the Green LED
    Low Amber                   ' Extinguish the Amber LED
    DelayMS GreenInterval       ' Wait for the appropriate length of time
    Low Green                   ' Extinguish the Green LED
    High Amber                  ' Illuminate the Amber LED
    DelayMS AmberInterval       ' Wait for the appropriate length of time
    Low Amber                   ' Extinguish the Amber LED
  Wend                          ' Do it forever
```

Type in the program above, remembering that you do not need to type in the comments. Click on the toolbar *Compiler and Program* button or press **F10** to compile the code and load it into the Amicus18's microcontroller. The three LEDs will then start sequencing.

Sensing the Outside World

Interacting with the outside world is always desirable when using a microcontroller, whether it's choosing a drink in a vending machine or deciding which way a pacman will move. The easiest method of outside influence is through the use of a switch or button.

However, there are certain rules that must be observed when adding a switch to a microcontroller's pin. When the pin is configured as an input, it can be brought high to 3.3 Volts or pulled low to ground, however if neither of these states is performed, the pin is neither high or low and this is termed *floating*. Even if a switch was placed from the microcontroller's input pin to ground, when the switch is not being operated the input pin can be high or low (*floating*).

What's required is a pull-up resistor or a pull-down resistor in order to force a single state when not in use. A pull-up resistor is a weak resistance from the input pin to the 3.3 Volt line, while a pull-down resistor is a weak resistance from the input pin to ground.

Switch Input (Pulled-Up)

The layout below shows a pull-up resistance:
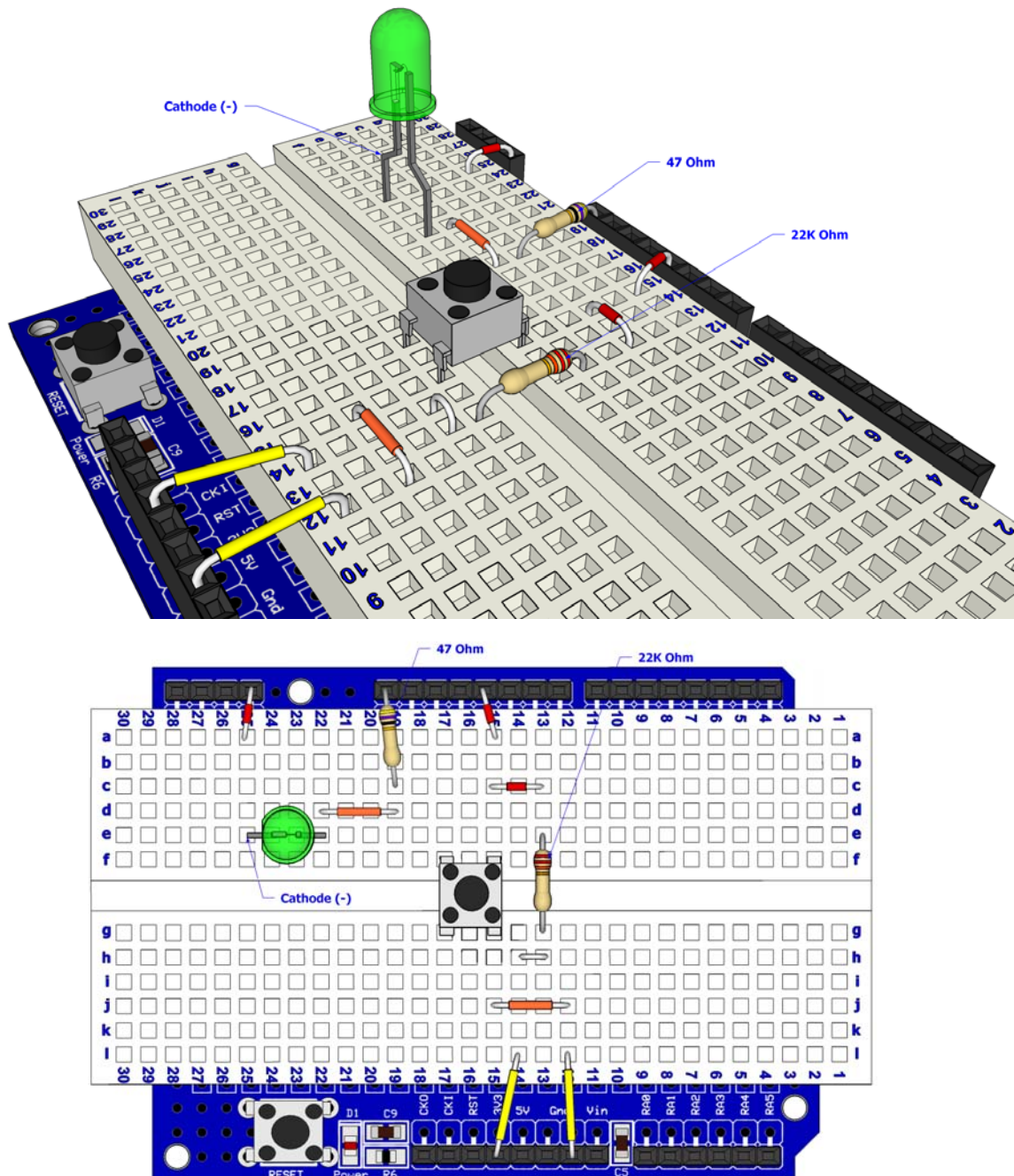
Amicus18 Companion Shield for Proton

The circuit for the pulled-up switch input is shown below:
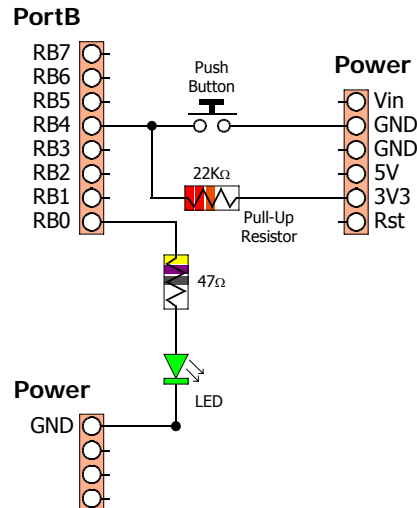
Open the Amicus IDE and type in the following program, or copy and paste from here:

```
' Demonstrate a switch input using a pull-up resistor
' Display state of the input pin RB4 when a push-button switch is operated
'
  Include "Amicus18.inc"          ' Configure the compiler to use the Amicus18 board
  Symbol Switch = PORTB.4         ' Button is connected to RB4 (PortB.4)

  Input Switch                    ' Make the button pin an input
  While 1 = 1                     ' Create an infinite loop
    HRSOut "Button = ", Bin1 Switch, 13 ' Display the input state
    DelayMS 500                   ' Delay for half a second
  Wend                            ' Do it forever
```

Click the toolbar icon *Compile and Program* or press **F10** to build the code and place it into the Amicus18's microcontroller.

Open the Amicus Serial Terminal by clicking on the toolbar, and open a connection to the Amicus18. Use the default baud of 9600. The serial terminal's window should show the text "Button = 1". This is displaying the state of the pin where the button is attached. Press the button and the test will change to "Button = 0":
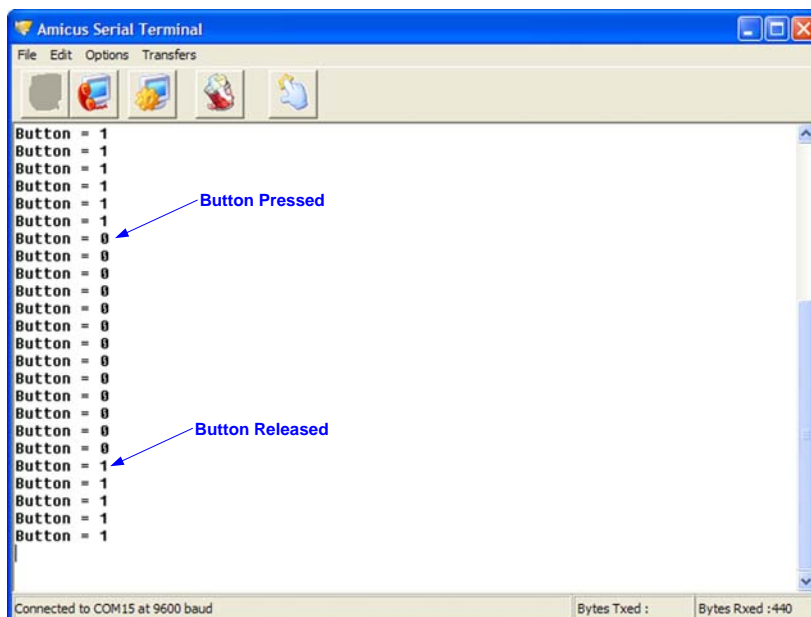
Notice how the state of the pin is 0 when the button is pressed. This is because the weak pull-up resistor (22KΩ) holds the pin to 3.3 Volts when it's not being operated, and the button pulls the pin to ground when it's operated.

Now that we know that the pin's state is 0 when the button is operated, decisions can be made upon it.

The program below will flash the LED 10 times when the button is pressed:

```
' Demonstrate a switch input using a pull-up resistor
' Flash an LED based upon a button press
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
  Dim Flash As Byte             ' Holds the amount of flashes
  Symbol Switch = PORTB.4       ' Button is connected to RB4 (PortB.4)
  Symbol LED = PORTB.0          ' LED attached to RB0

  GoTo Main                     ' Jump over the subroutine
'---------------------------------
' Subroutine to flash the LED
FlashLED:
  For Flash = 0 To 9            ' Create a loop of 10 iterations
    High LED                    ' Illuminate the LED
    DelayMS 100                 ' Wait 100 milliseconds
    Low LED                     ' Extinguish the LED
    DelayMS 100                 ' Wait 100 milliseconds
  Next                          ' Close the loop
  Return                        ' Exit the subroutine
'---------------------------------
' Main program starts here
Main:
  Input Switch                  ' Make the button pin an input
  While 1 = 1                   ' Create an infinite loop
    If Switch = 0 Then          ' Is the button pressed ?
      GoSub FlashLED            ' Yes. So flash the LED
    EndIf
  Wend                          ' Do it forever
```

Switch Input (Pulled-Down)

The layout below shows a pull-down resistance:

The circuit for the above layout is shown below:

PortB

RB7
RB6
RB5
RB4   22KΩ   **Power**
RB3   Pull-Down   Vin
RB2   Resistor   GND
RB1   GND
RB0   5V
                         3V3
        Push   Rst
        Button

        47Ω

Power
        LED
GND

Open the Amicus IDE and type in the following program, or copy and paste from here:

```
' Demonstrate a switch input using a pull-down resistor
' Display state of the input pin RB4 when a push-button switch is operated
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
  Symbol Switch = PORTB.4       ' Button is connected to RB4 (PortB.4)

  Input Switch                  ' Make the button pin an input
  While 1 = 1                   ' Create an infinite loop
    HRSOut "Button = ", Bin1 Switch, 13   ' Display the input state
    DelayMS 500                 ' Delay for half a second
  Wend                          ' Do it forever
```

Click the toolbar icon *Compile and Program* or press **F10** to build the code and place it into the Amicus18's microcontroller.

Open the Serial Terminal by clicking on the toolbar, and open a connection to the Amicus18. Use the default baud of 9600. The serial terminal's window should show the text "Button = 0". This is displaying the state of the pin where the button is attached. Press the button and the test will change to "Button = 1":
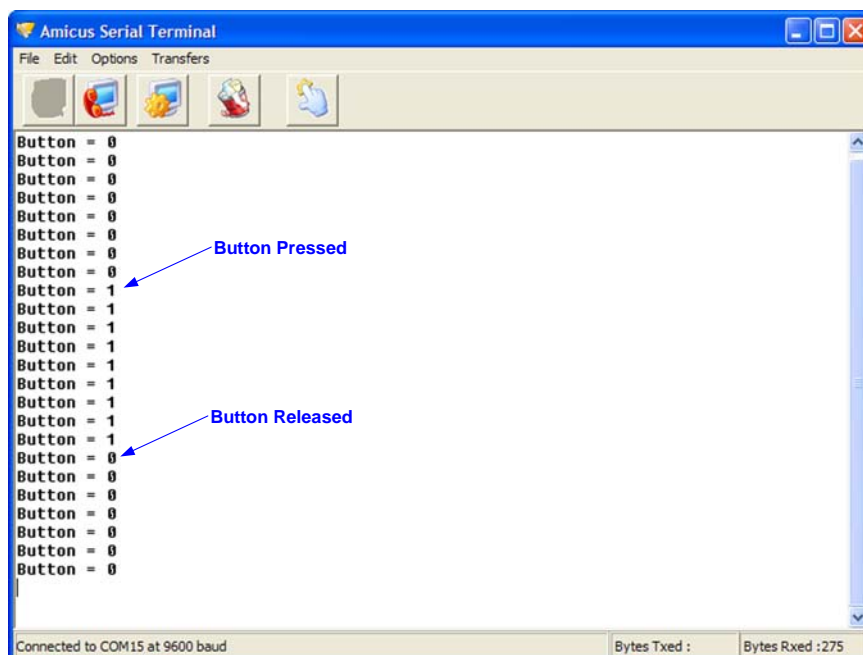
Notice how the state of the pin is 1 when the button is pressed. This is because the weak pull-up resistor (22KΩ) holds the pin to ground when it's not being operated, and the button pulls the pin to 3.3 Volts when it's operated. This is the exact opposite of using a pull-up resistor.

Now that we know that the pin's state is 1 (high) when the button is operated, decisions can be made upon it.

The program below will flash the LED 10 times when the button is pressed:

```
' Demonstrate a switch input using a pull-down resistor
' Flash an LED based upon a button press
'
    Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
    Dim Flash As Byte             ' Holds the amount of flashes
    Symbol Switch = PORTB.4       ' Button is connected to RB4 (PortB.4)
    Symbol LED = PORTB.0          ' LED attached to RB0

    GoTo Main                     ' Jump over the subroutine
'--------------------------------
' Subroutine to flash the LED
FlashLED:
    For Flash = 0 To 9            ' Create a loop of 10 iterations
        High LED                  ' Illuminate the LED
        DelayMS 100               ' Wait 100 milliseconds
        Low LED                   ' Extinguish the LED
        DelayMS 100               ' Wait 100 milliseconds
    Next                          ' Close the loop
    Return                        ' Exit the subroutine
'--------------------------------
' Main program starts here
Main:
    Input Switch                  ' Make the button pin an input
    While 1 = 1                   ' Create an infinite loop
        If Switch = 1 Then        ' Is the button pressed ?
            GoSub FlashLED        ' Yes. So flash the LED
        EndIf
    Wend                          ' Do it forever
```
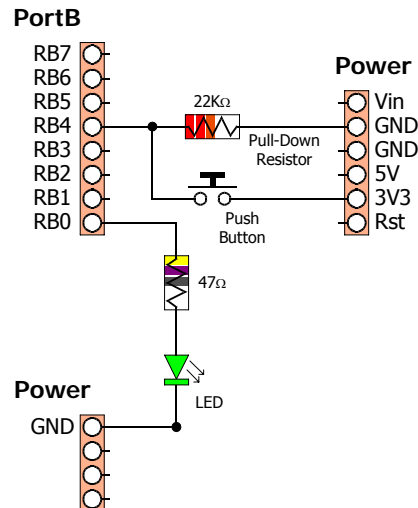
06-10-2009

Switch Debounce

Mechanical switches are frequently encountered in embedded processor applications, and are inexpensive, simple, and reliable. However, such switches are also often very electrically noisy. This noise is known as switch bounce, whereby the connection between the switch contacts makes and breaks several, perhaps even hundreds, of times before settling to the final switch state. This can cause a single switch push to be detected as several distinct switch pushes by the fast microcontroller used in the Amicus18 board, especially with an edge-sensitive input. Think of advancing the TV channel, but instead of getting the next channel, the selection skips ahead two or three.

Classic solutions to switch bounce involved low pass filtering out of the fast switch bounce transitions with a resistor-capacitor circuit, or using re-settable logic shift registers. While effective, these methods add additional cost and increase circuit board complexity. Debouncing a switch in software eliminates these issues.

A simple way to debounce a switch is to sample the switch until the signal is stable. How long to sample requires some investigation of the switch characteristics, but usually 5ms is sufficiently long.

The following code demonstrates sampling the switch input every 1mS, waiting for 5 consecutive samples of the same value before determining that the switch was pressed. Note that the tactile switches used for the layouts don't bounce much, but it is good practice to debounce all system switches.

```
' Debounce a switch input (Pulled-Up)
' The LED will toggle On and Off whenever the switch is pressed
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board

  Dim Switch_Count As Byte      ' Holds the switch counter amounts
  Symbol DetectsInARow = 5      ' The amount of counts to perform

  Symbol Switch_Pin = PORTB.4 ' Pin where the switch is connected
  Symbol LED = PORTB.0          ' Pin where the LED is connected

Main:
  Low LED                       ' Extinguish the LED
  Input Switch_Pin              ' Make the switch pin a input

  While 1 = 1                   ' Create an infinite loop
    While Switch_Pin <> 1 : Wend   ' Wait for switch to be released (Pulled-Up)

    Switch_Count = 5
    Repeat                 ' Monitor switch input for 5 lows in a row to debounce
      If Switch_Pin = 0 Then      ' Pressed state detected ?
        Inc Switch_Count          ' Yes. So increment the counter
      Else                        ' Otherwise...
        Switch_Count = 0          ' Reset the counter
      EndIf
      DelayMS 1                   ' Wait for 1ms
    Until Switch_Count >= DetectsInARow ' Exit when 5 iterations have been performed

    Toggle LED                    ' Toggle the LED On/Off
  Wend                            ' Do it forever
```

Amicus18 Companion Shield for Proton

The same layout as the pulled-up switch demonstration can be used:
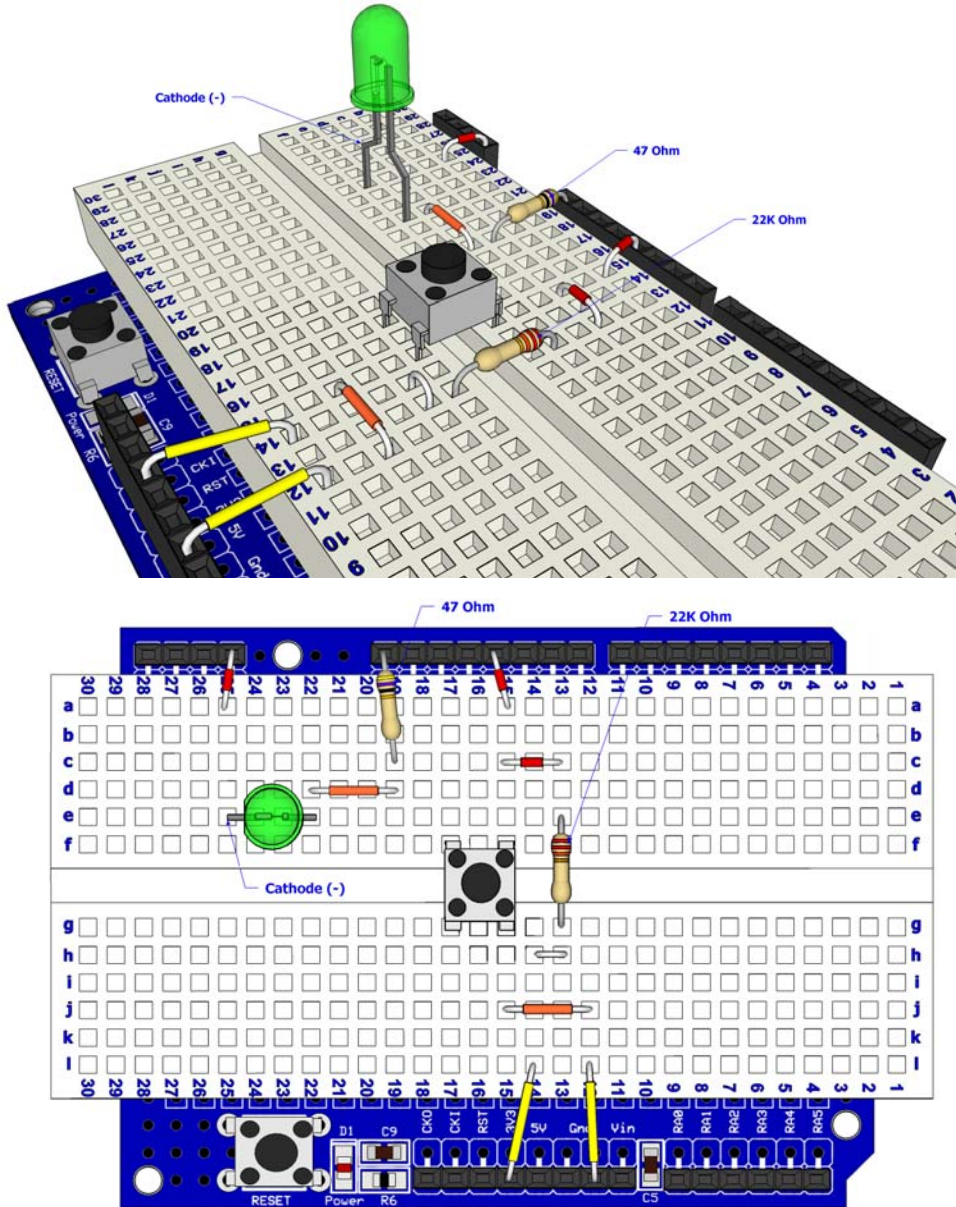
The circuit for the debounced pulled-up switch input is shown below:
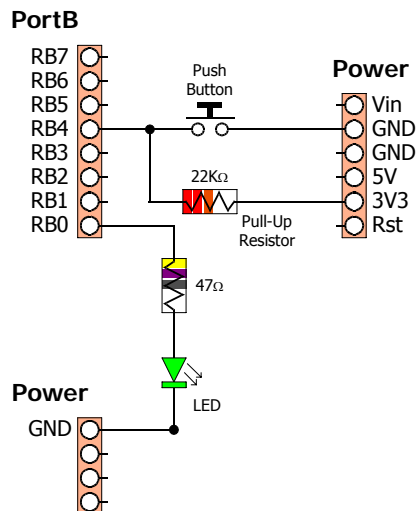
Amicus18 Companion Shield for Proton

In order to detect and debounce a switch that is pulled down to ground through a resistor, the following code can be used. It's essentially the same program as the pulled up version, but references to 0 now reference 1, and vice-versa:

```
' Debounce a switch input (Pulled-Down)
' The LED will toggle On and Off whenever the switch is pressed
'
    Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board

    Dim Switch_Count As Byte      ' Holds the switch counter amounts
    Symbol DetectsInARow = 5      ' The amount of counts to perform

    Symbol Switch_Pin = PORTB.4   ' Pin where the switch is connected
    Symbol LED = PORTB.0          ' Pin where the LED is connected
Main:
    Low LED                       ' Extinguish the LED
    Input Switch_Pin              ' Make the switch pin a input

    While 1 = 1                   ' Create an infinite loop
        While Switch_Pin <> 0 : Wend   ' Wait for switch to be released (Pulled Down)

        Switch_Count = 5
        Repeat                    ' Monitor switch input for 5 highs in a row to debounce
            If Switch_Pin = 1 Then     ' Pressed state detected ?
                Inc Switch_Count       ' Yes. So increment the counter
            Else                       ' Otherwise...
                Switch_Count = 0       ' Reset the counter
            EndIf
            DelayMS 1                  ' Wait for 1ms
        Until Switch_Count >= DetectsInARow ' Exit when 5 iterations have been performed

        Toggle LED                ' Toggle the LED On/Off
    Wend                          ' Do it forever
```

29

Amicus18 Companion Shield for Proton

Analogue Meets Digital

Not everything in the microcontroller world is made up of ons or offs, sometimes the input required is of an analogue nature i.e. a voltage. This is where an Analogue to Digital Converter (ADC) comes into it's own. An ADC samples the incoming voltage and converts it to a binary representation. The Amicus18 has nine ADC inputs, each capable of producing a 10-bit sample (0 to 1023). The ADC can measure resistance, current, sound, in fact anything that has a voltage.

To illustrate the use of the ADC peripheral, use the layout below:
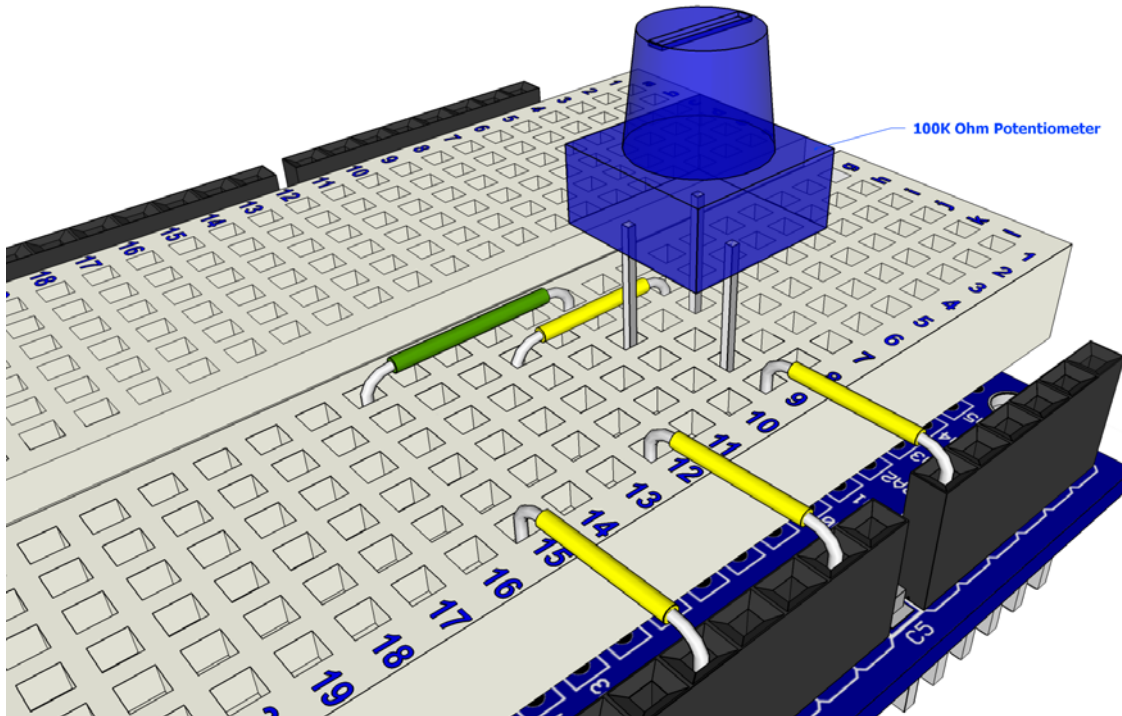
100K Ohm Potentiometer

The circuit for the above layout is shown below:

PortA
AN4/AN5
RA4
AN3/RA3
AN2/RA2
AN1/RA1
AN0/RA0

Power
Vin
GND
GND
5V
3V3
Rst

100KΩ Potentiometer
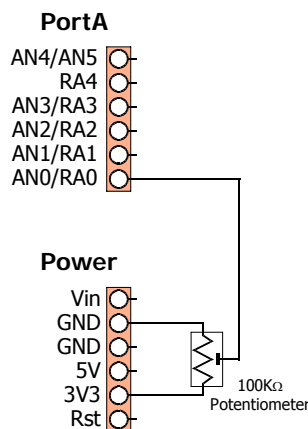
The program for the ADC demonstration is shown below:

```
' Demonstrate an ADC (Analogue to Digital Converter) input
' Display the state of AN0 (Channel 0 of the ADC) on the serial terminal
'
  Include "Amicus18.inc"      ' Configure the compiler to use the Amicus18 board
  Dim ADC_Input As Word       ' Create a variable to hold the 10-bit ADC result
  Include "Amicus18_ADC.inc"  ' Load the Amicus18 ADC macros into the program
'
' Open the ADC:
'               Fosc set for Fosc/32
'               Right justified for 10-bit operation
'               Tad value of 2
'               Vref+ at Vcc : Vref- at Gnd
'               Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                               ' Create an infinite loop
     ADC_Input = ReadADC(ADC_CH0)           ' Read the ADC from channel AN0
     HRSOut "ADC = ", Dec ADC_Input, 13     ' Display the ADC value
     DelayMS 500                            ' Delay for half a second
  Wend                                      ' Do it forever
```
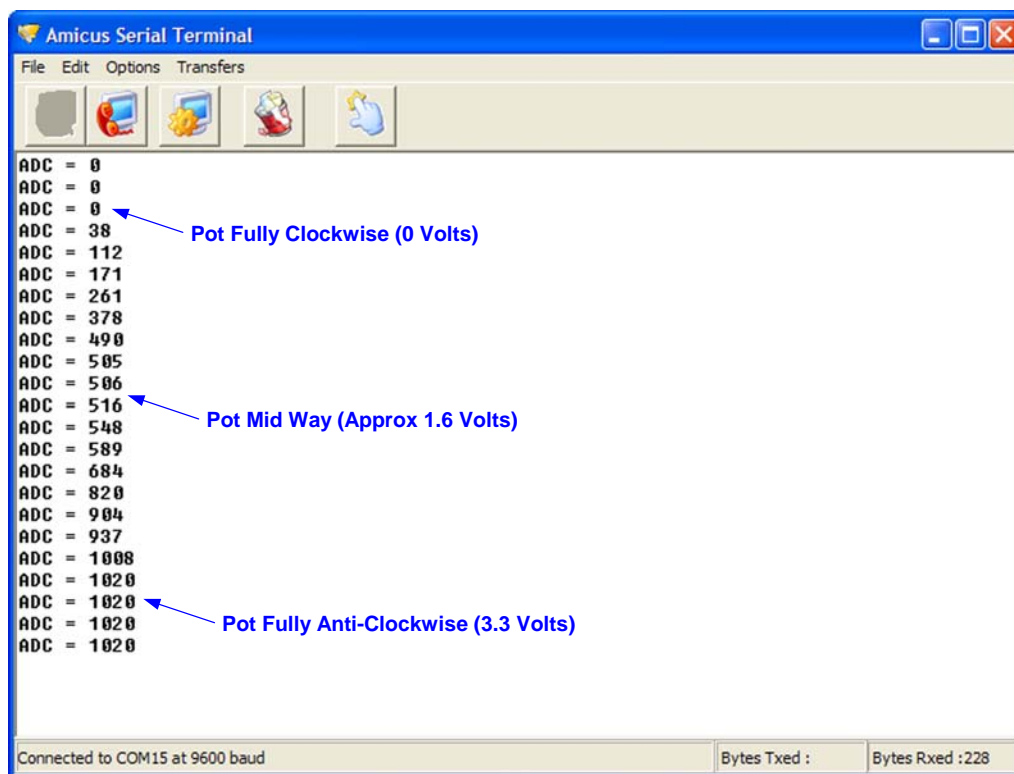
Once the program is compiled and loaded into the Amicus18 board by clicking on the toolbar *Compile and Program* or pressing **F10**, open the serial terminal and connect to the Amicus18 board's com port:

Turning the potentiometer anti-clockwise will increase the voltage to the ADC, therefore increasing the ADC's value. Turning the potentiometer clockwise will decrease the voltage to the ADC, and decrease the ADC's value, as can be seen from the screenshot above.

Don't worry too much is the ADC value isn't exactly 1023 for 3.3 Volts, as it's only a tiny fraction of the actual value, and this will make very little difference, if any, to most programs. This can be caused by many things, wrong *Tad* being used, wrong *Fosc*, losses in the wiring etc…

06-10-2009

Light Level Switch (Cockroach Mode)

We can use the ADC for a more practical example now that we know it works. We'll use an LDR (Light Dependant Resistor) as the input to the ADC, and turn on an LED when the light level drops beyond a certain level.

An LDR, as it's name suggests, alters it's resistance depending on the amount of light falling upon it. It's one of the oldest methods of light detection, and one of the simplest of all light level detectors to use, and one of least expensive.

And LDR layout is shown below:
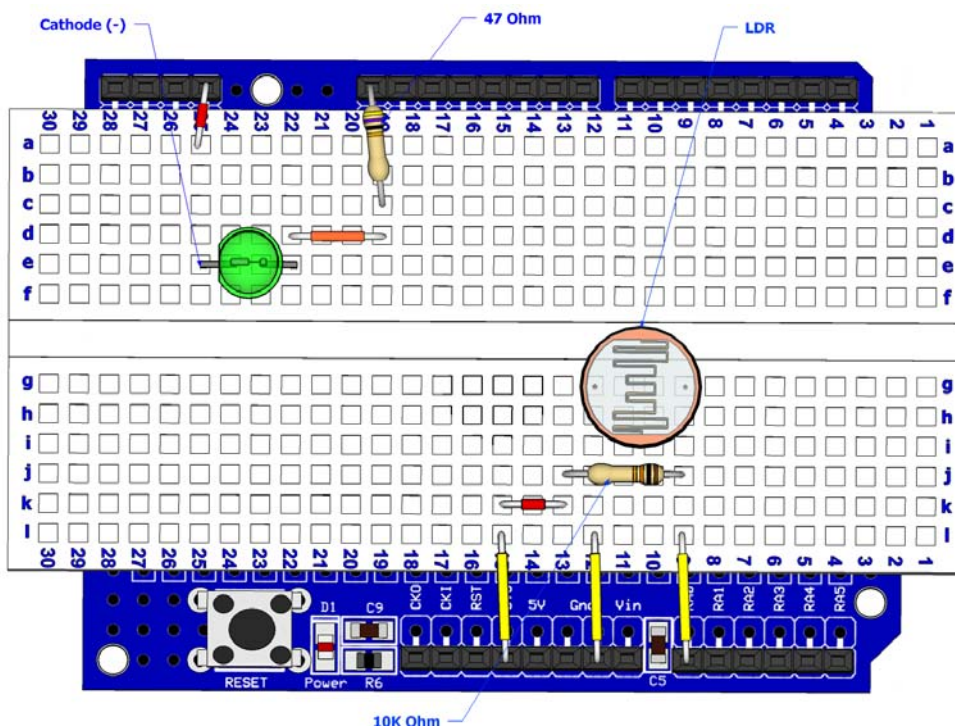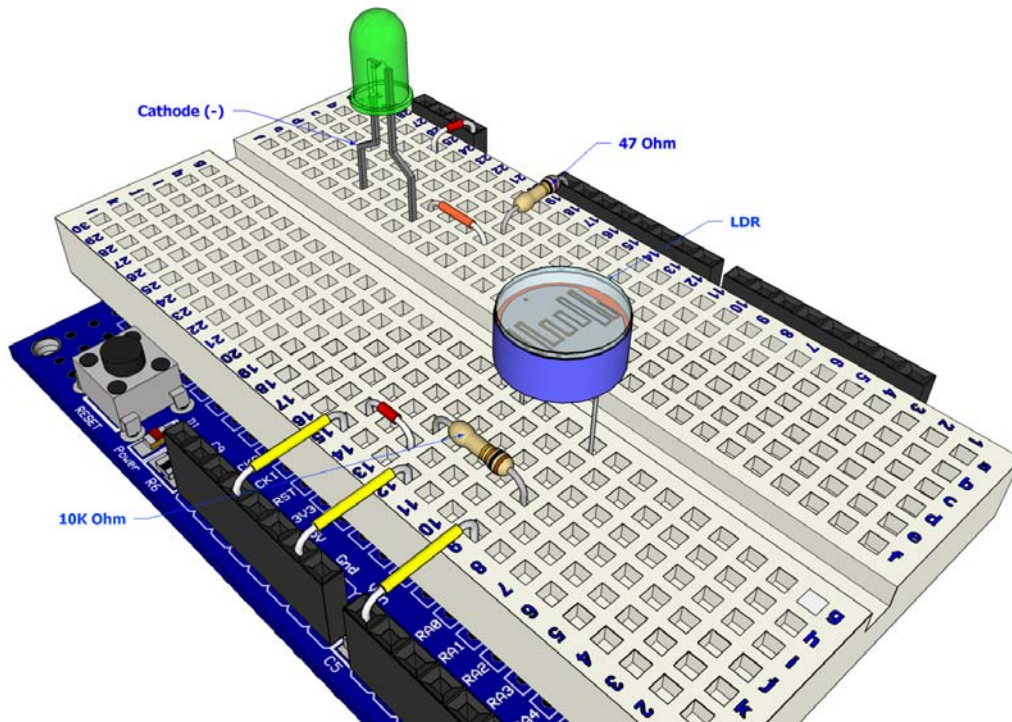
Amicus18 Companion Shield for Proton

Don't worry if the LDR you use doesn't look like the one used in the layout as LDRs come in all shapes and sizes, but they all perform the same task. However, their light level resistance may vary. But again, this doesn't actually matter, as we'll be detecting changes in light level, not the level itself .

The circuit for the LDR layout is shown below:
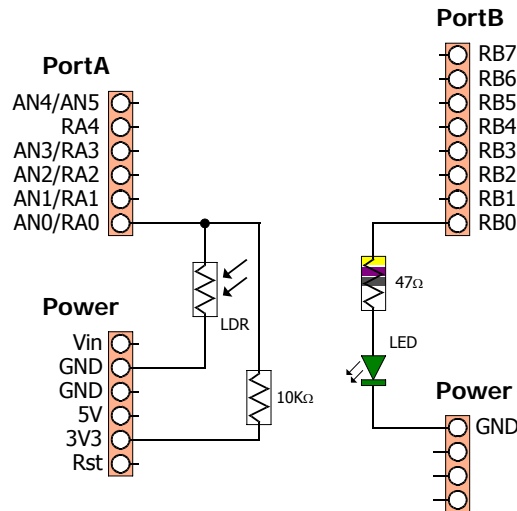
The program for the Light Level Detector is shown below. The code will activate the LED when the LDR sees a certain level of darkness, just like a cockroach:

```
' Illuminate an LED when an LDR connected to AN0 sees darkness
'
' Altering the value within the If-Then condition will set the light level threshold
' Any value from 0 to 1023 is valid, however, larger values indicate darkness
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board

  Dim LDR_Value As Word         ' Holds the 10-bit ADC value from the LDR
  Symbol LED = PORTB.0          ' Pin where the LED is connected. i.e. bit-0 of PortB

  Include "Amicus18_ADC.inc"    ' Load the Amicus18 ADC macros into the program
'
' Open the ADC:
'              Fosc set for Fosc / 32
'              Right justified for 10-bit operation
'              Tad value of 2
'              Vref+ at Vcc : Vref- at Gnd
'              Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                   ' Create an infinite loop
    LDR_Value = ReadADC(ADC_CH0)    ' Read the ADC value from AN0
    If LDR_Value > 400 Then     ' Is the ADC value above 400. i.e. getting darker
      High LED                  ' Yes. So illuminate the LED
    Else                        ' Otherwise...
      Low LED                   ' Extinguish the LED
    EndIf
    DelayUS 30                  ' Allow the ADC to recover
  Wend                          ' Do it forever
```

In order to change the level of darkness that the LDR will react too is simply a matter of changing the value within the line "**If** LDR_Value > *400* **Then**". A larger value will illuminate the LED at darker levels. The best way to calibrate the program is to examine the values produced by your particular LDR in light an dark situations. The program below will display the LDR values on the serial terminal:

```
' Display the value produced from an LDR
'
  Include "Amicus18.inc"      ' Configure the compiler to use the Amicus18 board
  Dim LDR_Value As Word       ' Holds the 10-bit ADC value from the LDR
  Include "Amicus18_ADC.inc"  ' Load the Amicus18 ADC macros into the program
'
' Open the ADC:
'              Fosc Set For internal RC Oscillator
'              Right justified for 10-bit operation
'              Tad value of 2
'              Vref+ at Vcc : Vref- at Gnd
'              Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_RC & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                         ' Create an infinite loop
    LDR_Value = ReadADC(ADC_CH0)      ' Read the ADC value from AN0
    HRSOut Dec LDR_Value, 13          ' Display the value on the serial terminal
    DelayMS 500                       ' Wait half a second
  Wend                                ' Do it forever
```

Once the code is compiled and loaded into the Amicus18, open the serial teminal:
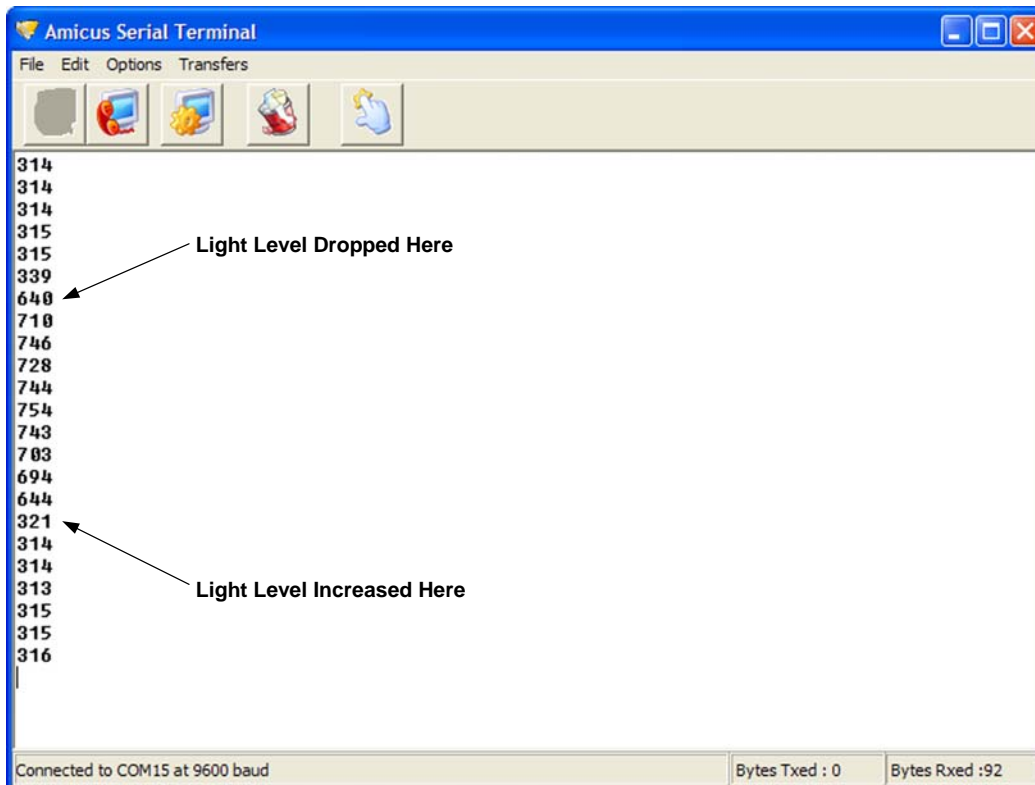
As can be seen from the above screenshot, ambient light levels give an approximate value of 315, so anything above this value will indicate a light level decrease. However, we don't want to make it too sensitive, so a value of 400 is ideal.

Light Level Switch (Moth Mode)

The same circuit and layout is used for the opposite reaction to light levels. The code below will illuminate the LED when light levels increase, just like a moth to a flame.

```
' Illuminate an LED when an LDR connected to AN0 sees light
' Altering the value within the If-Then condition will set the dark level threshold
' Any value from 0 to 1023 is valid, however, smaller values indicate lightness
'
  Include "Amicus18.inc"     ' Configure the compiler to use the Amicus18 board

  Dim LDR_Value As Word      ' Holds the 10-bit ADC value from the LDR
  Symbol LED = PORTB.0        ' Pin where the LED is connected. i.e. bit-0 of PortB
  Include "Amicus18_ADC.inc" ' Load the Amicus18 ADC macros into the program
' Open the ADC:
'               Fosc set for Fosc / 32
'               Right justified for 10-bit operation
'               Tad value of 2
'               Vref+ at Vcc : Vref- at Gnd
'               Make AN0 an analogue input
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_2_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                     ' Create an infinite loop
    LDR_Value = ReadADC(ADC_CH0)' Read the ADC value from AN0
    If LDR_Value <= 400 Then     ' Is the ADC value less than 400. i.e. getting lighter
      High LED                   ' Yes. So illuminate the LED
    Else                         ' Otherwise...
      Low LED                    ' Extinguish the LED
    EndIf
    DelayUS 30                   ' Allow the ADC to recover
  Wend                           ' Do it forever
```

The code is essentially the same as cockroach mode, except the LED illuminates when the ADC value is less that 400, instead of greater than 400.

35

Temperature Sensor

One of the simplest, and least expensive, temperature sensors is a thermistor. This is a special type of resistor that alters it's resistance based upon it's temperature. There are generally two types of thermistor; an NTC type (Negative Temparature Coefficient), whose resistance drops as the temparature increases, and a PTC type (Positive Temparature Coefficient), whose resistance increases as the temparature increases. For this demonstration, we'll use an NTC thermistor.

Just like there fixed resistance cousins, thermistors come in different packages and resistance-per-temperature values. These range anywhere from a few hundred Ohms to tens of thousands of Ohms.
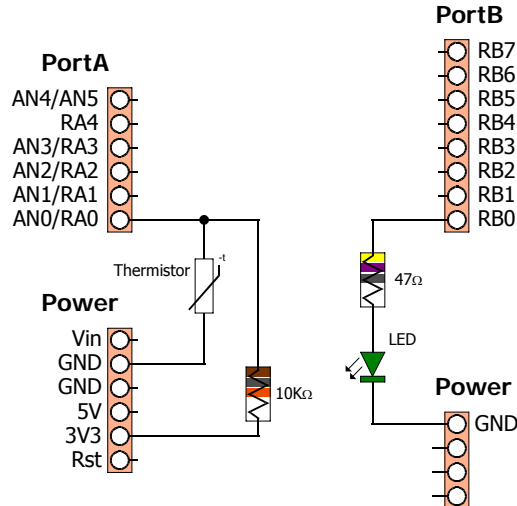
The device used in this demonstration is a bead thermistor with a resistance of 10KΩ at a temperature of 25° centigrade, but any thermistor will do with a few program code changes.

A thermistor layout is shown below:

06-10-2009
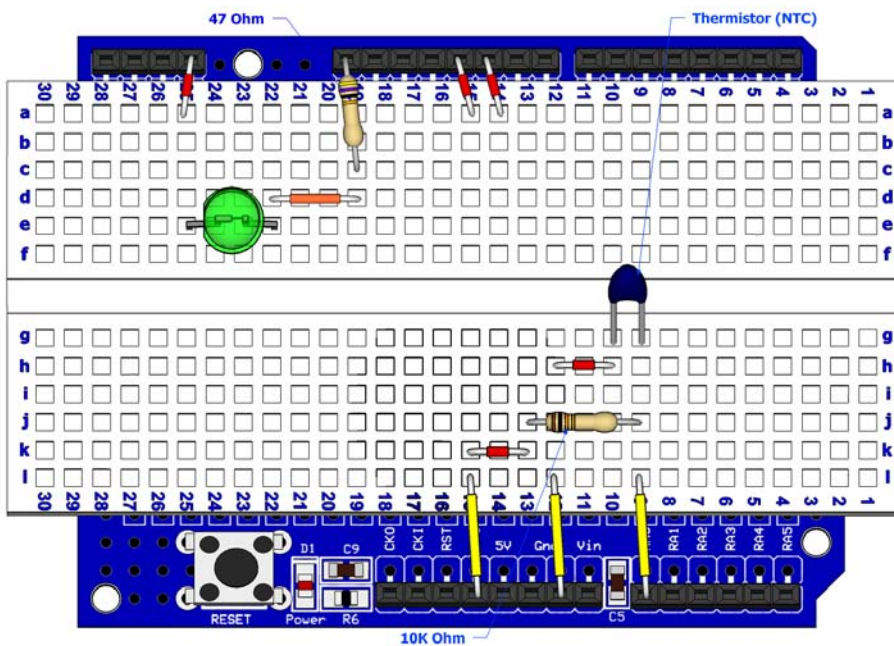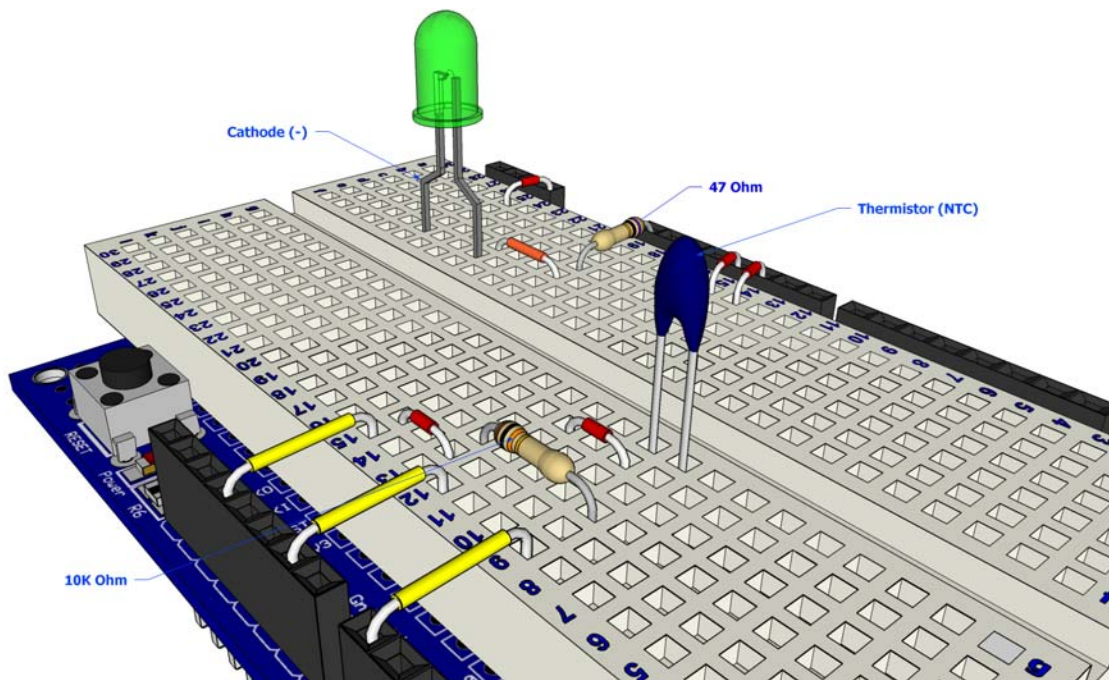
The circuit for the temperature layout is shown below:

A program to display the the values produced from the thermistor on the serial terminal is shown below:

```
' Display the value of an NTC thermistor on the serial terminal
' The thermistor is connected To AN0 (Channel 0 of the ADC)
  Include "Amicus18.inc"      ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_ADC.inc" ' Load the Amicus18 ADC macros into the program
  Dim ThermistorIn As Word     ' Create a variable to hold the 10-bit ADC result
' Open the ADC:
'             Fosc set for Fosc/32
'             Right justified for 10-bit operation
'             Tad value of 0
'             Vref+ at Vcc : Vref- at Gnd
'             Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_0_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
  While 1 = 1                                  ' Create an infinite loop
    ThermistorIn = ReadADC(ADC_CH0)            ' Read the ADC on AN0
    HRSOut "Thermistor = ", Dec ThermistorIn, 13 ' Display the ADC value
    DelayMS 500                                ' Delay for half a second
  Wend                                         ' Do it forever
```

Once the program has been loaded into the Amicus18 board, open the serial terminal and connect to the Amicus18's com port:
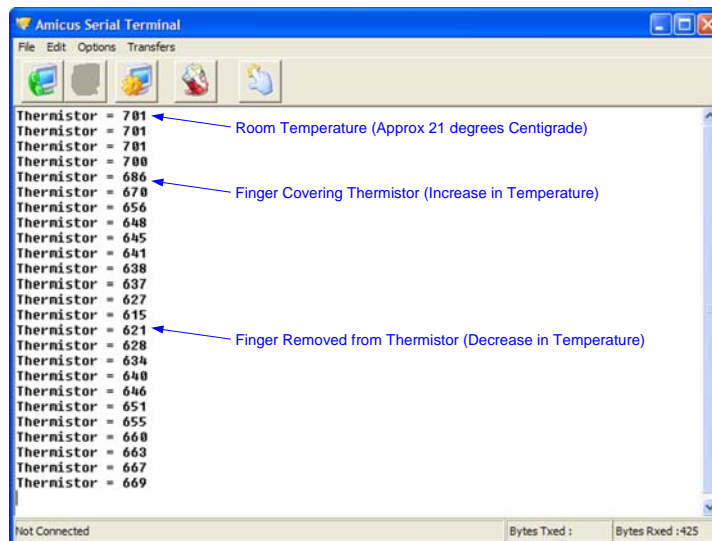
The display shows the decrease in voltage with the increase in temperature when a finger covers the thermistor, and is then removed. As can be seen, a thermistor is quite sensitive.

06-10-2009

Thermostat (increase in temperature)

We can use the information we have to trigger an external device, in this case an LED, when the thermistor reaches a pre-determined value. We know that room temperature give an ADC value of approx 701, and any value lower than this is an increase in temperature, and a lower value is a decrease in temperature, so even without knowing the actual temperature we can write some code:

```
' Illuminate an LED when the temperature increases
' Also display the ADC value of the thermistor connected to AN0
'
  Include "Amicus18.inc"      ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_ADC.inc" ' Load the Amicus18 ADC macros into the program

  Dim ThermistorIn As Word    ' Create a variable to hold the 10-bit ADC result
  Symbol LED = PORTB.0        ' Alias the name LED to pin RB0
'
' Open the ADC:
'              Fosc set for Fosc/32
'              Right justified for 10-bit operation
'              Tad value of 0
'              Vref+ at Vcc : Vref- at Gnd
'              Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_0_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                                ' Create an infinite loop
    ThermistorIn = ReadADC(ADC_CH0)     ' Read the value from the thermistor
    HRSOut "Therm = ", Dec ThermistorIn, 13 ' Display the ADC value on the terminal
    If ThermistorIn < 600 Then               ' Has there been an increase in temperature?
      High LED                               ' Yes. So illuminate the LED
    Else                                     ' Otherwise...
      Low LED                                ' Extinguish the LED
    EndIf
  Wend                                       ' Do it forever
```

Once the program is compiled and loaded into the Amicus18 board using the toolbar *Compile and Program* or pressing **F10**, placing a finger over the thermistor, thus increasing the temperature, will illuminate the LED. To adjust the threshold of the temperature trigger, alter the value within the code line: "**If** ThermistorIn < 600 **Then**". A lower value will illuminate the LED at higher temparatures.

Thermostat (decrease in temperature)

In order to illuminate the LED at lower temperatures, use the program below:

```
' Illuminate an LED when the temperature decreases
' Also display the ADC value of the thermistor connected to AN0
'
  Include "Amicus18.inc"      ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_ADC.inc" ' Load the Amicus18 ADC macros into the program

  Dim ThermistorIn As Word    ' Create a variable to hold the 10-bit ADC result
  Symbol LED = PORTB.0        ' Alias the name LED to pin RB0
'
' Open the ADC:
'             Fosc set for Fosc/32
'             Right justified for 10-bit operation
'             Tad value of 0
'             Vref+ at Vcc : Vref- at Gnd
'             Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_0_TAD, ADC_REF_VDD_VSS, ADC_1ANA)

  While 1 = 1                                ' Create an infinite loop
    ThermistorIn = ReadADC(ADC_CH0)    ' Read the value from the thermistor
    HRSOut "Therm = ", Dec ThermistorIn, 13 ' Display the ADC value on the terminal
    If ThermistorIn >= 750 Then              ' Has there been a decrease in temperature?
      High LED                               ' Yes. So illuminate the LED
    Else                                     ' Otherwise...
      Low LED                                ' Extinguish the LED
    EndIf
  Wend                                       ' Do it forever
```

Once the program is compiled and loaded into the Amicus18 board using the toolbar *Compile and Program* or pressing ***F10***, blowing over the thermistor, thus decreasing the temperature, will illuminate the LED. To adjust the threshold of the temperature trigger, alter the value within the code line: "`If ThermistorIn >= 750 Then`". A higher value will illuminate the LED at lower temparatures.
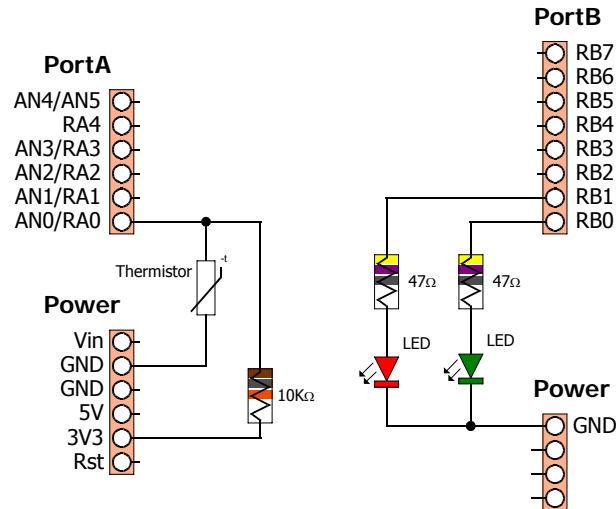
Thermostat (increase and decrease of temperature)

The layout and code below allows the demonstration of high, normal, and low temperature changes. Both LEDs will be extinguished when the temperature is normal, the Red LED will illuminate when the temperature rises above a pre-determined value, and the Green LED will illuminate when the temperature decreases beyond a pre-determined level:
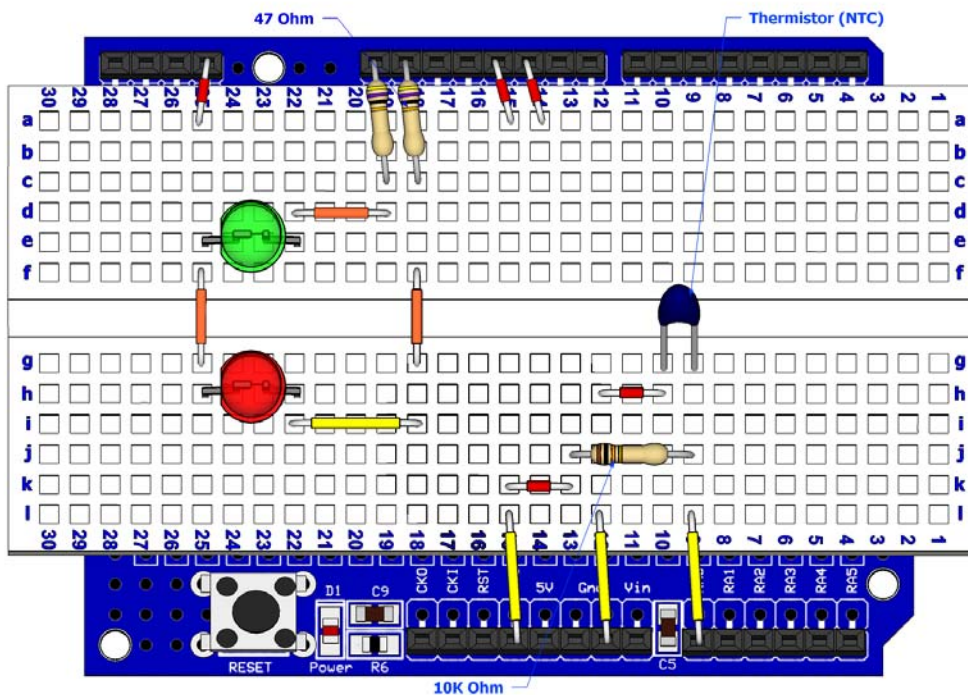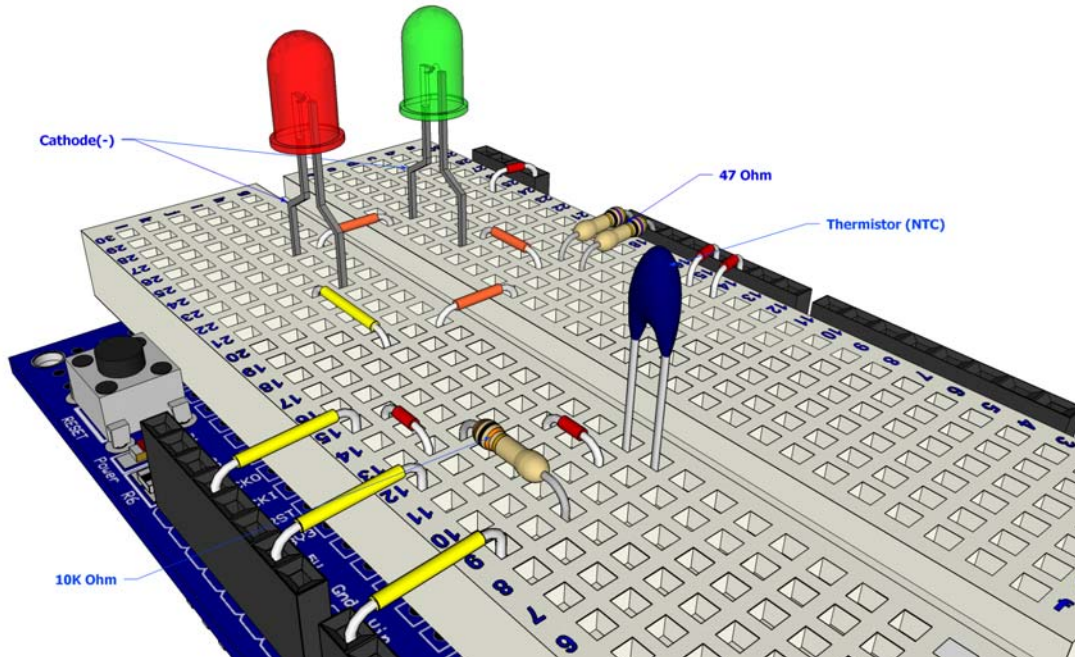
Amicus18 Companion Shield for Proton

The circuit for the thermistat layout is shown below:

The code for the two LED thermostat is shown below:

```
' Illuminate a Red LED when the temperature increases
' Illuminate a Green LED when the temperature decreases
' Also display the ADC value of the thermistor connected to AN0
'
  Include "Amicus18.inc"     ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_ADC.inc" ' Load the Amicus18 ADC macros into the program

  Dim ThermistorIn As Word   ' Create a variable to hold the 10-bit ADC result
  Symbol GreenLED = PORTB.0  ' Alias the name GreenLED to pin RB0
  Symbol RedLED = PORTB.1    ' Alias the name RedLED to pin RB1
'
' Open the ADC:
'            Fosc set for Fosc/32
'            Right justified for 10-bit operation
'            Tad value of 0
'            Vref+ at Vcc : Vref- at Gnd
'            Make AN0 an analogue input
'
  OpenADC(ADC_FOSC_32 & ADC_RIGHT_JUST & ADC_0_TAD, ADC_REF_VDD_VSS, ADC_1ANA)
  While 1 = 1                       ' Create an infinite loop
    ThermistorIn = ReadADC(ADC_CH0)' Read the value from the thermistor
    HRSOut "Therm = ", Dec ThermistorIn, 13 ' Display the ADC value on the terminal
    If ThermistorIn < 600 Then      ' Has there been an increase in temperature ?
      High RedLED                   ' Yes. So illuminate the Red LED
      Low GreenLED                  ' Extinguish the Green LED
    ElseIf ThermistorIn > 750 Then  ' Has there been a decrease in temperature?
      Low RedLED                    ' Yes. So Extinguish the Red LED
      High GreenLED                 ' Illuminate the Green LED
    Else                            ' Otherwise...
      Low GreenLED                  ' Extinguish the Green LED
      Low RedLED                    ' Extinguish the Red LED
    EndIf
  Wend                              ' Do it forever
```

Digital Meets Analogue

Sometimes the microcontroller needs to interface back to the real world with an analogue result. This is termed Digital to Analogue Conversion, or DAC. This can be performed several ways; by using a dedicated DAC peripheral device, by using a digital resistor device, or by using Pulse Width Modulation (PWM). PWM is the method that is built into the Amicus18's microcontroller, and requires no specialised devices to be used, so we'll discuss this method here.

Pulse Width Modulation (PWM)

Pulse Width Modulation fakes a voltage by producing a series of pulses at regular intervals, and varying the width of the pulses. The resulting average voltage is the result of the pulse widths. The Amicus18's microcontroller can produce a high voltage of 3.3 Volts and low of 0 Volts.

In the illustration below, the pin is pulsed high for the same length of time as it is pulsed low. The time the pin is high (called the pulsewidth) is about half the total time it takes to go from low to high to low again. This ratio is called the duty cycle. When the duty cycle is 50%, the average voltage is about half the total voltage. i.e. 1.6 Volts.
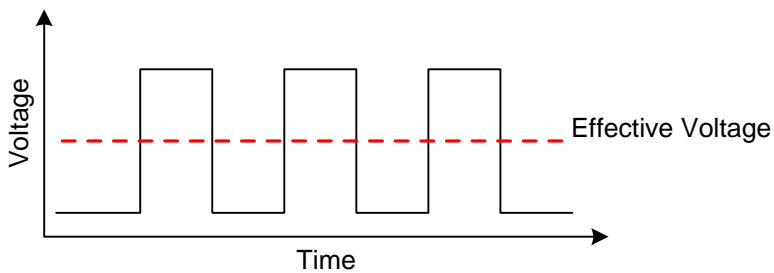
If the duty cycle is made less than 50% by pulsing on for a shorter amount of time, a lower effective voltage is produced:
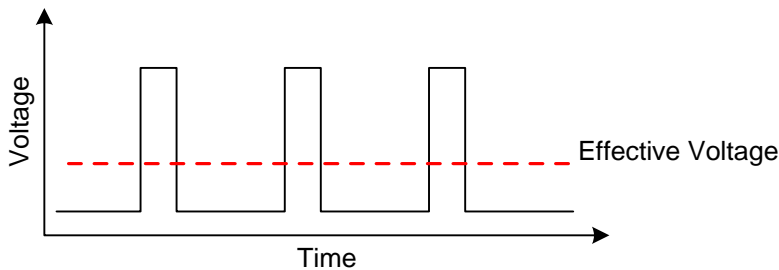
If the duty cycle is made greater than 50% by pulsing on for a longer amount of time, a higher effective voltage is produced:
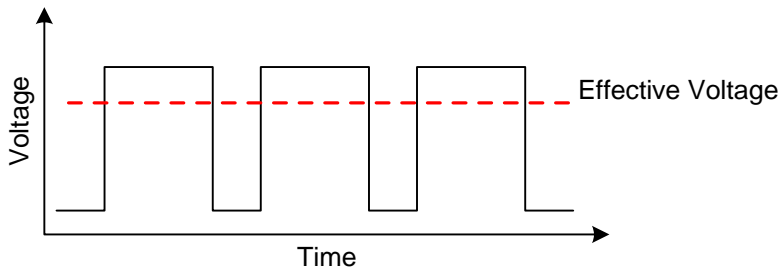
In order to create a constant voltage instead of a series of pulses, we need a simple RC low pass filter. As it's name suggests this consists of a Resistor and a Capacitor.

A filter is a circuit that allows voltage changes of only a certain frequency range to pass. For example, a low-pass filter would block frequencies above a certain range. This means that if the voltage is changing more than a certain number of times per second, these changes would not make it past the filter, and only an average voltage would be seen.

42

There are calculations for the values of the resistor and capacitor used, but we won't go into that here, but a search for *RC filter* on the internet will produce a huge amount of information. Here's two of them that are valid at the time of writing:

http://www.cvs1.uklinux.net/cgi-bin/calculators/time_const.cgi
http://www.sengpielaudio.com/calculator-period.htm

Channel 1 PWM

In this discussion, we'll be using the compiler's **WriteAnalog** macros, which produce either an 8-bit (0 to 255) or 10-bit (0 to 1023) output. It's important for further RC filter calculations to remember that the 10-bit PWM macro's operate at a frequency of 62.5KHz (62,500 Hertz), and the 8-bit PWM macros operate at a frequency of 125KHz (125,000 Hertz), but only if the Amicus18 board is using it's default oscillator speed of 64MHz. If the crystal is replaced with another value type, these frequencies will change. See *section 16* in the PIC18F25K20 microcontroller's data sheet for further information concerning the PWM peripherals.

From the paragraph above, we know that if we use the 10-bit PWM macros, we will be operating at a frequency of 62.5KHz. This relates to a duty cycle of 0.06 milliseconds (ms), or 16 microseconds (us).

If we choose a value of 47 Ohms for our resistor so that we don't loose too much current, we need a capacitance value of 340.425nF (0.34uF). There is no common capacitor of that value so we'll choose a close value, for example 330nF (0.33uF).

The circuit for a suitable RC low pass filter is shown below:
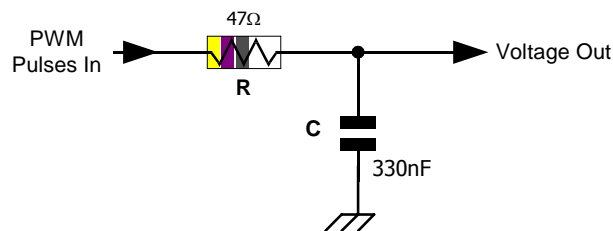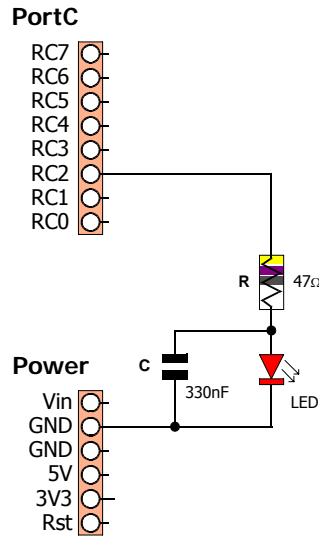

The Amicus18's microcontroller has two PWM peripherals; PWM1 from PortC pin RC2, and PWM2 from PortC pin RC1. Each pin can produce a differing duty cycle (average voltage), but each share the same frequency.

A demo layout for channel 1 of the PWM is shown below:

43
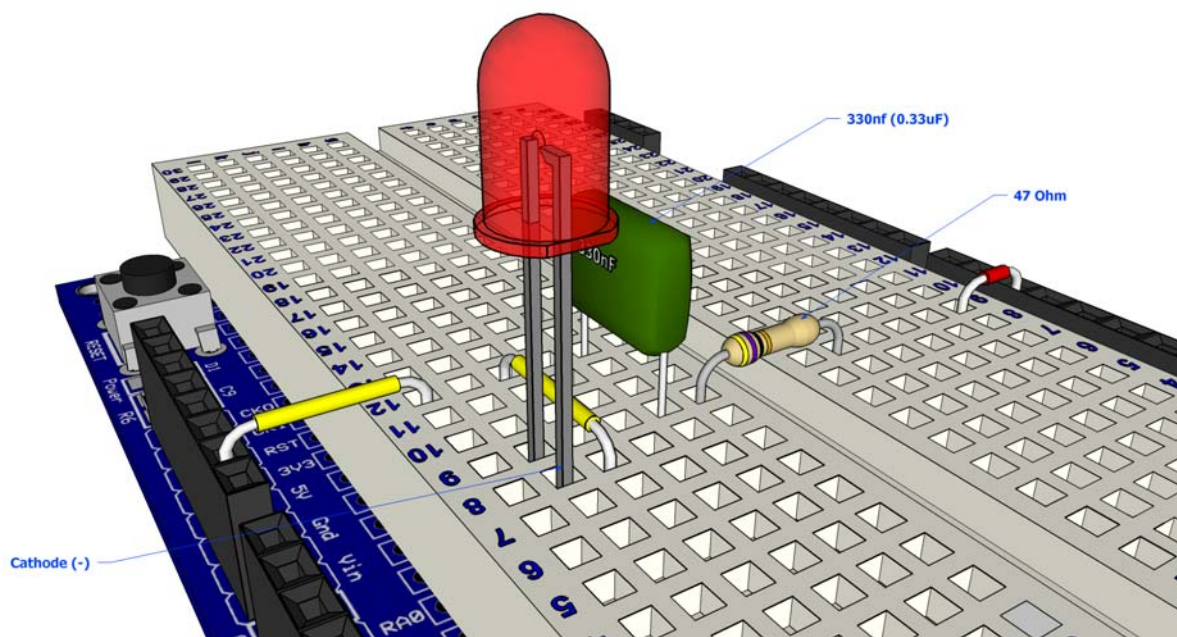
Amicus18 Companion Shield for Proton

The circuit for the PWM1 layout is shown below:

The PWM peripherals operate in the background, which means that once a PWM duty cycle is set, it does not block any other instructions from occurring.

Type in the following code and program it into the Amicus18 board by clicking on the toolbar *Compile and Program*, or pressing **F10**:

```
Include "Amicus18_Hpwm10.inc"

WriteAnalog1(512)
```

The LED will now be glowing, but not at full brightness. What's happening is that channel 1 of the PWM has been instructed to set the duty cycle to 50%, which is half the full range of 1023, which is 512. Try different values within the braces of the WriteAnalog1 command and see what it does to the LED's brightness.

A more sophisticated program is shown below that will cycle the LED to full brightness then back to off repeatedly:

```
' Amicus18 10-bit Hardware PWM (Pulse Width Modulation) Demo Program
' An LED attached to Bit-2 of Portc (RC2) will increase illumination, then dim
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_Hpwm10.inc"' Load the Amicus18 10-bit PWM macros into program

  Dim wDutyCycle As Word        ' Create a variable to hold the Duty Cycle
  OpenAnalog1()                 ' Enable and cofigure the CCP1 peripheral
  While 1 = 1                   ' Create an infinite loop
    '
    ' Increase LED illumination
    '
    For wDutyCycle = 0 To 1023  ' Cycle the full range of 10-bits. i.e. 0 to 1023
      WriteAnalog1(wDutyCycle)  ' PWM on CCP1 (Bit-2 of PortC)
    Next
    '
    ' Decrease LED illumination
    '
    For wDutyCycle = 1023 To 0 Step -1 ' Cycle the full range of 10-bits in reverse
      WriteAnalog1(wDutyCycle)        ' PWM on CCP1 (Bit-2 of PortC)
    Next
  Wend                                ' Do it forever
```

06-10-2009

Amicus18 Companion Shield for Proton

Channel 2 PWM

As has been mentioned, the Amicus18 has two hardware PWM channels, each can work independently of each other when adjusting the duty cycle, but share a common operating frequency and resolution. This is because they both operate from the microcontroller's Timer 2 module.

Operating the second channel of the PWM peripheral uses exactly the same procedure as operating channel 1, but uses a different pin of PortC (RC1).

A demo layout for channel 2 of the PWM is shown below:
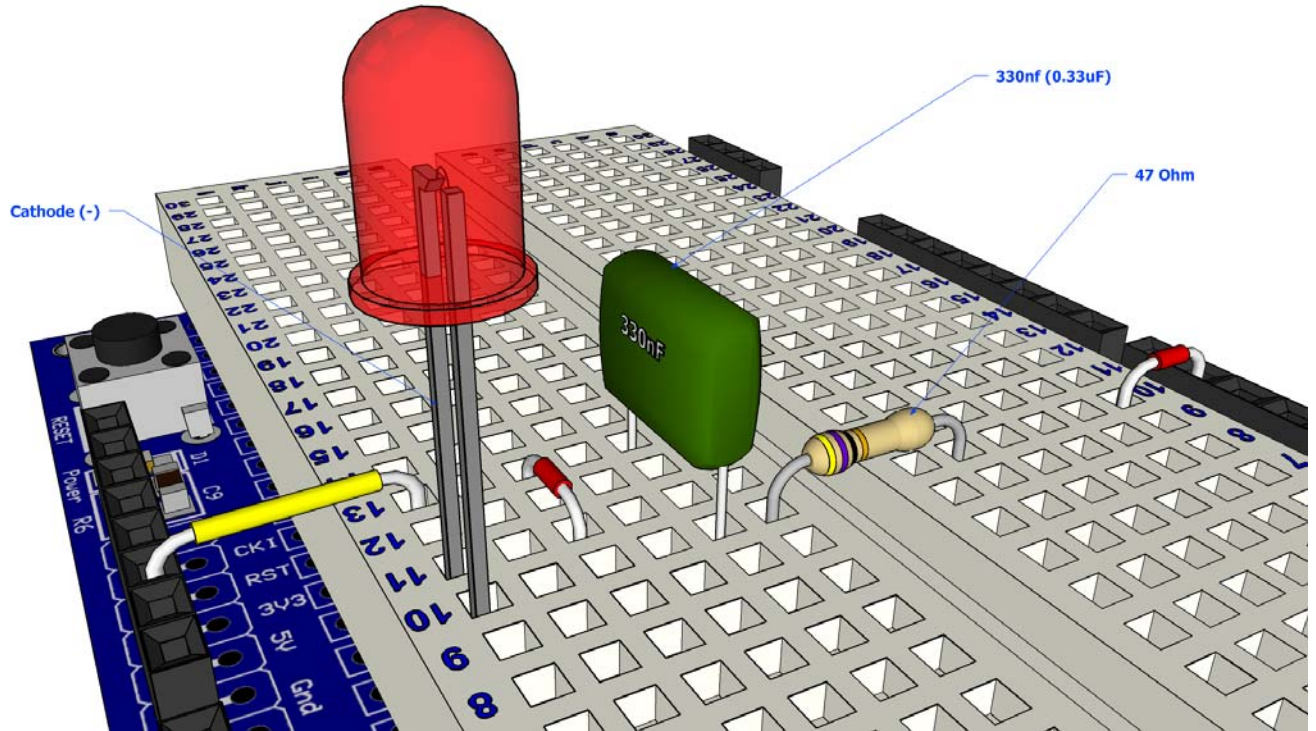
The circuit for the above layout is shown below:
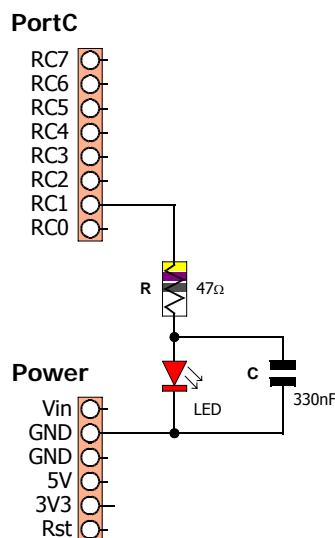
The PWM peripherals operate in the background, which means that once a PWM duty cycle is set, it does not block any other instructions from occurring.

Amicus18 Companion Shield for Proton

Type in the following code and program it into the Amicus18 board by clicking on the toolbar *Compile and Program*, or pressing **F10**:

```
Include "Amicus18_Hpwm10.inc"

WriteAnalog2(512)
```

The LED will now be glowing, but not at full brightness. What's happening is that channel 2 of the PWM has been instructed to set the duty cycle to 50%, which is half the full range of 1023, which is 512. Try different values within the braces of the WriteAnalog2 command and see what it does to the LED's brightness.

A more sophisticated program is shown below that will cycle the LED to full brightness then back to off repeatedly:

```
' Amicus18 10-bit Hardware PWM (Pulse Width Modulation) Demo Program
' An LED attached to Bit-1 of PortC (RC1) will increase illumination, then dim
'
  Include "Amicus18.inc"        ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_Hpwm10.inc"' Load the Amicus18 10-bit PWM macros into program

  Dim wDutyCycle As Word        ' Create a variable to hold the Duty Cycle
  OpenAnalog2()                 ' Enable and cofigure the CCP2 peripheral
  While 1 = 1                   ' Create an infinite loop
    '
    ' Increase LED illumination
    '
    For wDutyCycle = 0 To 1023  ' Cycle the full range of 10-bits. i.e. 0 to 1023
      WriteAnalog2(wDutyCycle)  ' PWM on CCP2 (Bit-1 of PortC)
    Next
    '
    ' Decrease LED illumination
    '
    For wDutyCycle = 1023 To 0 Step -1 ' Cycle the full range of 10-bits in reverse
      WriteAnalog2(wDutyCycle)         ' PWM on CCP2 (Bit-1 of PortC)
    Next
  Wend                                 ' Do it forever
```

46

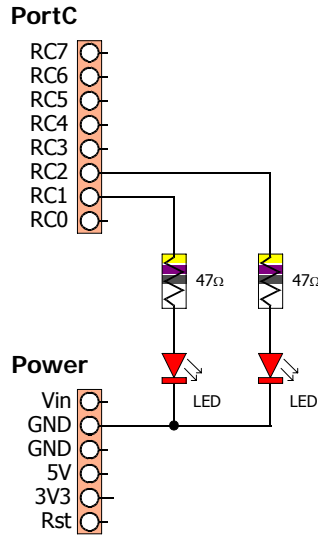## Two channels of PWM simultaneously (Pulsing Light)

The layout below demonstrates both PWM channels operating simultaneously:
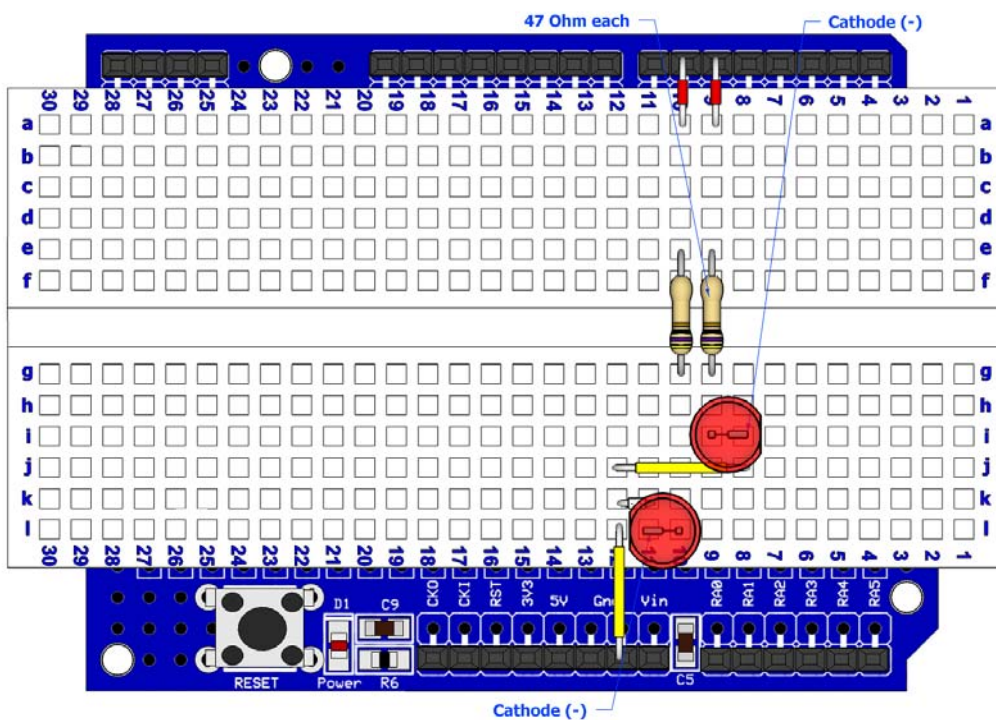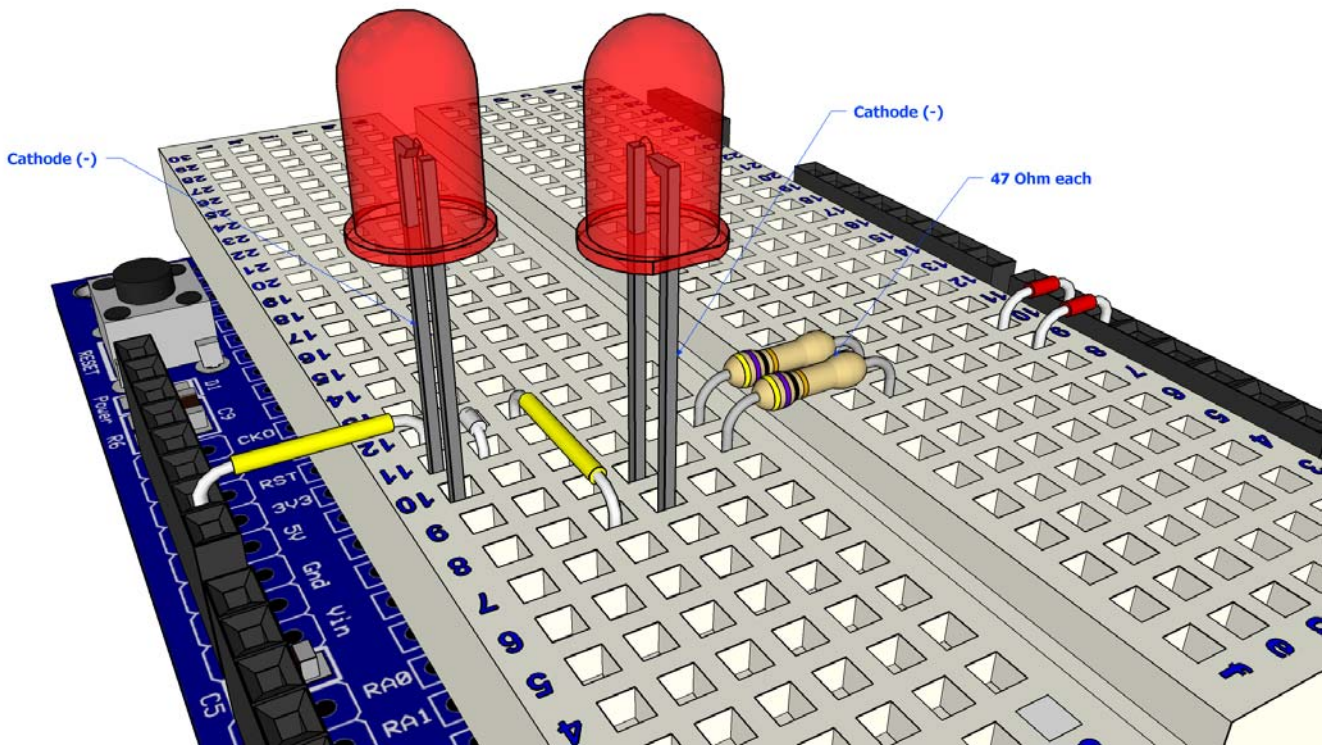
Amicus18 Companion Shield for Proton

The circuit for the 2 PWMs layout is shown below:

PortC

```
RC7 ◯
RC6 ◯
RC5 ◯
RC4 ◯
RC3 ◯
RC2 ◯
RC1 ◯
RC0 ◯
```

47Ω   47Ω

Power

```
Vin ◯
GND ◯
GND ◯
 5V ◯
3V3 ◯
Rst ◯
```

LED   LED

The capacitors normally associated with PWM output have been dispensed with because the operating frequency of the PWM channels is so high (62.5KHz) that no noticeable flicker from the pulses will be observed on the LEDs.

The code to produce the pulsing of the LEDs is shown below:

```
' Pulse both LEDs, one decreases while the other increases brightness
'
  Include "Amicus18.inc"          ' Configure the compiler to use the Amicus18 board
  Include "Amicus18_Hpwm10.inc"   ' Load the Amicus18 10-bit PWM macros into program

  Dim wDutyCycle As Word          ' Holds the duty cycle of the PWM pulses
  OpenAnalog1()                   ' Enable and cofigure the CCP1 peripheral
  OpenAnalog2()                   ' Enable and cofigure the CCP2 peripheral
  While 1 = 1                     ' Create an infinite loop
  '
  ' Increase LED1 illumation, while decreasing LED2 illumination
  '
    For wDutyCycle = 0 To 1023            ' Cycle the full range of 10-bits
      WriteAnalog1(wDutyCycle)            ' PWM on CCP1 (Bit-2 of PortC) (0 to 1023)
      WriteAnalog2(1023 - wDutyCycle)     ' PWM on CCP2 (Bit-1 of PortC) (1023 to 0)
      DelayMS 5                           ' A small delay between duty cycle changes
    Next                                  ' Close the loop
    DelayMS 5
    '
    ' Decrease LED1 illumation, while increasing LED2 illumination
    '
    For wDutyCycle = 1023 To 0 Step -1    ' Cycle the full 10-bit range (reversed)
      WriteAnalog1(wDutyCycle)            ' PWM on CCP1 (Bit-2 of PortC) (1023 to 0)
      WriteAnalog2(1023 - wDutyCycle)     ' PWM on CCP2 (Bit-1 of PortC) (0 to 1023)
      DelayMS 5                           ' A small delay between duty cycle changes
    Next                                  ' Close the loop
  Wend                                    ' Do it forever
```