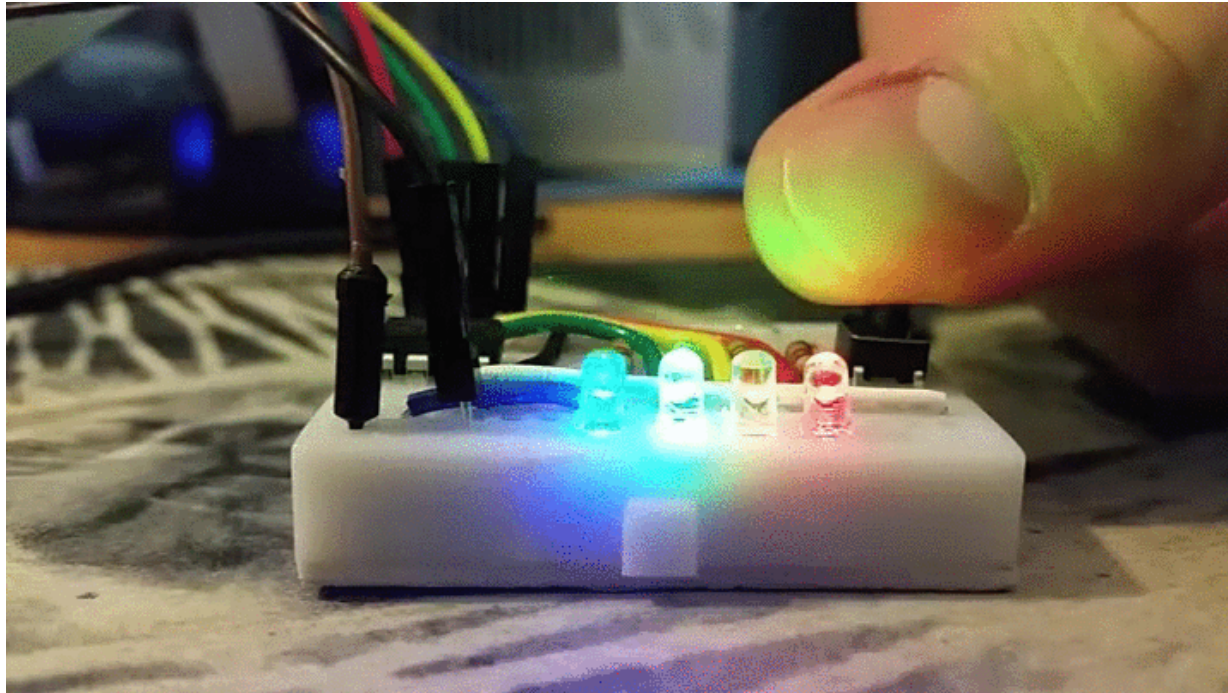


Embedded Thoughts

ATtiny85: Introduction to Pin Change and Timer Interrupts



Often when a microcontroller is being used, we want it to know when certain things occur, and then have something happen in response. A simple example is to have a pushbutton determine when an LED turns on.

One implementation that would satisfy this is called “polling”. Inside of the main while loop we could test the state of an input pin every loop iteration and turn on an LED depending on its state.

```
int main(void)
{
    initializePins();

    while(1)
    {
        if(pushbuttonPinIsHigh)
            turnOffLED;
        else
            turnOnLED;
    }

    return 0;
}
```

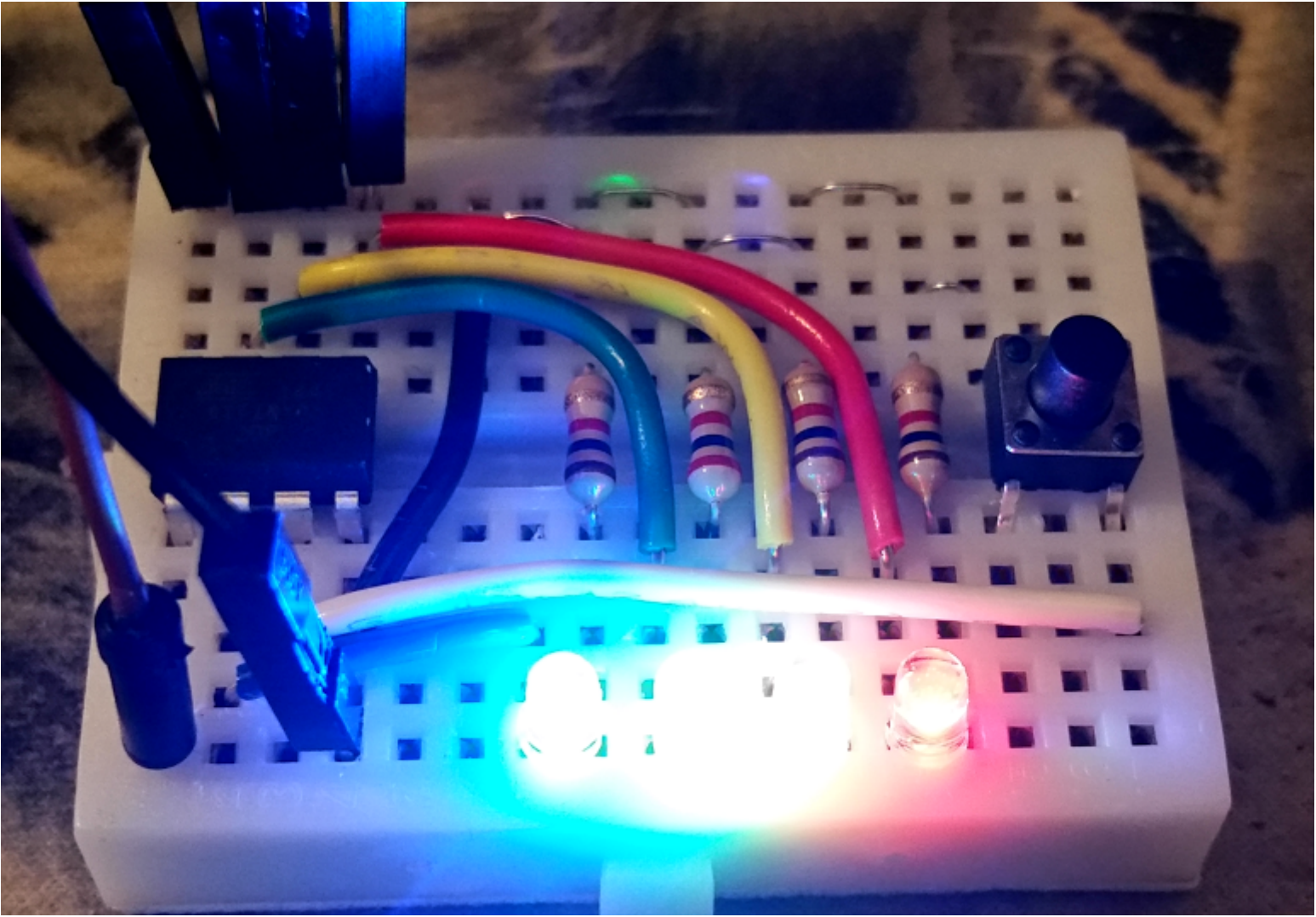
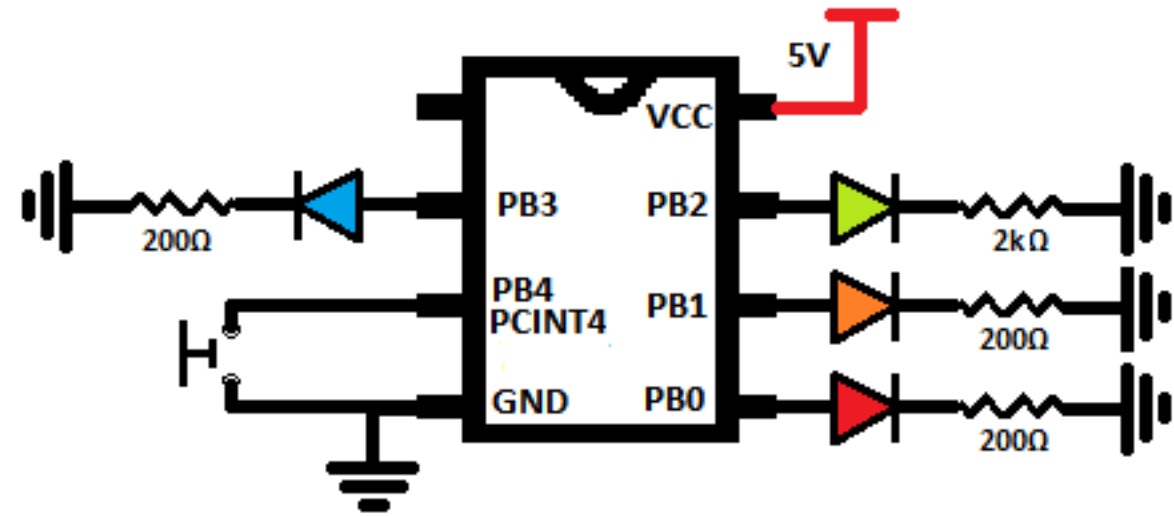
The above psuedocode checks the pin state of an input pin connected to a pushbutton and depending on the state turns on or off an LED.

Polling a pin state becomes tricky when you have to do other things in your event loop, especially things that take a lot of time. If for instance you needed to send or interpret some kind of serial data, doing so would take many clock cycles. This would essentially add latency to the response time of the LED to the pushbutton, as checking the pin state would have the same priority as every other sequential task in the while loop.

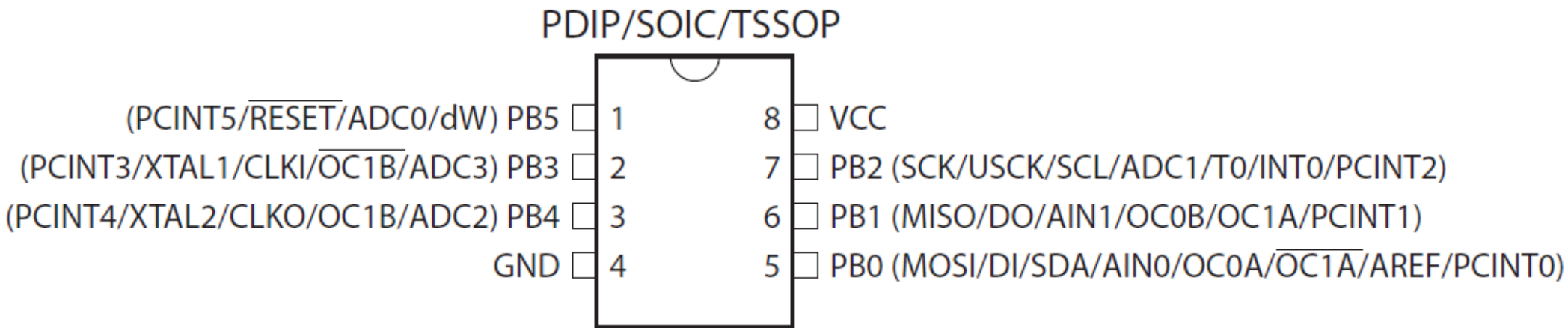
Hardware interrupts allow for asynchronous handling of system events. If an interrupt is set for a pin, when the pin state changes the code execution in the main loop halts and the code inside an Interrupt Service Routine (ISR) function is executed.

Let’s consider how we can use an interrupt to turn on 2 LEDs (green, red), while another two LEDs (blue, orange) are independently blinking. We will implement the blinking LEDs through the main while loop using a delay function, which would add a lot of latency to our pushbutton controlled LEDs if we controlled them by polling.

Keep in mind that while we could implement the blinking LEDs using timer hardware, we will use the delay implementation in place of more involved code that adds latency between code execution in the main while loop.



Above is the circuit diagram for our system and picture of an implementation on a tiny breadboard.



Above is the pinout for the ATtiny85, for reference.

Let’s consider the code implementation in chunks:

```
#define F_CPU 1000000UL
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
```

We define the CPU frequency as the default 1MHz, so that our delay function delays for the right time. We include, io.h for our pin and register macros (i.e. PB0, PORTB, etc.), delay.h for our delay function, and interrupt.h for our interrupt macros.


```
static inline void initInterrupt(void)
{
    GIMSK |= (1 << PCIE);    // pin change interrupt enable
    PCMSK |= (1 << PCINT4); // pin change interrupt enabled for PCINT4
    sei();                   // enable interrupts
}
```

We will initialize our interrupt in a function and use a pin change interrupt to allow our pushbutton to act as a hardware interrupt. A **pin change** interrupt on the ATtiny85 will look for a logical change on a PCINT pin (PCINT0-PCINT5), and if it finds one will set the pin change interrupt flag and call the PCINT0 interrupt vector. The PCINT0 interrupt vector will halt the execution of code in the main while loop and begin execution of the corresponding Interrupt Service Routine code which will act to turn on our green and red LEDs. Once the ISR toggles the LEDs on or off, the execution will return to where it left off in the main loop, keeping the other LEDs blinking away.

Chapter 9 of the [datasheet](http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf) (http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf) covers interrupts, explaining how to enable them and use them. To turn on the pin change interrupt and enable it for the correct pin (PCINT4) we will need to set some bits in specific registers.

GIMSK – General Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x3B	–	INT0	PCIE	–	–	–	–	–	GIMSK
Read/Write	R	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

We start by setting the Pin Change Interrupt Enable (PCIE) bit in the General Interrupt Mask Register (GIMSK).

PCMSK – Pin Change Mask Register

Bit	7	6	5	4	3	2	1	0	
0x15	–	–	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0	PCMSK
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Next we set the PCINT4 bit in the Pin Change Mask Register (PCMSK) to enable the PCINT4 (PB4) pin as a pin change interrupt.

Finally we call the function sei() from interrupt.h, which enables the global interrupt flag, thereby enabling interrupts for our system.

```
ISR(PCINT0_vect)
{
    PORTB ^= (1 << PB0) | (1 << PB2); // toggle pins PB0 and PB2, on logical change PCINT4 pin
}
```

Next we write our Interrupt Service Routine (ISR) for the Pin Change Interrupt vector (PCINT0_vect), which is the code that will be executed after a pin change interrupt occurs. When the pushbutton is pressed, the state of the PCINT4 pin will change, and the PCINT0 vector will be called, executing the code inside our ISR. Likewise, when the pushbutton is released, the pin state of PCINT4 will change again, and the code in the ISR will be executed. Inside our ISR we use XOR to toggle the state of PB0 and PB2 on PORTB, either turning on the green and red LED if off, or vice versa.

Keep in mind that ISR is a special function from interrupt.h that gets called when an interrupt flag is set and the corresponding interrupt vector is called.

```

int main(void)
{
    // initializations
    DDRB = 0x0F;           // enable PB0-PB3 as outputs
    PORTB |= (1 << PB4); // enable pullup on pushbutton output and PCINT4 interrupt
    initInterrupt();

    while(1)
    {
        _delay_ms(250);
        PORTB ^= (1 << PB1) | (1 << PB3);
    }

    return 0;
}

```

Inside our main loop we first set PB0-PB3 as outputs for our LEDs. To use our pushbutton with one end connected to PB4 and the other to ground, we will have to enable the internal pullup resistor. The internal pullup resistor is set by not enabling PB4 as an output, while setting the corresponding bit on PORTB.

We call our interrupt initialization function and enter our main while loop, which delays 250 ms between toggling PB1 and PB3 on and off.

ATtiny85 PCINT interrupt



Above is a short video demonstrating the behavior of the circuit.

One problem to keep in mind with using pushbuttons for hardware interrupts is that unlike with a polling implementation, there really isn't a way to debounce the input. This can lead to the springy/bouncy nature of the button introducing unintended interrupt triggers. Hardware debounced inputs can help with this, but typically for pushbutton inputs it is best to poll them and debounce the input.

While pin change interrupts don't work so well with pushbuttons, they do work well to interface with hardware peripherals that change one of their output lines to communicate something, perhaps that it is ready to receive data from the microcontroller, or that it is done processing some received data. These sort of implementations would not need a debounced input on the microcontroller and could be used with a pin change interrupt.

There are also interrupt vectors for when a timer has an compare match or when a timer overflows, i.e. it goes from 255 to 0. These can be useful when a system needs to do something regularly that is more involved than toggling an IO line.

We will use timer/counter 1 along with it's compare match interrupt to count from 0-15 in binary using the same circuit with 4 LEDs, having a 1 second delay between each increment. The previous post titled "ATtiny85: Blinking Without Clock Cycles" introduces timer/counter hardware, so make sure to reference it if you want to learn more.

Let's look at the code:

```

#define F_CPU 1000000UL
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint8_t i = 0;

```

We will be using an 8-bit variable ‘i’ to store the current value displayed on the LED counter. The **volatile** keyword makes sure that our compiler doesn’t optimize out (remove) our variable i. Since i will only be changed in an Interrupt Service Routine, which may never be executed, the compiler thinks that this variable is not going to be used and will remove it. By adding the volatile keyword, we prevent this from happening.

If you plan on altering a variable inside of an ISR make sure to give it the volatile keyword. Also, since ISR only takes an interrupt vector as an argument, any variable that it needs to change within it must be made global in scope.

```
static inline void initTimer1(void)
{
    TCCR1 |= (1 << CTC1); // clear timer on compare match
    TCCR1 |= (1 << CS13) | (1 << CS12) | (1 << CS11); //clock prescaler 8192
    OCR1C = 122; // compare match value
    TIMSK |= (1 << OCIE1A); // enable compare match interrupt
}
```

We initialize our timer and corresponding interrupt in a function that again sets the appropriate bits in the specific register detailed in the [datasheet](http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf) (http://www.atmel.com/images/atmel-2586-avr-8-bit-microcontroller-attiny25-attiny45-attiny85_datasheet.pdf) in chapter 12.

TCCR1 – Timer/Counter1 Control Register

Bit	7	6	5	4	3	2	1	0	
0x30	CTC1	PWM1A	COM1A1	COM1A0	CS13	CS12	CS11	CS10	TCCR1
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

We set the Clear Timer on Compare match (CTC1) Bit in the Timer/Counter 1 Control Register (TCCR1) to clear our counter to 0 on each match with our compare value.

We next set the bits that set the prescaler that will divide down our system clock frequency to be used by the timer/counter to increment the counter. Our system clock runs at 1 MHz, so if we set the prescaler to 8192, the clock used for the timer will have a period of $1/(1\text{E}6/8192) = 8.192\text{ ms}$. So we will need to count 122 of these adjusted clock cycles to amount to nearly 1 second ($122 \times 8.192\text{ms} = 999.424\text{ ms}$).

Table 12-5. Timer/Counter1 Prescale Select (Continued)

CS13	CS12	CS11	CS10	Asynchronous Clocking Mode	Synchronous Clocking Mode
0	1	1	1	PCK/64	CK/64
1	0	0	0	PCK/128	CK/128
1	0	0	1	PCK/256	CK/256
1	0	1	0	PCK/512	CK/512
1	0	1	1	PCK/1024	CK/1024
1	1	0	0	PCK/2048	CK/2048
1	1	0	1	PCK/4096	CK/4096
1	1	1	0	PCK/8192	CK/8192
1	1	1	1	PCK/16384	CK/16384

We consult the timer/counter1 prescale select table for the TCCR1 register, and see that we need to set bits CS13, CS12, and CS11 to set the 8192 clock prescaler.

Next we set the compare match value OCR1C to 122, such that the counter will count from 0 to 122 in around 1 second, at which point the compare match interrupt will be activated, the counter will be cleared, and the value i will be incremented.

TIMSK – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
0x39	–	OCIE1A	OCIE1B	OCIE0A	OCIE0B	TOIE1	TOIE0	–	TIMSK
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R	
Initial value	0	0	0	0	0	0	0	0	

Finally we set the Output Compare Interrupt Enable (OCIE1A) bit in the Timer/Counter Interrupt Mask Register (TIMSK), to enable the timer compare match interrupt we will use.

```
ISR(TIMER1_COMPA_vect)
{
    if(i == 15)
        i = 0;
    else
        i++;

    PORTB = i; // write updated i to PORTB
}
```

Next we define the ISR that receives the TIMER1_COMPA vector that is called when the timer compare value is matched. In this we increment i from 0 to 15 and then set PORTB to i, which takes the lower 4 bits of i that count up to 15 and sets pins PB0-PB3 to their values. This will display the value of i on the 4 LEDs.

```
int main(void)
{
    // initializations
    DDRB = 0x0F; // enable PB0-PB3 as outputs
    PORTB |= (1 << PB4); // enable pullup on pushbutton output
    initTimer1(); // initialize timer registers
    sei(); // enable interrupts

    while(1)
    {
    }

    return 0;
}
```

Inside main we set pins PB0-PB3 as outputs, enable the pullup on the pushbutton that is still attached, just in case, initialize the timer and timer interrupts with initTimer1(), and then enable all interrupts with sei(). This time our while loop is empty, left to take care of some other task, only to be briefly interrupted when it is time to update i and PORTB.

ATtiny85 Hexadecimal LED Counter With Timer1 and Interrupt



Above is a short video demonstrating the circuit behavior.

There are many different interrupt vectors to use on the ATtiny85.

Table 9-1. Reset and Interrupt Vectors

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset
2	0x0001	INT0	External Interrupt Request 0
3	0x0002	PCINT0	Pin Change Interrupt Request 0
4	0x0003	TIMER1_COMPA	Timer/Counter1 Compare Match A
5	0x0004	TIMER1_OVF	Timer/Counter1 Overflow
6	0x0005	TIMER0_OVF	Timer/Counter0 Overflow
7	0x0006	EE_RDY	EEPROM Ready
8	0x0007	ANA_COMP	Analog Comparator
9	0x0008	ADC	ADC Conversion Complete
10	0x0009	TIMER1_COMPB	Timer/Counter1 Compare Match B
11	0x000A	TIMER0_COMPA	Timer/Counter0 Compare Match A
12	0x000B	TIMER0_COMPB	Timer/Counter0 Compare Match B
13	0x000C	WDT	Watchdog Time-out
14	0x000D	USI_START	USI START
15	0x000E	USI_OVF	USI Overflow

Above is a table of all the interrupt vectors available, in order of priority. Priority makes it so that if two interrupt vectors are triggered at the same time, the one with higher priority is taken care of first.

The interrupt for INT0 is an external interrupt that is tied to pin PB2 that has a higher priority than a pin change interrupt, but essentially works the same with a few extra features, such as not only being able to trigger on a logical change but also a positive or negative edge change. The higher priority of INT0 over PCINT makes it useful for using for an external hardware interrupt that you want to be served before another that is set to PCINT. Getting your priorities straight is key.

To wrap it all up, interrupts make it possible to asynchronously handle system events. Being able to utilize the power of interrupts may seem intimidating at first, but taking the time to learn and master them will allow you to build more responsive, scheduled, and robust systems.

Advertisements



REPORT THIS AD



Zubný strojček Invisalign











Najpokročilejší systém priehľadných v na svete

invisalign

REPORT THIS AD

☐ June 6, 2016 ☐ July 29, 2016 ☐ Embedded

Thoughts

 Atmel  ATtiny  ATtiny85  AVR  Binary  counter  Interrupt  Interrupts  Microcontrollers  timer

3 thoughts on “ATtiny85: Introduction to Pin Change and Timer Interrupts”

1. **b harmon** says:

February 20, 2018 at 1:00 pm

Maybe I missed it but I couldn’t find the entire program so I could download it.

☐ [Reply](#)
2. **Fred Swart** says:

March 22, 2018 at 2:56 pm

Thanks for this post, it’s crystal clear and such a time saver for me. I studied it to succesfully make an interrupt-driven servo refresh. Helped a lot!

Google directed me to this post; in hindsight your post on no-cycle LED blinking might have been equally helpfull (yet more basic). I love ATtiny85’s and the stuff we can make them do.

Anyway, greatly motivated to read more of your posts; I will bookmark and return!

☐ [Reply](#)
3. Pingback: Noisy chips – gr33nonline

BLOG AT WORDPRESS.COM.



REPORT THIS AD