Atmel Startup 4: Blinky Two – Switches, Pull-Up Resistors and Bit Ops

M. A. Parker, Angstrom Logic LLC, Copyright 2015-07-28

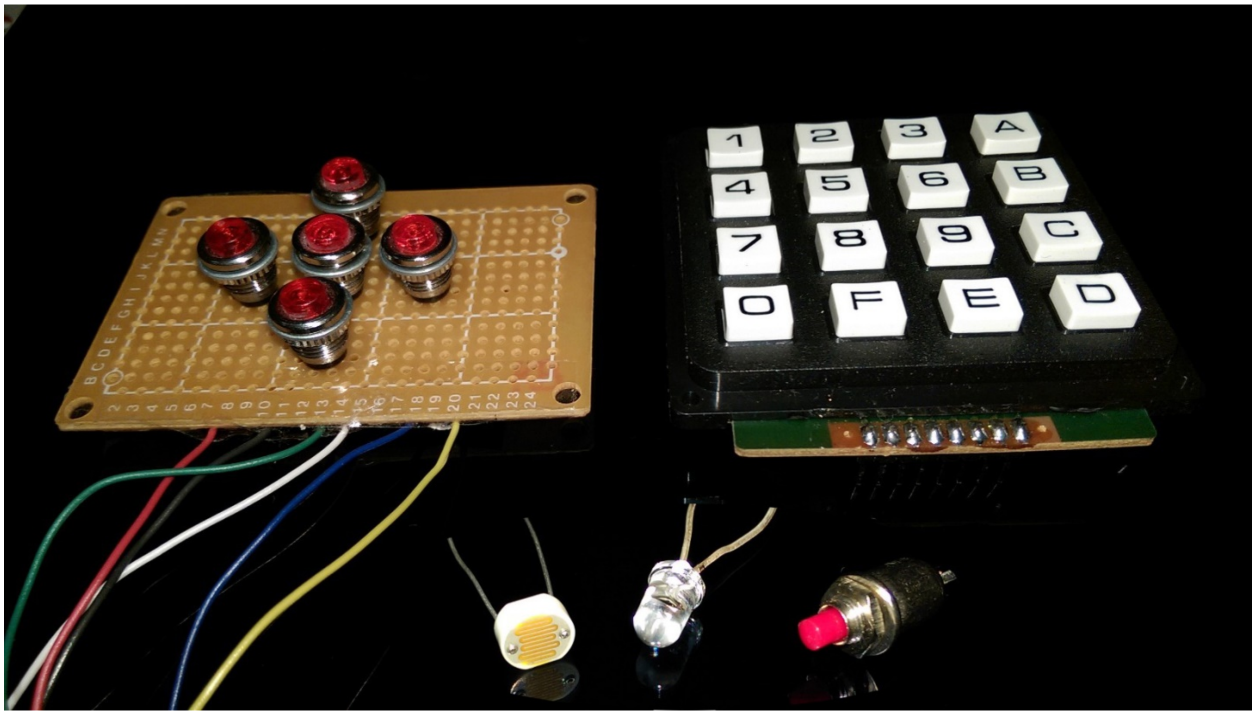

Figure 1: Example switches that can use the pull-up resisters embedded in the MCU, Front: Photo-resistor, photo-transistor, push button; Rear: example key pads.

Blinky Two demonstrates how the MCU can easily read the state of external switches, which includes push buttons and photo-switches, while minimizing the parts count using its embedded pull-up resistors. The discussion focuses on IO ports with mixed input/output, and briefly discusses bitwise operations and masking. Blinky Two uses the circuit platform from the previous two Startups and the cable adapter constructed in the first Startup [0]; however, brief construction details are provided for both. This Instructable is the fourth in a series of 'Startup' Instructables [0] focusing on 'getting started' with an Atmel Microcontroller MCU [1]. Working with the individual MCU offers significantly lower cost for each project with much better control over the size, function and power requirements compared with the MCU-on-a-board systems such as Arduino [2] and the Rhaspberry Pi [3]. The next Startup Instructable will convert the Blinky circuit and software to a system suitable for the Lifeline that, although easy, provides means to correct clock settings for an MCU. The first Startups installed the Atmel Studio and Programmer, constructed the adapter cable (see also parts list) and the basic test/Blinky platform, and discussed PORTs, PINs, DDRs, and LEDs. It should be pointed out that the Lifeline does not correct non-clock fuses - a high voltage programmer (12V) would be required as discussed in Startup #5.

We continue with the programming and coding [4] while providing a brief summary of required hardware from previous Startups. While this Instructable explains each coding statement, the reader might still want to consult some of the many excellent C/C++ references [5-11].

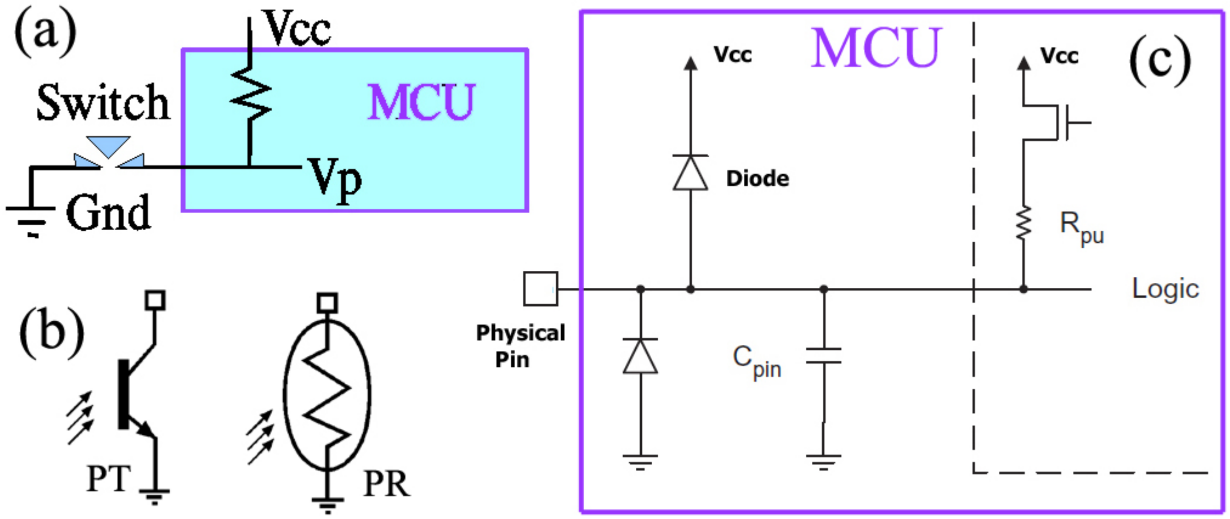

Figure 2: **(a)** Concept of external switch with internal pull-up resistor. **(b)** Symbol of npn photo-transistor PT and photo-resistor PR – top terminal connects to the MCU pin similar to the switch. **(c)** The internal components for the MCU IO pin [1].

Step 1: The Pull-Up Resistor, IO Pin Circuit, and the Switches

The pull-up resistor [12,13] as part of the MCU internal circuitry can be programmatically controlled for use with various devices including switches, open collector logic devices, transistors, and sensors such as NPN phototransistors. Additionally, it can provide a pre-defined state for an input pin when it doesn't have anything connected to it. Such a predefined state prevents random flipping of the input state during operation. As stated in the electrical specs for the MCU [1], the general IO pin resistor has a value of 20k while the reset pin has 30k. Neither value would work well if a long wire has been attached to the pin without terminating it at a lower impedance (more like 1k would be better). Interestingly, the pull-up resistor provides sufficient current to weakly drive LEDs as well as provides bias current for discrete stages since the 20k resistor can provide up to 2.5mA.

For the example shown in Figure 2a, the physical MCU pin attaches to an external switch and the internal resistor. When the switch is open as shown, the resistor pulls the pin voltage Vp up to Vcc (that is, Vp=Vcc). In such a case, the MCU converts the pin voltage Vp into a logic value of 1. Pressing the switch brings the pin voltage to zero (i.e., Vp=0).

The push button switch can be replaced by a sensor such as a phototransistor (PT in Figure 2b) [14]. Here the box on the upper PT terminal refers to the pin on the MCU shown in Figure 2c. Light entering the phototransistor produces electrons and holes that substitute for the normal base current for transistors. The amplified current passes through the pull-up resistor which can reduce the pin voltage Vp enough for the MCU to register a zero logic state. Likewise, the switch can be replaced with the photoresistor (PR in Figure 2b) [15] to become part of a voltage divider between PR and the pullup. Light absorbed in the PR semiconductor creates holes and electrons which effectively increases the conductivity of the material and thereby lowers its resistance. Consequently, Vp then decreases sufficiently for the MCU to register a logic zero. One can expect the photoresistor resistance to vary from about 100k Ohms to 800 Ohms under the lighting of near-dark to bright light, respectively. Refer to the spec sheet for exact values. It should be noted that the sensors such as phototransistors and photoresistors would usually be connected to the Analog to Digital Converter (ADC) in an MCU so that measurements for continuous variations can be made rather than the on-off nature of the digital inputs.

The MCU internal hardware [1] for the IO pin appears in Figure 2c. The program controls the transistor, which can engage/disengage the pull-up resistor, by using the DDR to set the pin as an input, and writing to the pin a logic 1 to engage the pull-up and logic 0 to disengage. As a side note, notice that the pin has the equivalent of two diodes which can function to protect the pin from slight over-voltages and negative voltages. For example, in the case when the pin voltage Vp is larger than Vcc then the top diode will be forward biased and it will tend to prevent the over voltage from exceeding about 0.6 volts. Similar comments apply for negative Vp. However, these diodes can only handle about 1mA – not much protection!

Step 2: Controlling the Pull-Up Resistor

So, how can the pull-up resistor [12,13] be programmatically controlled?

Recall from Start-up #3, the DDR, PORT, and PIN statements in Atmel Studio C/C++ control the function and state of the MCU ports. For example, the partial statement 'DDRB=' will set the 8bit data direction register for port B with 0s and 1s which the MCU then interprets to set the corresponding physical port B pins to input and output, respectively. The '=PINB' partial statement (such as in x= PINB), for example, reads the input register for the physical port B pins. The 'PORTB=' partial statement will write 8 bits to the port B output register which the MCU interprets to set the corresponding physical port B pins – with one proviso concerning the Pull-Up Resistors. And this starts our discussion for the present step. The present step uses the port B as an example but other available ports on an Atmel AVR MCU (such as A, B, …) behave similarly. Recall, the ATTiny2313A uses pins 12 through 19 for portB denoted by B0 through B7, respectively [1] as shown in Table 1.

We consider an example that shows the use of mixed input and output for portB that also activates a pull-up resistor. For the example, suppose we only need outputs on physical pins 13 and 14 (i.e., B1 and B2, see Table 1) but require inputs on the rest of the pins for Port B. Further suppose that pin 18 (i.e., B6) should have a pull-up resistor such as for a switch similar to that demonstrated in the previous step.

First, focus on the pull-up resistor. It can be engaged by WRITING a ‘1’ to an INPUT pin. Let that sink in for a while. Writing a ‘0’ to an input pin disables the internal pull-up resistor and sets the input to a high impedance state [1].

Table 1: An example showing pins 12-19 with both inputs and outputs and pull-up PU activated and high impedance HZ.

Physical Pins	19	**18**	17	16	15	**14**	**13**	12
Name	B7	B6	B5	B4	B3	B2	B1	B0
DDRB	0	**0**	0	0	0	**1**	**1**	0
Physical PortB	In	**In**	In	In	In	**Out**	**Out**	In
PORTB=0b0100100	0	**1**	0	0	0	**1**	**0**	0
Physical Pin Result	HZ	**PU**	HZ	HZ	HZ	**Vcc**	**0V**	HZ

So for this example, in order to obtain the inputs and outputs shown in the fourth row of Table 1, we need to first set DDRB=0b00000110 so that physical pins 13-14 are both outputs and the rest are inputs. To activate the pull-up (PU) resistor for pin 18, we write a ‘1’ to B6, which is an input, as in the sixth row. Next, we set PORTB=0b0100100 as shown in the sixth row of the table. The last row shows the resulting physical states of the physical pins 12-19. For physical pins 13 and 14, the logical values in the DDRB register have been translated to the voltages on the pins as expected. Pins 12, 15-17 and 19 are inputs and have been set in a high impedance state, as indicated by the HZ, because DDRB set them as inputs and the corresponding bits in the PORTB write has disabled the pull-up resistors. Often (but certainly not always) the program sets DDRB and the Pull-Up resistors at the start of execution and does not reset them. To use the mixed input/output port, one needs a method to programmatically distinguish between the input and output pins as well providing the ability to independently read and write them without affecting the pull-up resistors and port settings. Here is where the masking becomes important.

Step 3: Bitwise Operations

Ports with mixed inputs and outputs, such as in Table 1, often require masking operations that consist of a mask and a bitwise operation [16]. The masking selects out individual bits or a subset of port bits for testing. The bitwise operations make the masking operations possible. In this way, the input and output bits of an IO port can be separated and manipulated.

Consider a few illustrative examples that will be used with Blinky Two. These examples write out and manipulate the bits of an 8bit variable A given by A=rstuvwxy where the letters r, s, t, …, y are bits with values of either 0 or 1. For example, A=10001000 means that r=1 and v=1 and the rest are zero. However, we are working with the general A and have not assigned values to r, s, …, y. In C-like notation, one might write A=‘0brstuvwxy’ where 0b tells the compiler the number is binary; however, this notation won’t work in a program.

1. **Bitwise 'AND' represented by ampersand '&'**

   Consider the 8bit variable A=rstuvwxy. Suppose we wish to check the value of bit #1 (not bit #0). Is x=0 or is x=1? So we define Mask=00000010. Note the 1 value in the bit #1 position which corresponds to the position of the bit in variable A that we wish to check. The word Mask is meant to imply that certain bits will be removed from consideration and others highlighted for consideration. Now, consider the bit-by-bit logical AND operation represent by '&'. Consider

   Test = A & Mask.

   Table 2 shows the bits of A, Mask and the result of a bit-by-bit AND. Notice the far right column of the table represents bit #0 which is also the least significant bit.

   Table 2: Example showing the bit-wise AND between two 8bit variables

Variable A	r	s	t	u	v	w	x	y
Mask	0	0	0	0	0	0	1	0
Test = A & Mask	0	0	0	0	0	0	x	0

   The '&' operator is essentially bit-by-bit multiplication with the following definitions 1&1=1, 0&1=0, 1&0=0, 0&0=0 which can be seen to be equivalent to Boolean Algebra or Aristotelian Logic with 1=True and 0=False. The first two of the products (specifically, 1&1=1, 0&1=0) shows the reason x&1=x in Table 2. This is the property that we need for masking with the '&' since all the bits in the 'Variable' will be mapped to zero except for those where the Mask has a 1. See reference [16] for more information on bitwise operations. We can now ask whether 'Test' is equal to zero. If Test=0 then x must be zero.

2. **Bitwise OR represented by the vertical line '|'**

   Again, consider the variable A=rstuvwxy where r, …, y represent bits with values of 0 or 1. Suppose this time we want to make sure that bits #0 and #4 have the value of 1 without changing any of the others in A. That is, we want to set y=1 and u=1. To do so, consider a mask with all zero bits except for #0 and #4.

   Mask2 = 00010001

   The masking operation in this case will be the bit-by-bit OR operation represented by the vertical line |. The OR has the following effect on bits: 1 | 1 = 1, 0 | 1 = 1, 1 | 0 = 1, 0 | 0 = 0. The OR can be seen to be similar to the OR in the Aristotelian logic system with 1=True and 0=False. Table 3 shows the OR operation for A | Mask2.

   Table 3: Example showing the bit-wise OR between two 8bit numbers

8bit Variable A	r	s	t	u	v	w	x	y
Mask2	0	0	0	1	0	0	0	1
A | Mask2	r	s	t	1	v	w	x	1

The two OR definitions of 1 | 0 = 1, 0 | 0 = 0 (i.e., equivalently x | 0 = x) show that those mask bits in Table 3 which are zero do not change the corresponding bits in the Variable (see bits r-t, v-x) when calculating Variable|Mask2 in the bottom row. But the bitwise OR definitely produces a 1 in the result bit where the Mask has a 1 such as for u and y in the table.

Finally then, if we assign the result of 'A | Mask2' back to A (using the C/C++ assignment operator of '='), that is, A = A | Mask2, then we arrive at the desired result of setting bits 0 and 4 to the value one.

3. **Bitwise NEGATION represent by Twiddle ~**

Again, consider the variable A=rstuvwxy where r, …, y represent bits with values of 0 or 1. Suppose this time we want to make sure that bits #0 and #4 have the value of 0 without changing any of the others in A. That is, we want to set y=0 and u=0. To do so, consider a mask with all zero bits except for #0 and #4.

Mask3 = 00010001

which conveniently happens to be the same as Mask2 in the previous example. It might be easiest to look at Table 4 to see what must happen.

Table 4: Example showing the bit-wise AND between two 8bit numbers

8bit Variable	r	s	t	u	v	w	x	y
Mask3	0	0	0	1	0	0	0	1
~Mask3	1	1	1	0	1	1	1	0
Variable & ~Mask3	r	s	t	0	v	w	x	0

We know from the previous examples, that OR can be used to change a bit to a 1, and that AND can be used to change a bit to a zero. So it appears that we need the AND. As shown in Table 4, Mask3 has a 1 at those locations that the result has a 0. To combine the Mask3 with the AND, we have a brain storm and realize that we just need to flip the bits in Mask3 and then apply the AND.

The logical NOT or NEGATION has the following definition ~0 = 1, ~1 = 0. So now the bit positions in ~Mask3 (see Table 4) with a 0 are the positions of variable A that should be set to zero. Using the results for the first example and the fact that y & 0 = 0 (etc) we arrive at the bottom line of Table 4. If we store the result back in variable A according to

A = A & ~Mask3

then we have completed our task of setting bit #0 and #4 to zero in the orginal number. Here are using the '=' as a C/C++ assignment.

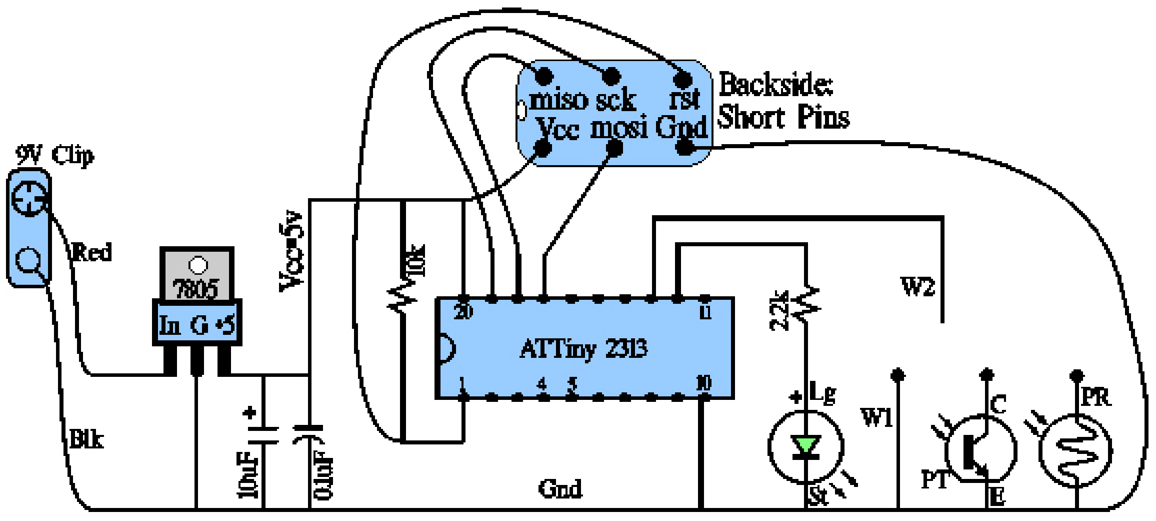

Figure 3: Schematic diagram of the Blinky circuit replicated from Startup #2. Note white dot on the adapter next to the Miso and Vcc pins.

Step 4: The Circuit

By this point, two separate components should have been constructed: the programming cable adapter and a circuit on an experimenter's board. These should have been tested in Startup #2 [0] by setting the fuses for the ATTiny2313A.

We summarize the relevant points from Blinky One in Start-up 2: Be careful that the 9v battery positive terminal connects to the regulator input and NOT the 5v rail at the top of the breadboard. Keep in mind: do not reverse bias the MCU (or else kaput), do not reverse bias the capacitor (or else stinky kaput), do not reverse bias the LED (or else maybe nothing). The capacitors and resistors either have the value printed on the side or else they have codes as shown in References [17,18]. For the LED [0.C], choose Red or Green for use with the 2.2k resistor shown in the schematic. Blue will be ok but you might want to change the resistor to 1.5k to make it brighter. As a point worth remembering, do not connect any powered 5v device to an MCU running on less than 5v since generally an over voltage of 0.5V – 0.7V is enough to smoke the MCU.

The modified circuit for Blinky Two appears in Figure 3 although the phototransistor PT and the photoresistor PR are optional.

1. Wires W1 and W2 as a switch: One end of W2 should be connected to pin 13 (B1) while the other end should not be connected yet – don't let it touch any of the other components though. Wire W1 is not really needed since W2 can be connected to the ground rail when needed.

2. Photo-transistor PT: For the PT [19], the emitter E should be connected to the ground rail and the collector should not be connected yet – but do press it into the experimenter's board. Note the NPN phototransistor PT334-6C recommended in the parts list places C at the flat side of the cylinder and makes E the longer of the two leads [17]. If the PT doesn't work, then you should be able to reverse it without damage (at 5V or less).

3. Photo-resistor PR: One terminal (either terminal since the PR is not polar) should be connected to the ground line while the other should not be connected – but do press the loose end into the board. The various components can be tested just by connecting the wire W2 to one of the devices: wire W1, PT collector, PR terminal.

Step 5: Create the Lifeline solution if not previously created

This Step should be skipped if the Lifeline solution was previously created in Startup 2 or 3. The abbreviated procedure appears below.

1. Connect the programmer to the USB port and start Atmel Studio 6 (AS6).
2. Click the menu sequence Files > New Project. Choose C++Executable.
3. Next to 'name' type 'Atmel Lifeline'. Check the box next to 'Create Directory for Solution'.
4. The Device Selection Dialog pops up. Select 'ATTiny 2313A'. Click OK.
5. Select the Atmel menu sequence: Project > Atmel Lifeline Properites.
Select the left hand Device tab and verify Device = ATTiny2313A
Select the Tool tab and Select your programmer. The ISP Clock of 125 kHz will work – this value must match the one set in Tools > Device Programming. It must be less that ¼ of the clock rate of the MCU to be programmed.
6. Connect the 9V battery to the experiment's board circuit and connect the programmer to the board through the adapter cable previously constructed.
7. Select the Atmel menu sequence: Tools > Device Programming. Verify that the Tool box displays your programmer and the device box displays ATTiny2313A, and the acronym ISP should appear in the 3rd box. Read the Target Volts – it should be within a few percent of 5V. Set the ISP Clock to 125 kHz to match that in Item 5 above.
8. Save and close the solution. Note: The full solution can be saved by clicking the icon with the multiple disks in the tool bar or else use File > Save All.

Step 6: Open the existing Lifeline solution and Set-up

As mentioned in previous Startups, the Atmel Studio AS essentially uses the Microsoft Integrated Development Environment IDE for the Visual Studio VS. The files produced by AS (and VS) are bundled together to form a Solution rather than a program. Open the existing Atmel Lifeline solution as follows. We follow this procedure each time we open an existing solution (except we don't mess with the fuses once they have been set).

1. Connect the programmer to the USB port and attach it to the experimenter's breadboard through the cable adapter constructed in Startup #1 (see also the parts list for brief construction). Connect the battery to the 9v batter clip.

2. Start Atmel Studio. Open Lifeline using the menu sequence:

File > Recent Projects and Solutions > Atmel Lifeline.atsln

3. Select the Atmel menu sequence: Project > Atmel Lifeline Properites.
Select the left hand Device tab and verify Device = ATTiny2313A
Select the Tool tab and verify your programmer. The ISP Clock of 125 kHz will work – this value must match the one set in Tools > Device Programming. It must be less that ¼ of the clock rate of the MCU to be programmed.

4. Select the Atmel menu sequence: Tools > Device Programming. Verify the Tool box names your programmer and the device box names ATTiny2313A, and the 3rd box should show 'ISP'. **Click the APPLY button.** If AS wants to upgrade the programmer firmware then go ahead and do it. If an error occurs then refer back to Startup#1 [0]. Next, read the Target Volts – it should be within a few percent of 5V. Set the ISP Clock to 125 kHz to match that in Item 5 above. Do not close the dialog – proceed to Item 5 below.

5. If the fuses have not been set for the current ATTiny2313A, then verify the following settings by clicking the fuses tab on the left. Refer to Startup #2 for details. Caution Caution Caution: Do not unnecessarily play with the fuses! And definitely don't look cross-eyed at the 'Lock Bits' at this time – they are used to prevent the viewing of flash content – different MCUs have different lock bits.

SELFPRGEN: No Check; **DWEN**: No Check; **EESAVE**: No Check; **SPIEN**: Checked; **WDTON**: No Check;

BODLEVEL: Disabled; **RSTDSBL**: No Check; **CKDIV8**: No Check; **CKOUT**: No Check;

SUT_CKSEL: INTRCOSC_8MHz_14CK_0MS

6. Close the dialog for Tools > Device Programming. Do not close the Atmel Studio (etc.).

Step 7: Blinky Two – Setup and Code

Blinky Two demonstrates the use of the DDR for input and the use of the pull-up resistors. It intentionally uses a complicated input-output arrangement for a single physical port to illustrate concepts for the DDR and pull-up resistors.

1. Check Step 6 above.
2. Make sure the Atmel Lifeline.cpp form appears in the IDE and that any typing/Editing will be entered onto the form (select the tab near the top or click on the file name in Solutions Explorer on the right side).
3. Enter the following statements being very careful of the capitalization, parentheses, braces, and semicolons. Yes, modify the Blinky One program as needed.

```
#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>

int main(void)
{
        DDRB = 0b00000001;            // or 0x01; Sets B0 as output and the rest as input
        PORTB = 0b00000010;           //Sets Pull-UP on B1 and Sets B0=0

        uint8_t Mask = 0b00000010;    //Mask to test when pin B1 is pulled low
        uint8_t MaskL = 0b00000001;   //Mask use to control LED
        uint8_t Test = 1 ;                    //Test result of when pin B1 is low=0 or high=1

  while(1)
  {
        Test = PINB & Mask;
        if(Test == 0)             //Runs when switch activated, W2=ground
        {
                _delay_ms(1000);
                PORTB |= MaskL;           //Illuminates LED; No effect on Pull-Up R
                _delay_ms(1000);
                PORTB &= ~MaskL;  //Extinguishes LED; No effect on Pull-Up R
        }
  }
}
```

4. Go ahead and compile Blinky Two and load it into the MCU as described in Startup #3, Step 5. In brief: Press F5 or click the small triangle on the tool bar with the drop down box for Debug (not the one with the two vertical bars). If there are any errors or warnings, then correct them and try again.
5. At this point, the LED should **NOT** be blinking.

Step 8: Test the Switch and Photo-Sensors

We first test the switch formed by the wires and then the optional sensors although these are not really the main point of this Instructable. Recall that an extra wire was attached to pin #13 of the ATTiny2313A MCU on the Experimenter's board. The three switches will be tested by connecting wire W2 first to W1 then the top terminal of the photo-transistor and then the top terminal of the photo-resistor. Make sure the battery has been connected.

1. Switch

Attach the free end of wire W2 to 0 Volts (i.e., ground) on the experiments board (or equivalently, the free end of wire W1). The LED should start blinking. Remove the wire - the LED should stop blinking. The wire is being used as a switch and the internal pull-up resistor keeps pin 13 at Vcc (logic 1) unless the wire is attached to Gnd (logic 0).

2. Phototransistor PT

Attach the free-end of wire W2 to the free-terminal of the phototransistor (i.e., the collector). In my case, I needed to shine a flashlight on the phototransistor to enable the LED blinking. Room light was not enough to start the blinking. If your phototransistor has a lot of gain, then you might need to cover it and place it in total darkness to stop the blinking. When finished, remove wire W2 from the collector of the phototransistor.

3. Photoresistor PR

Next, attach the loose end of wire W2 to the free end of the photoresistor. Under room lighting, the photoresistor can have sufficiently low resistance that the LED will blink. In such a case, cover the surface of the photoresistor with fingers or black plastic tape to stop the blinking. If room lights don't work then get a flashlight and shine it on the PR to start the blinking.

4. Try NOT EQUAL

Try replacing the program statement of 'if(Test == 0)' with the statement 'if(Test != 0)' and repeat the items 1-3. What happens? The != means NOT EQUAL. It's a good idea to undo any changes when finished in case you want to experiment with the original version some more.

Step 9: Blinky Two – The Meaning of the Statements

The overall idea for the Blinky Two program: First, set B0 as an output and B1 (and the rest) as an input. B0 drives the LED and B1 is an input for the switch. We want the LED to blink, which means B0 must constantly switch states whenever the input B1 is at logic 0 (i.e., pin at ground). For LED blinking, we need to alternately write 0 and 1 to exactly one pin, namely B0 without affecting any other pin. We do this by using bitwise operations in such a manner as to affect only B0. To determine if the input on B1 is logic 0, we use a mask (and a bitwise operation) to single out the B1 bit and test if it is zero, which corresponds to the LED in a blinking mode. Normally, one would not use such a complicated arrangement unless there weren't any other choice. The switch would most likely be placed on another available port separate from the LED.

For some of the discussion below, you might want to print out the Blinky Two coding in order to easily compare the coding with the explanation.

1. The #include <avr/io.h>, #define F_CPU 8000000UL, and #include <util/delay.h> have not changed from Blinky One, nor have the statements DDRB=0b00000001, while(1), and the _delay_ms statements.
2. Notice that DDRB=0b00000001 sets B0 as an output whereas all the other B physical pins are inputs to the MCU. In particular, B1 is an input. B0 drives the LED and B1 senses the switch.
3. PORTB = 0b00000010: Here, since B0 is an output, the right hand 0 in this PORTB assignment sets pin B0 to zero volts (LED 'off'). Next, since B1 is an input, the 1 in the PORTB assignment writes a 1 to an input (B1) which activates the B1 pull-up resistor.
4. uint8_t: Unsigned 8-bit integer which is usually called a 'byte'. This variable type hold numbers consisting of at most 8 bits. This variable can range from 0 to 11111111 which is 255 (or 0 to 0xFF). The uint8_t is a 'defined type' and is not native to C/C++ although those languages do have 'char' and 'unsigned char'. Note that a 'signed' byte such as int8_t would also have a value 0 to 0xFF but the most significant bit (i.e., the z in 0bzxxxxxxx) is 1 for negative and 0 for positive numbers.
5. uint8_t Mask = 0b00000010: Defines an 8bit variable Mask which is given the value of 0b00000010. The mask will be used to eliminate bits 0 and 2-7 from consideration.
6. Uint8_t MaskL = 0b00000001: Defines an 8bit masking variable MaskL having a value of 0b00000001. The L in MaskL serves as a reminder that it applies to the LED. MaskL will be used to remove bits B1 through B7 from consideration and will only affect bit B0.
7. uint8_t Test = 1: An 8 bit variable given the initial value of 0b00000001. In this case, the initial value does not matter since the variable Test will be given a value prior to checking its value. The '=1' part could have been left off; however, the semicolon is still required.
8. Test = PINB & Mask: The PINB statement will read the bits in the register representing the bits B7 through B0. The '&' represents a bit-by-bit logical AND (see Table 2). Only bit #1 in the Test variable (i.e., the bit corresponding to B1) will retain the original PINB value after the Masking (i.e., & Mask) [16]. Consequently as per Table 2, the variable Test will have all zeros except possibly for bit B1 which can be either 0 or 1.
   When the end of the wire is connected to the 0v rail, pin B1 will be zero and so PINB & Mask will produce all zero for all the bits and Test will therefore have the value of 0. The MCU will then

drive the LED 'on'. If the end of the wire is removed, the pull-up resistor will place Vcc on the B1 pin so that the value of PINB & Mask will have the value 0b00000010. The MCU will see a nonzero value for Test and it will disable the LED.

9. if(x): A flow control that executes statements between the two following braces { } when the statement x is true. In this case, the statement x has the form Test == 0. The double equals-sign '==' asks whether or not the variable 'Test' is the same as 'zero' (i.e., 0x00). In C/C++, the logical 'false' is equivalent to the value '0', but the logical 'true' is anything not '0' such as '1' or '5' and so on. For Blinky Two, the variable 'Test' is zero when the switch is closed and that's when we want the LED to blink. So we use the statement 'if (Test == 0)'.
10. PORTB |= MaskL: Generally, the vertical line | means bitwise OR [16]. The statement is a shortcut for

    PORTB = PORTB | MaskL

    The right hand PORTB reads the previous value sent to portB. The left hand PORTB then writes the results of the bitwise OR back to portB. The masking uses MaskL = 0b00000001 with a 1 only in position #0 (corresponding to B0), which controls the LED. The OR will then only affect bit B0 by setting it to Logic 1 (pin 12 will be Vcc) as discussed in conjunction with Table 3. The other bits in the portB write will be the same as they were when first read. So when PORTB is written, none of the Pull-Up resistors will be activated or deactivated! Only the least significant bit (i.e., B0) will be written to 1 which will then place 5v on the physical B0 pin and that will activate the LED.
11. PORTB &= ~MaskL: This statement is shortcut for PORTB = PORTB & ~MaskL where & is the bitwise AND operation (c.f., Table 4). The right hand PORTB reads the values previously written to portB. The left hand PORTB writes the result of the AND back to portB. Given that MaskL = 0b00000001 then ~MaskL = 11111110. The AND will set bit #0 in PORTB to logic 0. The remaining bits from PORTB will not be affected and the original values will be written back to portB. Therefore the connectivity of the pull-up resistors will not be affected. The pin B0, which drives the LED, will be set to zero and the LED will be 'off'.

Step 10: Parts List

The parts should already have been accumulated in previous startups. However, the list is repeated here in brief for convenience.

1. Adapter: The construction of the programmer adapter appears in Startup 1 (see Reference [0]). In brief, the adapter is constructed using a piece of double row male header available from Amazon.com for example. Clip off a piece of the header so as to have 2 rows, 3 columns. Solder 6-inch 24guage plastic-coated solid wires to the short side of the pins. Apply quick dry epoxy to the side of the header with the short pins being sure to cover part of the wires (no more than about 1/4 inch and being careful not to put any epoxy on the longer side of the pins. Label the wires according to Figure 2 in this text.
2. Atmel Programmer: ATATMEL-ICE-BASIC at Mouser.com or Digikey.com for about $55.
3. Atmel Studio AS6 (free): http://www.atmel.com/tools/atmelstudio.aspx

4. ATTiny 2313A: Digikey.com ATTINY2313A-PU-ND at $1.70 (pDIP package, 20pins).
5. Startup #5: 20 pin socket for DIP package (0.3" row spacing, 0.1" pin spacing), low profile: Digikey.com ED3054-5-ND or AE9998-ND for about 30cents. Similar sockets can be found on Amazon.com.
6. LM7805 Regulator: Digikey.com LM7805CT-ND or LM7806ACT-ND for about 70Cents.
7. Electrolytic Capacitor 10uF, 50V: A large variety of capacitors but aluminum type will work ok: Digikey.com: P997-ND (mfr. number: Panasonic ECE-A1HKS100) about 30cents. Actually Amazon has a capacitor kit with both ceramic and electrolytic for $20 called "Joe Knows Electronics 33 Value 645 Piece Capacitor Kit" which looks like a reasonable set.
8. Ceramic Capacitor 0.1uF 50V: Again, a large variety available here. Digikey.com 399-4454-1-ND (Kemet C410C104M5U5TA7200) for about 25cents. See also the kit listed in #7 above.
9. Optional: Two Ceramic Capacitors 22pF (25-100V ok) such as Digikey.com 490-8663-ND at 34 cents.
10. Optional: Purchase a crystal from the HC-49US series or a series with a similar profile. Capacitance loads in the range 18-22pF more or less and less than about 50 Ohm equivalent series resistance ESR will work ok. Amazon has nice packages of crystals as found by searching Amazon for 16MHz Crystal or 8MHz Crystal etc. Here are some from Digikey.com:
    16MHz Crystal from Digikey 300-6034-ND at 54cents each.
    20MHz Crystal from Digikey 300-6042-ND at 54cents each.
    8MHz:
        8MHz Crystal from Digikey X164-ND or X1093-ND at 81cents each (but ESR=80).
        8MHz Crystal from Digikey X100-ND 70cents each (but larger profile)
        Amazon.com: Search 8MHz Crystal – good selection, low profile, good ESR
    10MHz Crystal from Digikey 887-1010-ND at 33 cents each.
11. LED: Amazon has a package of 50 LEDs of various colors for $10 (see Microtivity IL 185). Ebay has many.
12. Resistor: Search Amazon.com for 'resistor kit'. The SparkFun 500 1/4W resistor kit and the E-Projects – 400 Piece, 16 Value resistor kit look OK. Here are the values listed for Digikey:
    1K Digikey.com: CF14JT1K00CT-ND 10cents each
    1.5k Digikey.com: CF14JT1K50CT-ND 10cents each
    2.2k Digikey.com: CF14JT2K20CT-ND 10cents each
    2.7k Digikey.com: CF14JT2K70CT-ND 10cents each
    3.3k Digikey.com: CF14JT3K30CT-ND 10cents each
    4.7k Digikey.com: CF14JT4K70CT-ND 10cents each
    5.6k Digikey.com: CF14JT5K60CT-ND 10cents each
13. Resistor 10k required for the RST and 1k for using Lifeline with 3.3V.
    1k Digikey.com: CF14JT1K00CT-ND 10cents each
    10k Digikey.com: CF14JT10K0CT-ND 10cents each
14. Optional for Startup #4: Photo-transistor. Order from either Digikey.com or Amazon.com
    Digikey: Everlight PT334-6C 38cents each
    Amazon: Search 'Everlight PT334-6C' about $5 for ten.
15. Optional for Startup #4: Photo-resistor from Amazon. Just about any will work ok.
    Amazon.com search for 'photoresistor GM5539'. Cost $5 for ten.
    Digikey PDV-P8001-ND by Advanced Photonics mfg # PDV-P8001 for $2.22
16. 9v Battery clip: Radio Shack or search amazon.com for '9v battery clip' (10 for $2).

Digikey.com: BS6I-ND 60 cents.

17. The plastic housing for the Lifeline is constructed from a battery holder for 4 AA batteries that also has a switch. These come from Radio Shack or Amazon.com. For amazon, search for ‘plastic battery holder 4 AA switch’. Make sure the case holds 4 AA batteries and has a switch. I found a deal of five cases for $8.
18. Experimenter’s breadboard: There are many types and sizes and places to purchase.
    Amazon.com: MB-102 Point Prototype PCB breadboard for $5 includes wires
    Amazon.com: MB-102 830 Solderless Breadboard for $4 with wires. 8-32 screw 1.5” long.
19. Screw 8-32 1.5inches long: check HomeDepot.com or Lowes.com. Any 8-32” screw longer than 2” will be ok so long as it is later cut to a length of 1.5”. You need two 8-32 nuts. Many times the 8-32 screws come with either a large flat head (truss head) or the smaller round heads (round head). Find the round head since these fit better with the spring. The truss head would likely need to be filed down.
20. 24 gauge solid core copper wire. 0ld telephone/intercom wire and intercom wire will work and has many colors. The 24ga has a diameter of approximately 0.51 mm but it’s not critical.
    Amazon.com: search on ‘4 conductor 24 awg cable solid copper’ either 60cents/foot or $20 for 100 feet. Ebay.com has nice 24 ga. solid core wire with multicolors. Search: ‘solid 24 gauge wire kit’.
    Home Depot: 100 ft. 6-conductor Indoor Phone wire for $20. I have not checked this for solid core copper. For soldering the ICs and components, you might want to use 24ga stranded copper wire since it is more flexible and less likely to pull loose from the joint. Amazon.com: search for ‘24 gauge stranded wire’. Nice 6 spool kits (150’ total) will come up for $20.

Step 11: References

[0] The Atmel Startup articles, Instructables.com, 2015, M. A. Parker, Angstrom Logic:

A. Atmel Startup 1: Atmel Studio and Programmer
B. Atmel Startup 2: Microcontroller Circuits and Fuses
C. Atmel Startup 3: Blinky One – PORT, PIN, DDR and LED
D. Atmel Startup 4: Blinky Two – Switches, Pull-Up Resistor, and Bit Ops
E. Atmel Startup 5: Lifeline

As a note, these Instructables made extensive use of the free HTML conversion tool at

http://word2cleanhtml.com/

by cutting and pasting into the tool and then cutting and pasting the results into the Instructables page using the HTML view.

[1] The complete Atmel manual for the ATTiny 2313A: http://www.atmel.com/Images/doc8246.pdf. More information on the ATTiny2313A can be found at http://www.atmel.com/devices/ATTiny2313A.aspx.

[2] Arduino products and learning: http://www.arduino.cc/

[3] Raspberry Pi products and learning: https://www.raspberrypi.org/

[4] Coding vs. Programming http://workfunc.com/differences-between-programmers-and-coders/

[5] The classic masterpiece: B.W. Kernighan, D. M. Ritchie, "The C Programming Language", Prentice, 2nd Ed. (1988). Available on amazon.com for $7.

[6] Good C++ review after reading the Kerninghan-Ritchie book and some background in OOP. "C++ Super Review" written by The Staff of Research and Education Association, published by Research and Education Associated (2000). ISBN: 0878911812. Amazon.com for $8.

[7] A good C++ tutorial site can be found at http://www.cprogramming.com/tutorial.html#c++tutorial
[8] Good overview of Atmel concepts and C but uses the AVR Butterfly:
J. Pardue, "C Programming for Microcontrollers Featuring ATMEL's AVR Butterfly and the free WinAVR Compiler", Published by Smiley Micros (2005). ISBN: 0976682206. Amazon.com
[9] Keep in mind that avrfreaks.net has tutorials and project help
http://www.avrfreaks.net/forum/newbie-start-here?name=PNphpBB2&file=viewtopic&t=70673
http://www.avrfreaks.net/forums/tutorials?name=PNphpBB2&file=viewforum&f=11
[10] SparkFun website has a variety of tutorials ranging from electronics to software:
https://learn.sparkfun.com/tutorials/tags/concepts.
And a blinky here: https://www.newbiehack.com/MicrocontrollerTutorial.aspx.

[11] GCC reference:
http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf
https://gcc.gnu.org/onlinedocs/
https://gcc.gnu.org/onlinedocs/libstdc++/

[12] Pull-up resistor: https://learn.sparkfun.com/tutorials/pull-up-resistors

[13] http://www.avrfreaks.net/forum/avr-input-pin-wpull-resistor-explanation
[14] Phototransistors
http://hades.mech.northwestern.edu/index.php/Photodiodes_and_Phototransistors
http://www.radio-electronics.com/info/data/semicond/phototransistor/photo_transistor.php
http://www.hofoo.com.cn/uploadfiles/phototransistor-ired%20data%20book.pdf
[15] Photoresistors: https://en.wikipedia.org/wiki/Photoresistor
[16] Bitwise operations: http://en.wikipedia.org/wiki/Bitwise_operations_in_C

[17] Capacitor codes: http://www.wikihow.com/Read-a-Capacitor

[18] Resistor color code https://en.wikipedia.org/?title=Electronic_color_code

[19] Spec Sheet PT334-6C: http://www.digikey.com/product-search/en?keywords=everlight%20PT334-6C%20