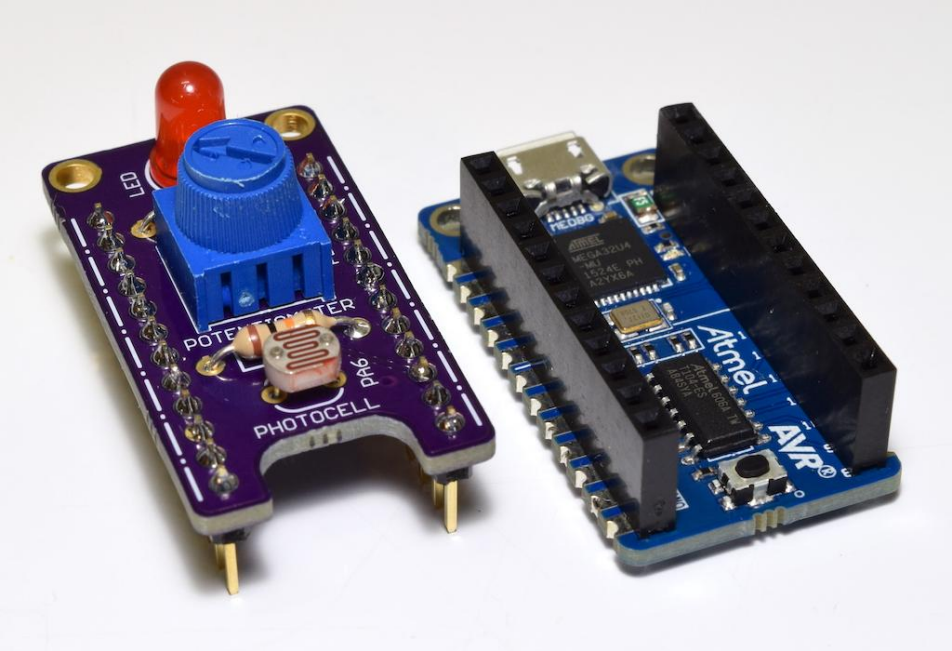
Learning Embedded Electronics with the Atmel ATtiny104 Xplained NANO



By: Daniel Watson

syncchannel.blogspot.com

Introduction

Contained in this guide are eight lessons that provide a brief introduction to embedded electronics and programming. The lessons are intended for individuals with basic knowledge of electronics and some prior programming experience. If you have used an Arduino before, then you should have no problem following along here. However, these lessons will make use of Atmel Studio 7, not the Arduino IDE. They are ideal for someone that wants to learn a little about embedded programming in standard C while exploring the capabilities of a modern 8-bit microcontroller.

The lessons make use of the Atmel ATtiny104 Xplained NANO and a small add-on board that was designed specifically for this project. The ATtiny104 XNANO can be purchased from major distributors for under $5 USD at the time of this writing. The PCB for the add-on board is a shared project on OSHPark. You can order the PCBs there and acquire the remaining parts from your favorite vendor. The necessary parts are so common that you probably already have them on hand. The PCB and parts make up a simple kit that would be suitable as a first soldering kit for yourself or your students.

Before starting these lessons, please examine the official Atmel documentation on the ATtiny104 XNANO that is linked below. The User Guide explains all of the features of the board and the pinout. The Getting Started guide fully describes how to connect the board to Atmel Studio 7 on your computer and start a new project. Additionally, an example program is provided that makes use of the LED and button on the board. Please test your board with that program and make sure everything is good to go with your setup before starting the lessons. Finally, I have included a link to the ATtiny102/104 datasheet.

Some basic soldering tools will be required to complete Lesson 1. You will also need a micro USB cable to connect the ATtiny104 XNANO to your computer. Some basic theory will be explained in each lesson that is relevant to the features of the microcontroller being explored. Even so, keep an internet browser or a knowledgeable instructor handy as you work through the lessons to augment the material contained in this guide.

Atmel ATtiny104 Xplained NANO User Guide:

http://www.atmel.com/Images/Atmel-42671-ATtiny104-Xplained-Nano_User-Guide.pdf

Atmel ATtiny104 Getting Started Guide:

http://www.atmel.com/Images/Atmel-42678-Getting-Started-with-Atmel-ATtiny102-104_ApplicationNote_AT12489.pdf

Atmel ATtiny102/104 Datasheet

http://www.atmel.com/Images/Atmel-42505-8-bit-AVR-Microcontroller-ATtiny102-ATtiny104_Datasheet.pdf

Lesson #1: Time to Solder

Collect up all of the parts you need to assemble the add-on board:

- PCB ordered from OSHPark (https://oshpark.com/shared_projects/AR0rxCyz)
- 10kΩ trimpot with knob
- Small photoresistor (5mm width)
- 5mm LED (color of your choice)
- 470Ω resistor, 1/4 watt, 5%
- 10kΩ resistor, 1/4 watt, 5%
- (2) 12-pin 0.1" female headers (snap off from a strip of breakaway headers)

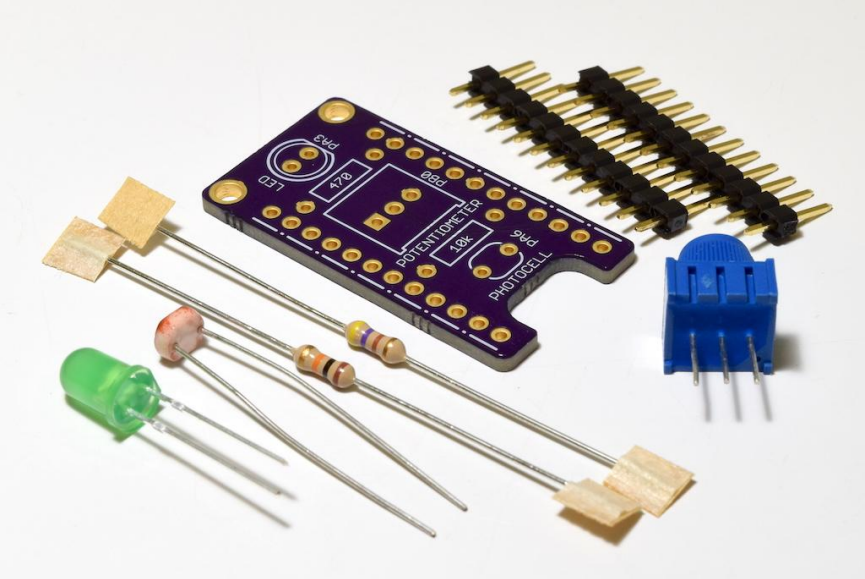


Figure 1: The parts necessary to assemble the add-on board for your ATtiny104 XNANO.

Warm up your soldering iron and insert the two resistors from the top of the board. The values are labeled on the silk screen to indicate where they go. The 470Ω resistor has a color code of [yellow-violet-brown], and the 10kΩ resistor has a color code of [brown-black-orange]. Flip the board over and solder the leads on the reverse side, then trim them flush.

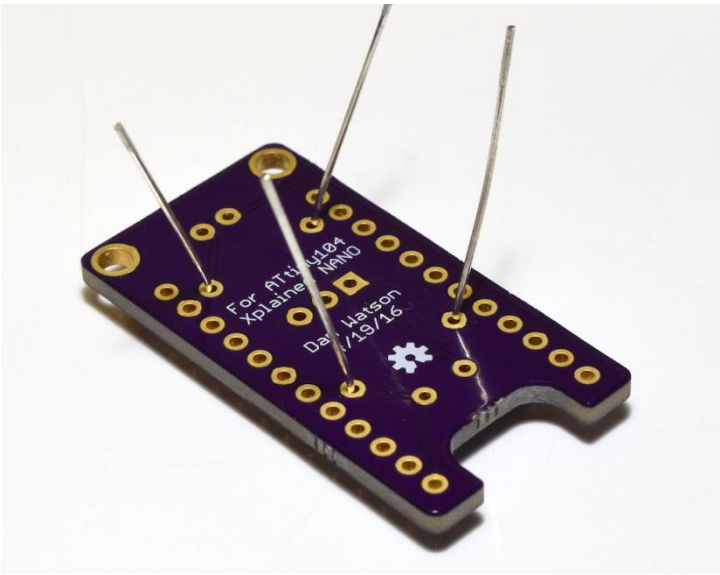


Figure 2: Start with the resistors when soldering the add-on board.

With the resistors done, insert the photoresistor. It is not polarized, so it can be inserted either way around. Once you solder that in place, install the LED and then the trimpot. Make sure to note the silk screen outlines of the parts on the board when you insert them. If you install the LED backwards it will not work.

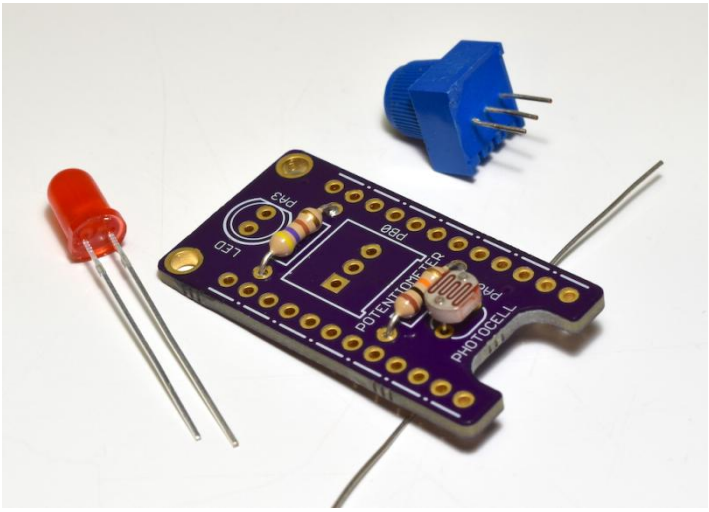


Figure 3: After the resistors, solder on the photoresistor, the LED, and then the trimpot.

To complete the add-on board, solder on the headers in the orientation shown in Figure 4. You can use a small breadboard to hold the headers and help line them up straight on the board. Tack down opposite corners and then visually check to make sure the headers look good and are flush against the bottom of the board. Once you are happy with the placement, finish soldering all of the pins. While we are taking care of the headers, also solder male headers onto your XNANO board as shown in Figure 5. This will allow you to connect the two boards together.

Once you are finished with the soldering, you can use some isopropyl alcohol or flux cleaner to clean the boards. An old toothbrush is also helpful for scrubbing away the flux residue. This is not entirely necessary, but it will make your work look a lot nicer.

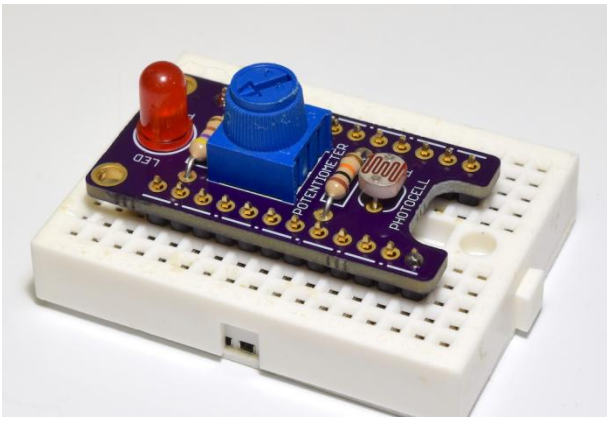


Figure 4: Use a small breadboard to help line up the header pins when you solder them.

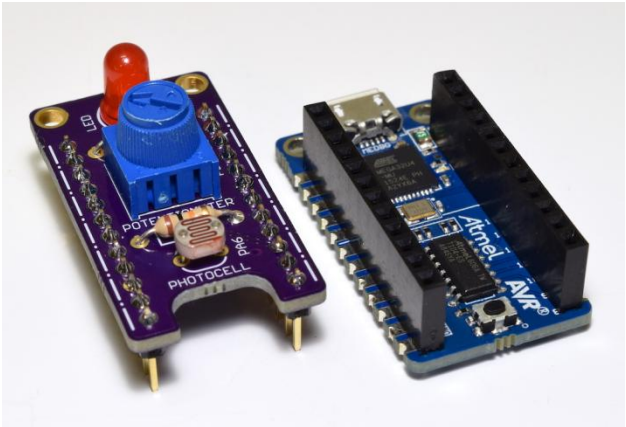


Figure 5: Our boards are assembled and ready to go!

Lesson #2: Hello World

You should have already entered and uploaded the Hello World program in the Getting Started guide, but let's try our own program that uses the LED on our freshly-assembled add-on board. Start a new project in Atmel Studio and type in the following program, then build it and upload it to the XNANO board.

```
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRA |= (1 << DDRA3); // PORTA, pin 5 output

    while (1) // loop forever
    {
           PORTA |= (1 << PORTA3); // Turn on LED
           _delay_ms(500); // Wait 500 ms
           PORTA &= ~(1 << PORTA3); // Turn off LED
           _delay_ms(500); // Wait 500 ms
    }
}
```

Program 1: Hello World Blinking LED

The comments in green give some explanation of what each line of code does. Even though this is a simple program, it makes use of some core features of the AVR microcontroller that we need to be aware of. The default clockspeed of the ATtiny104 is 1MHz on the XNANO board. We have to define that value before we include the *delay.h* library. The delay library available in Atmel Studio makes it easy to wait set periods of time. In our case, if we just toggled the LED at the full clock speed, it would change so fast we wouldn't be able to see it. By inserting a 500ms delay after we turn it on and turn it off, we cause the LED to blink approximately once per second.

The GPIO pins on an AVR microcontroller are inputs by default on start-up. To blink an LED, we need to make the pin it is connected to an output. This allows the pin to push current through the LED and light it up. This is done by modifying the Data Direction Register (DDR) for the port the pin is a part of. The LED is connected to pin 3 on PORT A (**PA3**), so we modify the

DDRA register. Notice how the syntax works, because you will be using this a lot in future lessons. A bit field is created with the bit corresponding to pin 3 turned on. This is then ORed with the current contents of **DDRA** to set the pin 3 bit while leaving the other bits alone. We can examine the **DDRA** register listing in the datasheet of the microcontroller so see that our code does indeed make sense.

Bit	7	6	5	4	3	2	1	0
	DDRA7	DDRA6	DDRA5	DDRA4	DDRA3	DDRA2	DDRA1	DDRA0
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bits 7:0 – DDRAn: Port A Input Pins Address [n = 7:0]

Figure 6: The **DDRA** register listing in the ATtiny104 datasheet. Set the **DDRA3** bit to make pin 3 on **PORTA** an output. (Source: ATTiny102/104 datasheet)

Similar logic is used to set pin 3 high and low to blink the LED. However, we need to modify the **PORTA** register this time to change states. When the corresponding bit is set, the pin is driven high (nominal 5V output) and the LED is turned on. When the pin is set low (nominal 0V output), the LED is reverse-biased and turned off. Notice that the syntax necessary to clear a single bit is different than the syntax necessary to set a bit in a register. We create an inverted bit field with only the necessary bit cleared, and then AND it with the register. This clears the bit and leaves all of the others unchanged. (Yay for Boolean logic!)

Extension:

For each lesson, optional extensions will be included to help you apply the knowledge you have gained in creative ways. For this Hello World program, modify it to turn on the LED for 100ms, then turn it off and wait for 900ms.

Lesson #3: Pushing Buttons

Button presses were part of the example program in the Getting Started guide, but let's revisit them to examine how the code works. Start a new project and enter the following program, then build it and upload it to the XNANO board.

```
#include <avr/io.h>

int main(void)
{
    DDRA |= (1 << DDRA3); // PORTA, pin 5 output
    DDRB &= ~(1 << DDRB1); // PORTB, pin 1 input

    PUEB |= (1 << PORTB1); // Enable pull-up function on PORTB, pin 1
    PORTB |= (1 << PORTB1); // Activate pull-up on PORTB, pin 1


    while (1) // loop forever
    {
       if (!(PINB & (1 << PINB1))) {  // The button is pressed
              PORTA |= (1 << PORTA3); // Turn on LED
       }

       else {
              PORTA &= ~(1 << PORTA3); // Turn off LED
       }
    }
}
```

Program 2: Button Press to turn on LED

We don't need delays in this program, so that eliminates two lines of setup code. However, we have some new lines of code to enable a pull-up resistor on pin **PB1**, which is connected to the button on the XNANO board. Pins used with buttons have to be set to a known state (either high or low) even when the button is un-pressed. If an input pin is left floating, it will drift high and low unpredictably. GPIO pins on AVR microcontrollers have built-in pull-up resistors that we can activate. We enable the function by modifying the **PUEx** register (**PUEB** in this case as our pin is on **PORTB**), then set the pin high in the **PORTx** register.

Also notice that the program sets pin **PB3** as an input. Input is the default mode, so the program will still work if you don't include this. However, it is good practice to always explicitly set your pin modes and states instead of relying on default conditions. This can become very

important in more complex programs where another line of code might (intentionally or unintentionally) modify the registers.

Reading the current state of an input pin is done through the Port In register (**PINx)**. We create a bit field with the pin 1 bit set, then AND it with the **PINB** register. If this returns TRUE, that means the button is **un-pressed**. Remember that the pin is pulled high, so we get inverted logic on our pin reads. When you press the button, it connects the pin to ground, returning FALSE. We use these logic states in an *if statement* to turn the LED on or off as we push and release the button.

Extension:

Modify the program to turn the LED on and leave it on for two seconds after a button press, then turn it off. To accomplish this, add back in the lines of code needed for the delay library and then incorporate a two second delay at the correct location in your main loop. Additionally, remember that if the button is un-pressed, we can simply *do nothing*.

Lesson #4: Using the USART

For this lesson, we are going to use the USART on the ATtiny104 to send data to the computer. You will need to use a serial terminal program for this lesson. You can use free programs such as CoolTerm or Putty, which are available online. Atmel also has a free terminal extension that you can download and add to Atmel Studio. To download it, go to Tools – Extensions and Updates in the program. Search for "Terminal", then download and install it. After Atmel Studio restarts, you can launch it by going to Tools – Terminal Window.

Start a new project, enter the following program and then build and upload it:

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

// Defines for USART setup
#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((F_CPU / (USART_BAUDRATE * 16UL)) - 1)

int main(void)
{
	// Change to 8MHz clock speed (Clock Division = 1)
	CCP = 0xD8;
	CLKPSR = 0x00;

	// Set up USART
	UCSRC = (1 << UCSZ0) | (1 << UCSZ1); // Set 8 bit characters for USART
	UBRRH = (BAUD_PRESCALE >> 8); // Set baud rate on USART
	UBRRL =  BAUD_PRESCALE;
	UCSRB = (1 << TXEN); // Enable Tx on USART

	uint8_t counter = 0;

    while (1) // loop forever
    {
		UDR = counter; // Send a single byte of data
		while (!(UCSRA & (1 << UDRE))) {}; // Wait for Tx to complete

		counter++;

		_delay_ms(1000);
    }
}
```

Program 3: Using the USART

There are several new lines of code in this program. First off, we are changing the clock speed of the microcontroller to 8MHz by disabling an internal divide-by-8 prescaler. This is done by writing a special value to the Configuration Change Protection register (**CCP**) and then immediately modifying the **CLKPSR** register. Read page 22 of the datasheet to learn more about the CCP register and page 31 to learn more about changing clock frequencies. The internal oscillator of the ATtiny104 is *actually* 8MHz, but thus far we've been using it at 1MHz with the default prescaler enabled. Every family of microcontroller has a different way of changing clock frequencies, but it often involves modifying some special bits of memory as we did here.

Next, we have to set up the USART for transmissions. The baud rate has to be defined, and this involves a calculation based on the clock speed of the microcontroller. Starting reading on page 90 of the datasheet to learn more about this. For now, we'll pick a very common baud rate of 9600 bps and let the lines of code shown calculate the correct prescaler setting for us. We then modify four registers to complete the setup. Some bits in **UCSRC** have to be changed to make our transmissions follow a common format of 8 bits per character. The prescaler for our baud rate goes into the **UBRRx** registers (one high byte and one low byte). Note the bit shift on the BAUD_PRESCALE value. Because it is longer than 8 bits and we need to fit it into two separate 8-bit registers, we have to perform a bit shift to fit the upper byte into the **UBRRH** register. When storing the lower byte into the **UBRRL** register, the upper bits will automatically get chopped off.

The last step in setting up the USART is to enable it. We only want to transmit here, so we will enable the transmitter but not the receiver. This is done by setting the **TXEN** bit in the **UCSRB** register. After that line of code executes, the USART is ready to send messages! On most AVRs, this is done by writing the byte of data you want to send to the **UDR** register. The transmission will begin automatically. Poll the status of the **UDRE** bit in the **UCSRA** register as shown to find out when the transmission is complete. Once the transmit buffer is empty, you can write in another byte to send more data. It is also possible to set up an interrupt and be notified when the transmission is complete, significantly improving the efficiency of your

program. For now, we can get by with this “blocking” method where we sit and wait in the program until the byte is transmitted.

All we need now is some data to send. A variable is created to act as a counter. After each serial transmission, the counter is incremented. The unsigned 8-bit variable type used only supports decimal values up to 255, so you will see it wrap back around to 0 in short order. Everything going on in the main loop should now make sense. At the start of the loop, the current count value is written into the **UDR** register to send it. Once the transmission is complete, we increment the counter, wait one second and then do it all over again.

Next, we want to receive these counter values on our computer. Open up the serial terminal program of your choice and set it to the COM port that corresponds to “mEDBG Virtual COM Port”. Set the baud rate to 9600, and change the byte representation to decimal or hexadecimal. Connect the terminal to the COM port and you should see your counter values streaming in once per second.

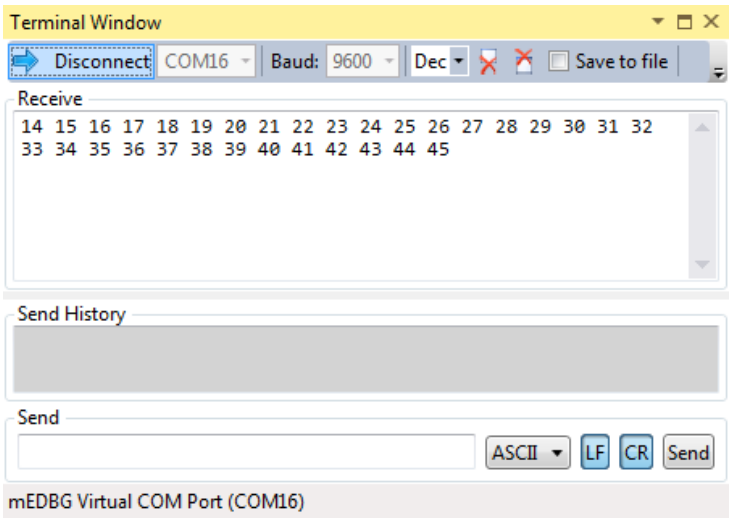


Figure 7: Counter values from the ATtiny104 XNANO being received using the Terminal extension in Atmel Studio. This terminal viewer is able to set the byte representation to Decimal. Hexidecimal representation would also work. However, do not leave the terminal in ASCII mode. We are sending raw bytes, not ASCII values.

This program was quite a bit more complicated than the programs used in previous lessons. Setting up a peripheral in a microcontroller often involves modifying several registers. Some of the lines of code may seem cryptic, but everything will make a lot more sense when you become familiar with the datasheet for the device. We only performed a minimal setup of the USART in our program. There are many more configuration options available to customize the microcontroller and make it do exactly what you need for your project.

The USART is a valuable tool that will be used extensively in future lessons. We are now able to read out button states, digital pin readings, analog voltage readings, and any other data we like from the microcontroller. The microcontroller is able to "talk" to our computer and give us the data. The USART allows bidirectional communication, so later we will even be able to send commands to the board and act on them in the program. USART communication is an important part of embedded electronics and programming.

Extension:

Modify the program to send out the current state of the button on the XNANO board (pressed or un-pressed). You will need to set up the appropriate pin as was done in Lesson #3, and then write different data into the UDR register.

Lesson #5: Using the Analog-to-Digital Converter

In this lesson, we will use another peripheral on the ATtiny104 called the analog-to-digital converter (ADC). It allows us to read analog voltage readings on certain pins, which gives us a lot more information than just the digital values of high or low. The resolution of the ADC in this microcontroller is 10-bit. Therefore, when an analog reading is performed a value between 0 and 1023 is returned that represents the voltage on the pin from 0V to 5V in 4.89mV steps. We are going to take analog readings from the potentiometer and then send them to the computer via USART. Upload the following program to your XNANO board:

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

// Defines for USART setup
#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((F_CPU / (USART_BAUDRATE * 16UL)) - 1)

int main(void)
{
	// Change to 8MHz clock speed (Clock Division = 1)
	CCP = 0xD8;
	CLKPSR = 0x00;

	// Set up the ADC
	ADCSRA |= (1 << ADEN) | (1 << ADPS2); // Enable ADC with prescaler = 16
	ADMUX |= (1 << MUX2); // Connect ADC4 input on PB0
	DIDR0 |= (1 << ADC4D); // Power saving feature

	// Set up USART
	UCSRC = (1 << UCSZ0) | (1 << UCSZ1); // Set 8 bit characters for USART
	UBRRH = (BAUD_PRESCALE >> 8); // Set baud rate on USART
	UBRRL =  BAUD_PRESCALE;
	UCSRB = (1 << TXEN); // Enable Tx on USART

	while (1) { // loop forever
		ADCSRA |= (1 << ADSC); // start single conversion
		while (ADCSRA & (1 << ADSC)) { } // wait until conversion is done

		uint16_t analogReading = ADCL | (ADCH << 8); // Store the analog reading
		UDR = (uint8_t)(analogReading >> 2); // Send a single byte of data
		while (!(UCSRA & (1 << UDRE))) {}; // Wait for Tx to complete

		_delay_ms(1000);
	}
}
```

Program 4: Using the ADC

The code to set up the USART and change the clock speed to 8MHz is retained from Lesson #4. The ADC is a peripheral so of course we have some lines of code to set that up as well. We are going to read the output voltage from the wiper of the potentiometer. This is connected to **PB0** on the microcontroller. The ADC in the microcontroller has numerous channels, and channel 4 is the one that is connected by default to **PB0**. We know this by looking at the pinout of the microcontroller in the datasheet:

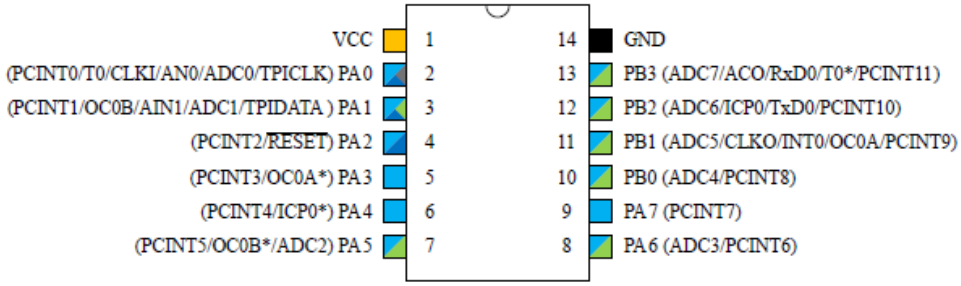


Figure 8: ADC4 is the channel connected to **PB0** and the wiper of the potentiometer. The appropriate bit in the **ADMUX** register is set to connect the ADC to that channel. (Source: ATTiny102/104 datasheet)

Once again we are performing a very minimal setup here for the purposes of this lesson. There are some very important customizations you can do in the ADC setup such as using a different voltage reference, enabling auto-triggering of conversions, and using interrupts to be notified when a conversion is completed. Take this opportunity to read the section of the datasheet that covers the ADC (starting on page 170).

To start a single conversion in the main loop, set the **ADSC** bit in the **ADCSRA** register. You can then poll the same bit waiting for it to change to 0, indicating that the conversion is complete and the analog reading is ready to be accessed (similar to polling the **UDRE** bit waiting for a USART transmission to complete). The lower eight bits of the reading are stored in the **ADCL** register, and the upper two bits are stored in the **ADCH** register. We read out both registers and store the bytes in the correct order into a new 16-bit unsigned variable. This requires some bit shifting as shown.

Now we want to send the reading over the USART. There is a slight problem though, in that we can only send eight bits at a time over serial. To keep things simple we will shift the analog reading left two bits when we write it into the UDR register. This throws away the lower two bits of the reading and scales our value to decimal 0-255. That is still plenty of resolution to see changes in the position of the potentiometer knob and the voltage on the wiper. Upload the program to the XNANO board, and then open the serial terminal window to see the values being sent once per second. Rotate the potentiometer knob left and right and watch the values change. When it is turned fully clockwise, the full 5V across the potentiometer is present on the wiper, so a value of 255 is returned. When the knob is fully clockwise, the wiper is effectively grounded and a value of 0 is returned.

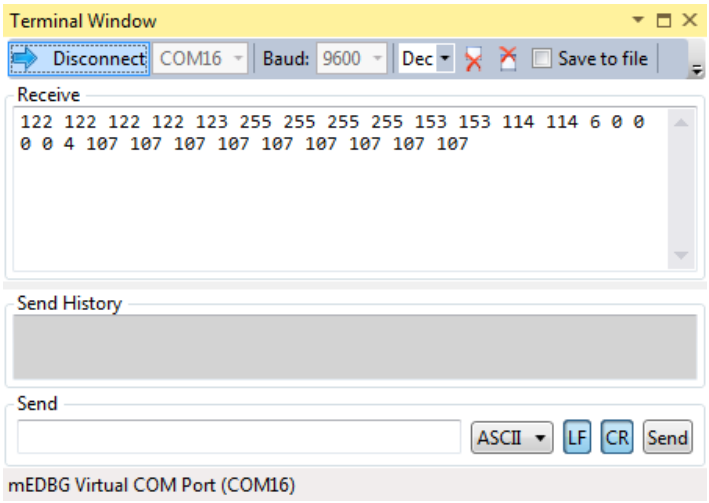


Figure 9: Viewing the ADC readings in the serial terminal window.

There is an excellent learning opportunity here. Modify the line of code that stores the ADC value into the analogReading variable as shown:

```
uint16_t analogReading = (ADCH << 8) | ADCL;
```

Build and upload the modified program and view the readings in the serial terminal while twisting the knob. You will find that the readings no longer update! This slight change to the code broke the program. On this microcontroller (and most AVRs), you MUST read the low byte of the reading in the **ADCL** register before reading the high byte in the **ADCH** register. If you

read them in the wrong order, your values will seem to never update, even though everything else in the program is correct. Little traps like this can cause a lot of frustration when you are trying to use a new peripheral on a microcontroller.

We have used two very important peripherals on the ATtiny104. You now know how to take analog voltage readings using the ADC and send them to a serial terminal on the computer with the USART. This will allow you to use a large number of sensors and devices in your future projects. The ability to read and interact with the real world is what makes microcontrollers and embedded electronics so interesting and useful.

Extension:

Modify the program to turn on the LED when the analog reading is above 2.5V, and turn it off otherwise. Use the full 10-bit reading stored in the analogReading variable for this. (2.5V equals a value of approximately 511 at that resolution) You will need to add back in the code to set up pin 3 on PORTA, as well as an *if statement* to check the value and act appropriately.

Lesson #6: Pulse Width Modulation (PWM)

In this lesson, we are going to use a Timer/Counter in the microcontroller in PWM mode to change the apparent brightness of the LED. PWM changes the duty cycle of the digital output signal, meaning that the signal will be high for a variable amount of time. If we send this signal to the LED, it will turn on whenever the signal is high and be off otherwise. Doing this at a very low frequency would cause the LED to blink. However, if we do it at a very fast frequency, our eyes will average out the fast blinks and make the LED appear brighter or dimmer depending on the duty cycle we set. This is a very useful feature on microcontrollers that can also be used to drive electromechanical devices such as servo motors.

Start a new project, enter the following program, build it and upload it to the board:

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
	// Change to 8MHz clock speed (Clock Division = 1)
	CCP = 0xD8;
	CLKPSR = 0x00;

	DDRA |= (1 << DDRA3); // Set pin 3 on PORTA as an output

	// Set up PWM on Timer 0 (8-bit timer)
	GTCCR |= (1 << REMAP); // Remap OC0A to PA3
	TCCR0A = (2 << COM0A0 ) | (1 << WGM01) | (1 << WGM00 ); // Set PWM mode
	TCCR0B = (1 << CS00 ); // Set PWM frequency

	// Set up the ADC
	ADCSRA |= (1 << ADEN) | (1 << ADPS2); // Enable ADC with prescaler = 16
	ADMUX |= (1 << MUX2); // Connect ADC4 input on PB0
	DIDR0 |= (1 << ADC4D); // Power saving feature

	while (1) { // loop forever
		ADCSRA |= (1 << ADSC); // start single conversion
		while (ADCSRA & (1 << ADSC)) { } // wait until conversion is done

		uint16_t analogReading = ADCL | (ADCH << 8); // Store the analog reading
		OCR0A = analogReading;
		_delay_ms(10);
	}
}
```

Program 5: Using PWM

The lines of code needed to set up the ADC are retained from Lesson #5. We also set **PA3** as an output to drive the LED. However, the LED will not be turned on by a normal digital output as we did in previous lessons. Instead, we are going to use a special pin function called the Output Compare Match. There are two compare match outputs (A and B) from the Timer, and we will use **OC0A**. If you look at the device pinout in Figure 8, you will see that **OC0A** has an asterisk next to it on **PA3**. This compare match output normally goes to pin **PB1**, but we can *remap* some of the peripheral outputs to send them to different pins. In our program, we need to set the **REMAP** bit in the **GTCCR** register to send the PWM signal to **PA3** and the LED.

A few bits need to be set in the **TCCR0A** register to put the Timer/Counter into one of the PWM modes. Additionally, **TCCR0B** is modified as shown to set the frequency of the PWM output and enable it. Take some time to read through the Timer/Counter section of the datasheet (starting on page 125) to better understand these settings. We are using only one of hundreds of different possible configurations.

To change the duty cycle of the PWM output, we store a compare value in the Output Compare Register (**OCR0A**). The counter will constantly count at the set frequency. A comparison between the count and **OCR0A** is used by the microcontroller to know when to toggle **OC0A** high or low. If the compare value is made smaller, then the compare match happens earlier in the count, changing the duty cycle. The opposite happens if we set a higher compare value. Thus we can change the duty cycle and the brightness of the LED just by changing the contents **OCR0A**.

We use the analog readings from the potentiometer as new compare values in OCR0A. This allows us to change the brightness of the LED by turning the knob. Try this out on your own board. When the knob is turned fully clockwise, the LED will appear to be off. As you rotate it counter-clockwise, the LED will gradually get brighter and brighter.

Extension:

Add back in the code to set up the USART and send your analog readings to the serial terminal as we did in Lesson #5. See how the various analog read values correspond to the brightness levels you are able to set on the LED by rotating the knob.

Lesson #7: Using the Photoresistor

Besides the potentiometer, we have a sensor on our board that can be read using the ADC: the photoresistor. It is set up in a voltage divider with the 10kΩ resistor. When bright light hits the photoresistor, its effective resistance goes down and the output voltage from divider increases. When the light level is very dim, the effective resistance goes up and the output voltage of the divider decreases. The divider output is connected to pin **PA6** (ADC channel 3). We will make a light level sensor by using the ADC to read the output voltage and turn on the LED when bright light hits the photoresistor.

Start a new project, enter the following program, build it and upload it to the board:

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
	// Change to 8MHz clock speed (Clock Division = 1)
	CCP = 0xD8;
	CLKPSR = 0x00;

	DDRA |= (1 << DDRA3); // Set pin 3 on PORTA as an output

	// Set up the ADC
	ADCSRA |= (1 << ADEN) | (1 << ADPS2); // Enable ADC with prescaler = 16
	ADMUX |= (1 << MUX1) | (1 << MUX0); // Connect ADC3 input on PA6
	DIDR0 |= (1 << ADC3D); // Power saving feature

	while (1) { // loop forever
		ADCSRA |= (1 << ADSC); // start single conversion
		while (ADCSRA & (1 << ADSC)) { } // wait until conversion is done
		uint16_t analogReading = ADCL | (ADCH << 8); // Store the analog reading

		if (analogReading > 800) { // Change value to adjust the threshold
			PORTA |= (1 << PORTA3);
		}

		else {
			PORTA &= ~(1 << PORTA3);
		}
		_delay_ms(10);
	}
}
```

Program #6: Making a Light Detector

This program is very similar to the one used in Lesson #5. The important changes include setting **PA3** as an output and connecting channel 3 to the ADC instead of channel 4 (which is connected to the potentiometer). A simple *if statement* is used to check the 10-bit analog reading and set the state of the LED. I have used a comparison value here of 800. Experiment with different values; you may have to change it depending on the brightness of your room. To test the light sensor, shine a bright light onto the photoresistor. The LED should turn on. When you remove the light and/or cover the sensor with your hand, the LED should turn off.

Extension:

Drive the LED with PWM as we did in Lesson #6. Use the analog reading from the photoresistor to set the brightness of the LED instead of the potentiometer reading.

Lesson #8: Sending Commands Over USART

In Lesson #4 we sent data to the computer via serial. In this final lesson, we are going to revisit the USART and use it to send commands *from* our computer to control the LED on the ATtiny104 XNANO add-on board. Start a new project and enter the following program, then built it and upload it to the board.

```
#define F_CPU 8000000UL

#include <avr/io.h>
#include <avr/interrupt.h>

// Defines for USART setup
#define USART_BAUDRATE 9600
#define BAUD_PRESCALE ((F_CPU / (USART_BAUDRATE * 16UL)) - 1)

int main(void) {
	// Change to 8MHz clock speed (Clock Division = 1)
	CCP = 0xD8;
	CLKPSR = 0x00;

	DDRA |= (1 << DDRA3); // PA3 output

	// Set up USART
	UCSRC = (1 << UCSZ0) | (1 << UCSZ1); // Set 8 bit characters for USART
	UBRRH = (BAUD_PRESCALE >> 8); // Set baud rate on USART
	UBRRL =  BAUD_PRESCALE;
	UCSRB = (1 << TXEN) | (1 << RXEN) | (1 << RXCIE); // Enable Tx, Rx and interrupt

	sei(); // Global interrupt enable

	while (1) { // loop forever
		// Do nothing!
	}
}

ISR(USART_RXC_vect) {
	char receivedByte = UDR;

	if (receivedByte == '1') { // If we got a '1', turn LED on
		PORTA |= (1 << PORTA3);
	}
	else if (receivedByte == '0') { // If we got a '0', turn LED off
		PORTA &= ~(1 << PORTA3);
	}

	UDR = receivedByte; // Echo the received byte back on the serial terminal
	while (!(UCSRA & (1 << UDRE))) {}; // Wait for Tx to complete
}
```

Program 7: Sending Commands Over USART

A few additional bits are set during the USART setup. The serial receiver is activated (**RXEN** bit) along with the receive complete interrupt (**RXCIE** bit). Additionally, global interrupts are activated by calling `sei()`. Interrupts allow tasks to be performed by the microcontroller in the background without tying up the main loop. You start some function on a peripheral, such as sending a serial character or taking an ADC reading, and then move on to do other things in your code. When the task is complete, an interrupt will fire that causes the microcontroller to execute a routine (Interrupt Service Routine or ISR) to handle it. Interrupts can also be generated externally on the pins of the microcontroller by things such as buttons.

Understanding and using interrupts is extremely important in embedded development. A single-core microcontroller such as the ATtiny104 cannot multitask (do multiple things at once). However, we can give the appearance of multitasking in our programs by using interrupts. Recall that in previous lessons we polled a status bit to wait for serial transmission to complete. Imagine if we had to do that on this program, waiting indefinitely in the main loop until a character arrived from the computer. We could get away with it if the microcontroller had no other tasks to perform in our project. However, in nearly all situations interrupts are the right way to do it. We can perform some core functions in the main loop, and background tasks can be monitored by interrupts. When an interrupt fires, the main loop is temporarily halted to handle it, and then the program returns exactly where it left of when the ISR is done.

Thanks to interrupts, the main loop in the program has no lines of code at all! Everything happens in the ISR. When it executes, we know that a received character must be waiting in the buffer. We store the contents of the UDR register into a variable and then check it. If it is an ASCII 1, the LED is turned on. If it is an ASCII 0, the LED is turned off. The received character is then echoed back to the computer using the serial transmitter. Notice the syntax for representing ASCII character in the *if statements*. When the value is surrounded by single quote marks (i.e., '0') the compiler will treat it as an ASCII character and store the appropriate hex value. If you didn't include the single quote marks, the compiler would treat it as a decimal value. We need to watch for ASCII 1s and 0s from the serial terminal (decimal 48 and 49 respectively) instead of decimal 1s and 0s.

Open up the Terminal window on your computer, type a 1 in the Send field, and hit enter. The character is sent to the microcontroller and the LED should turn on. Sending a 0 instead will turn the LED off. Also notice that the characters you send are being echoed back in the receive pane of the Terminal window.

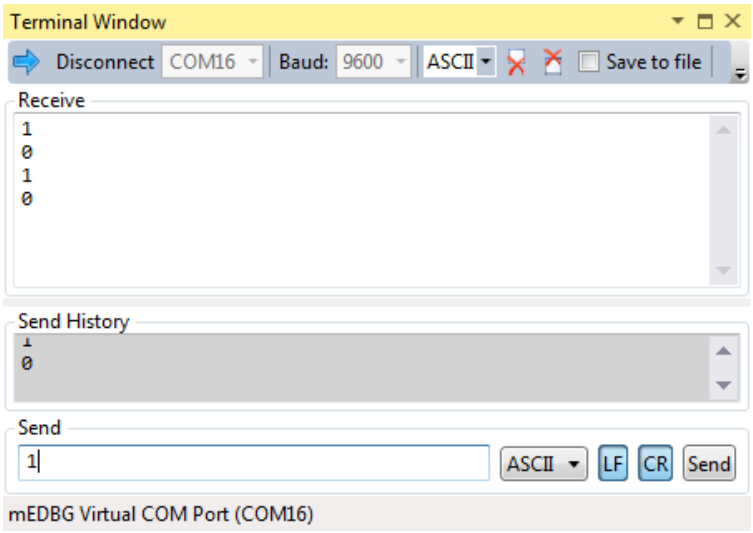


Figure 10: Controlling an LED using serial commands from the computer.

Extension:

Set the brightness of the LED using PWM and commands sent from the computer. To keep things relatively easy, stick to the single character commands 0 through 9. A '0' command should turn the LED all the way off, and then each successive command should set a higher brightness level until the LED is fully on at command '9'. You will need to figure out the appropriate **OCR0A** values for each command (the range of values is 0 to 1023) and create logic in the ISR to update **OCR0A** when a new command is received.

Going Forward

I hope you have enjoyed these lessons and learned a little about embedded programming with microcontrollers. We touched on only a small fraction of what you can do with your ATtiny104 XNANO board. Once you are done experimenting with the add-on board, you can remove it and use the XNANO board on its own. The headers are compatible with breadboard jumper wires, so you can hook up your own circuits on a breadboard to do many more experiments and projects.

There are numerous books and online resources available to further your knowledge. If you didn't fully understand something that was done in one of the lessons, please do not ignore it and move on. Research the topic and learn everything you can about it. Some aspects of working with microcontrollers can be confusing at first. It is important to move past that confusion and fully explore the capabilities of the device. This often involves reading and re-reading the datasheet or a more in-depth tutorial until you "get it".

You will eventually want to purchase more complex and powerful microcontroller development boards to further your studies. When you do, I recommend writing a demo program for it that uses every possible feature. It might take some time to do that, as well as a lot of exploration of the documentation. However, I think it is one of the best ways to learn a new board. Not to mention that you will have a goldmine of tested code when you are done to re-use in future projects!

Enjoy the world of embedded electronics.

Appendix A: Schematic

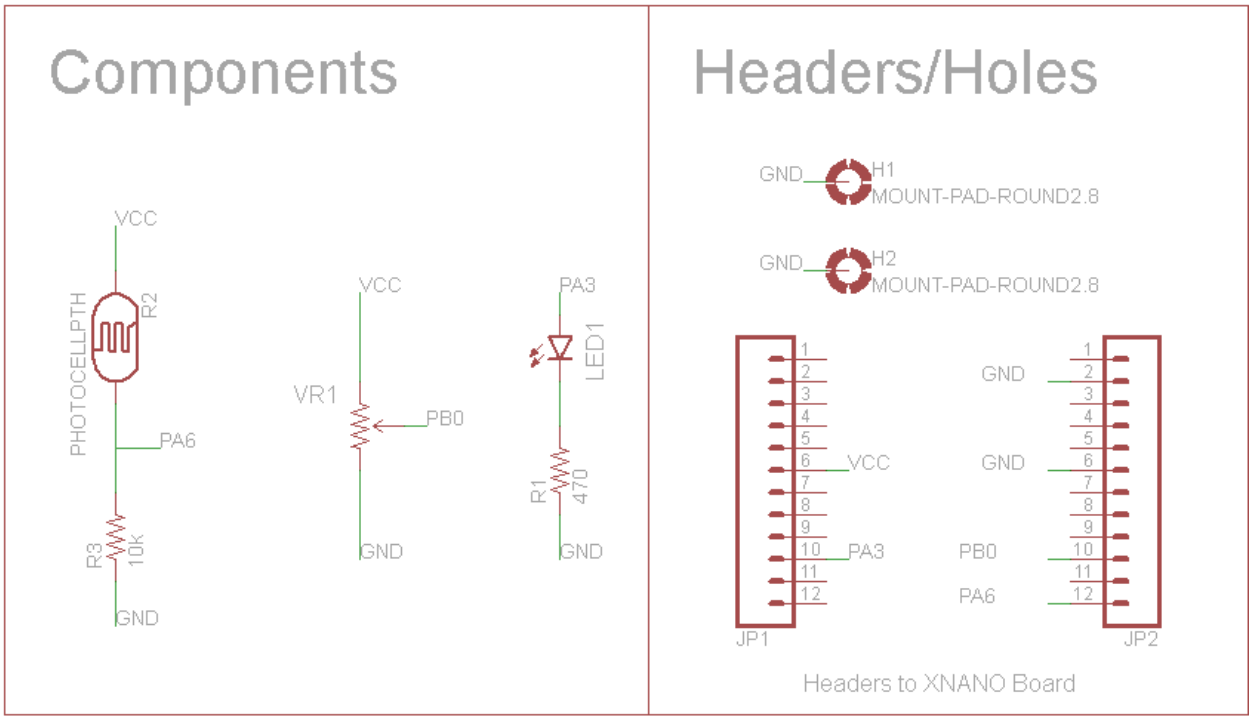


Figure 11: Schematic for the ATtiny104 Xplained Nano Add-On Board