AVR ASM INTRODUCTION

Search this site

AVR ASSEMBLER TUTOR >

5. TIMER COUNTERS & PWM

A MORON'S GUIDE TO TIMER/COUNTERS v2.4

by RetroDan@GMail.com

TABLE OF CONTENTS:

INTRODUCTION:

A computer timer/counter is used to perform a task at a set interval. A timer that increments a counter once every second could be used to make a real-time clock by keeping track of the hours, minutes & seconds that have elapsed.

For this primer I use the AVR ATtiny13 for its simplicity & small number of pins. This chip has one timer and one I/O port (Port B) of which Bits 0-4 can be used for output. The ATtiny13 runs at 1.2MHz (9.6MHz Oscillator/8). The example programs should run on the ATtiny13 ATtiny25, ATtiny45 or ATtiny85. The circuits & code should

be easy to adapt to most any AVR Core chips.

To show our progress we connect an LED on Port B, Pin 0 to ground and make it blink. If you use more than 3 volts, insert a 100-220 Ohm resistor in-line with the LED.

The circuit diagram for our ATtiny13 circuit is:

```
                +3VDC      PINOUTS: 1-PORTB,5 (/RE
 ATTINY13       |                   2-PORTB,3
  .-----.       |                   3-PORTB,4
 --|1 A 8|--'                       4-GROUND
 --|2 T 7|--                        5 PORTB,0
 --|3 N 6|--                        6-PORTB,1
.--|4 Y 5|--D LED (PORTB0)          7-PORTB,2
|  `-----'   |                      8-Vcc
|            |
|            |
`------------+ GND
```

CHAPTER 1: A PAUSE ROUTINE:

A simple type of timer is a software pause or wait routine. They go around a software loop and do nothing but waste time. By changing the number of times the program goes around the loop we can adjust the time it spends there.

```
           PAUSE          PAUSE          PAUSE
ON:         +------+       +------+       +----->
            |//////|       |//////|       |///// ETC.
            |//////|       |//////|       |/////
OFF   +-----+//////+------+//////+------+/////
                  PAUSE          PAUSE          PAUSE
```

Here is a program that blinks the LED about twice per second. (An explanation follows the listing).

```
;-----------------------------------------
; ATNT_BLINKY1.ASM
; AUTHOR: DANIEL J. DOREY (RETRODAN@GMAIL.COM
;-----------------------------------------
```

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO

.DEF A = R16             ;GENERAL PURPOSE ACC
.DEF I = R20             ;INDEX
.DEF N = R22             ;COUNTER

.ORG 0000
ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP

;--------------;
; MAIN ROUTINE ;
;--------------;
MAIN_LOOP:
      SBI   PINB,0       ;TOGGLE THE 0 BIT
      RCALL PAUSE        ;WAIT/PAUSE
       RJMP MAIN_LOOP    ;GO BACK AND DO IT A

;----------------;
;PAUSE ROUTINES  ;
;----------------;
PAUSE: LDI N,0           ;DO NOTHING LOOP
PLUPE: RCALL MPAUSE      ;CALLS ANOTHER DO NO
       DEC N             ;CHECK IF WE COME BA
        BRNE PLUPE       ;IF NOT LOOP AGAIN
         RET             ;RETURN FROM CALL

MPAUSE:LDI I,0           ;START AT ZERO
MPLUP: DEC I             ;SUBTRACT ONE
        BRNE MPLUP       ;KEEP LOOPING UNTIL
         RET             ;RETURN FROM CALL
```

First the assembler reads the file TN13DEF.INC that contains standard definitions for the Attiny13, then we add some of our own definitions for the registers that we use.

```
.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC
.DEF I = R20             ;INDEX
.DEF N = R22             ;COUNTER
```

.ORG tells the assembler we want our program at bottom of memory (since we have no interrupts):

```
.ORG 0000
```

We use bit zero of Port B as output, so we write a one to the first bit of the Data Direction Register for Port B (DDRB) to indicate it is an output:

```
 ON_RESET:
     SBI DDRB,0            ;SET PORTB0 FOR OUT
```

In the main loop we flip the 0-bit of Port B with an SBI command, then call the pause routine, all in a loop that goes around for ever:

```
MAIN_LOOP:
      SBI   PINB,0        ;FLIP THE 0 BIT
      RCALL PAUSE         ;WAIT/PAUSE
       RJMP MAIN_LOOP     ;GO BACK AND DO IT A
```

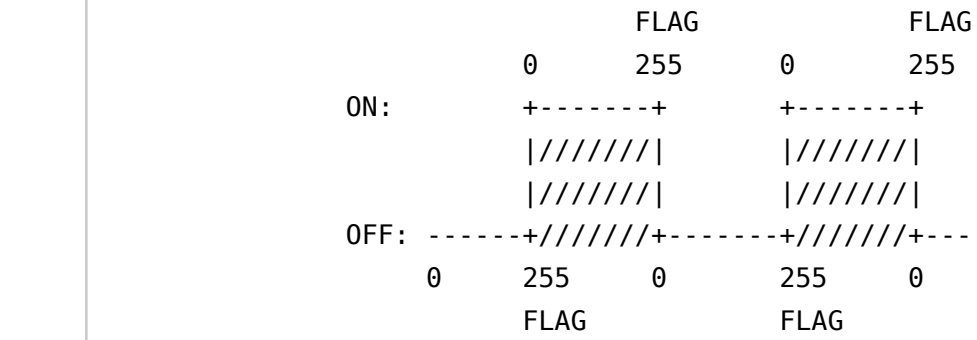
The next two subroutines waste time by going in loops. The first PAUSE routine calls the second routine MPAUSE to slow things down so we can see the LED blinking. Both routines load zero into a register then repeatedly subtracts one until it hits zero again (subtracting one from zero gives 255):

```
PAUSE: LDI N,0            ;DO NOTHING LOOP
PLUPE: RCALL MPAUSE       ;CALLS ANOTHER DO NO
       DEC N              ;CHECK IF WE COME BA
        BRNE PLUPE        ;IF NOT LOOP AGAIN
         RET              ;RETURN FROM CALL

MPAUSE:LDI I,0            ;START AT ZERO
MPLUP: DEC I              ;SUBTRACT ONE
        BRNE MPLUP        ;KEEP LOOPING UNTIL
         RET              ;RETURN FROM CALL
```

CHAPTER 2: WAIT-FOR-TIMER "NORMAL" MODE:

With this method we use the chip's timer. When operated in "normal" mode, it will count up from zero to 255 then set an over-flow flag. So

we wait for this flag to be set each time. Confusion about timers can be eliminated if we call them counters instead. They are counting circuits that you can use to create timers.



We set a "pre-scaler" to 1024 that divides-down the system clock by 1024 so we can see the LED blinking. Much of the confusion about pre-scalers can be avoided if they we refer to them as dividers instead. Note, setting the pre-scaler also activates the timer/counter.

```
LDI A,0b0000_0101    ;SET TIMER PRESCALER
OUT TCCR0B,A
```

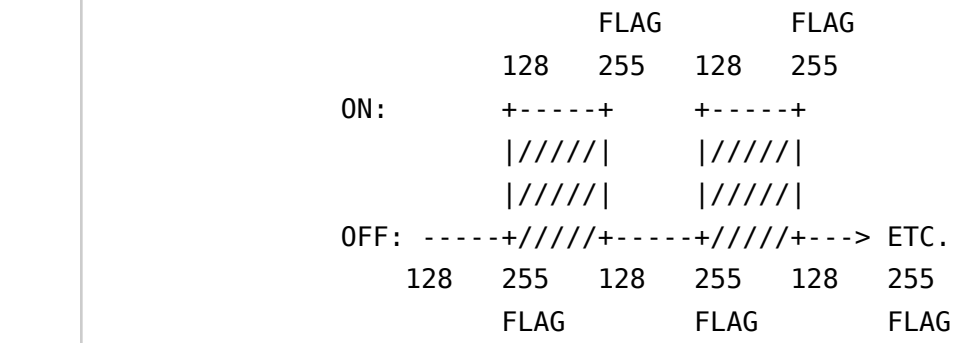
The remainder of the code is identical as our last program, except for the PAUSE routine. This time we wait for the timer to overflow past 255 and the overflow flag gets set. Then we reset the flag by writing a one (not a zero):

```
.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG 0000
ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0000_0101    ;SET TIMER PRESCALER
    OUT TCCR0B,A

MAIN_LOOP:
      SBI   PINB,0       ;FLIP THE 0 BIT
      RCALL PAUSE        ;WAIT
       RJMP MAIN_LOOP    ;GO BACK AND DO IT A

PAUSE:
```

```
PLUPE: IN   A,TIFR0        ;WAIT FOR TIMER
       ANDI A,0b0000_0010  ;(TOV0)
        BREQ PLUPE
       LDI  A,0b0000_0010  ;RESET FLAG
       OUT  TIFR0,A        ;NOTE: WRITE A 1 (
        RET
```

CHAPTER 3: MODIFIED WAIT-FOR-TIMER "NORMAL" MODE:

The last method provides us with only five speeds, because the pre-scaler/divider can only be set to five different values (1, 8, 64, 256, 1024). For more control we can pre-load the counter to any value from zero to 255, this will shorten the time it takes to reach 255 and over-flow.



For example, to blink the LED twice as fast we pre-load the timer to 128 with the following two lines:

```
       LDI   A,128        ;PRELOAD TIMER WITH
       OUT   TCNT0,A
```

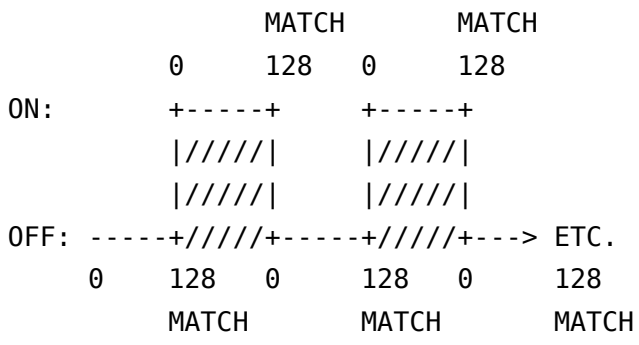
Our modified program goes twice as fast and is identical to our last program except that we have inserted the above two lines into the main routine which now looks like this:

```
;--------------;
; MAIN ROUTINE ;
;--------------;
MAIN_LOOP:
```

```
SBI   PINB,0       ;FLIP THE 0 BIT
RCALL PAUSE        ;WAIT
LDI   A,128        ;PRELOAD TIMER WITH
OUT   TCNT0,A
 RJMP MAIN_LOOP    ;GO BACK AND DO IT A
```

CHAPTER 4: THE TIMER-COMPARE METHOD: (CTC MODE CLEAR-TIMER-on-COMPARE)

The next method is very similar to the last. This time we set a “compare” value and a flag goes off when the timer reaches this value instead of 255. As the counter attempts to count up from zero to 255, when it matches our compare value, it sets a flag and the count restarts from zero. This simplifies our program because we don't have to reload the counter each time. By using different values for our compare we can alter the frequency of our output.



First we configure the timer into Clear-Timer-on-Compare (CTC) mode and set the pre-scaler to 1024:

```
LDI A,0b0100_0010    ;SET TO CTC MODE
OUT TCCR0A,A
LDI A,0b0000_0101    ;SET PRESCALER TO /1
OUT TCCR0B,A
```

We need to set the compare register to the value we want. Once again we use a value of 128:

```
LDI A,128            ;OUR COMPARE VALUE
OUT OCR0A,A          ;INTO THE COMPARE RE
```

Then we wait for the compare flag to be set. Note that we are waiting

for a different flag this time:

```
PAUSE:
PLUPE: IN   A,TIFR0        ;WAIT FOR TIME
       ANDI A,0B0000_0100  ;(OCF0A)
        BREQ PLUPE
```

Finally we reset the compare flag after it is triggered, to be ready for the next compare:

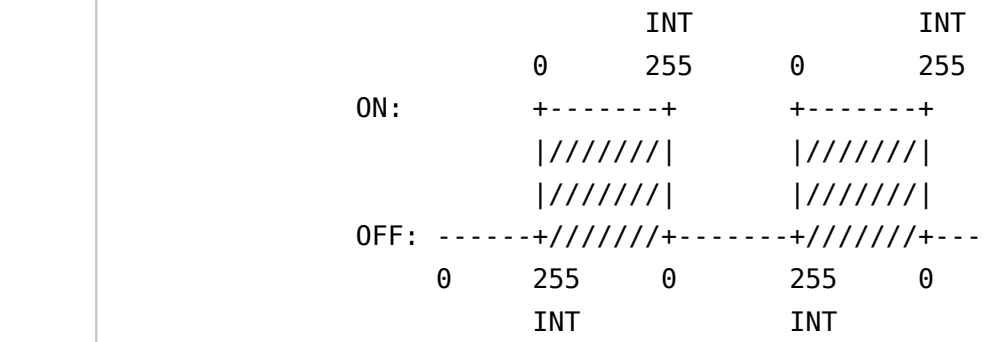
```
LDI  A,0b0000_0100  ;CLEAR FLAG
OUT  TIFR0,A        ;NOTE: WRITE A
 RET
```

The resulting code should now look like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG 0000
ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0100_0010    ;ACTIVATE CTC MODE
    OUT TCCR0A,A         ;SET FLAG ON COMPARE
    LDI A,0b0000_0101    ;
    OUT TCCR0B,A         ;SET PRESCALER TO /1
    LDI A,128            ;OUR COMPARE VALUE
    OUT OCR0A,A          ;INTO THE COMPARE RE
MAIN_LOOP:
    SBI   PINB,0         ;FLIP THE 0 BIT
    RCALL PAUSE          ;WAIT
     RJMP MAIN_LOOP      ;GO BACK AND DO IT A

PAUSE:
PLUPE: IN   A,TIFR0        ;WAIT FOR TIMER
       ANDI A,0B0000_0100  ;(OCF0A)
        BREQ PLUPE
       LDI  A,0b0000_0100  ;CLEAR FLAG
       OUT  TIFR0,A        ;NOTE: WRITE A 1 (
        RET
```

CHAPTER 5: THE TIMER OVER-FLOW INTERUPT METHOD:

This time we use a different strategy. Sitting inside a loop, or waiting for a flag, ties-up the processor. With the interrupt method, we set aside a small piece of code that gets automatically called whenever the over-flow flag is set. The timer/counter starts at zero and counts up toward 255. When it hits 255 and rolls over back to zero an interrupt is called and inside this interrupt routine we toggle the output:



Using an interrupt leaves the ATtiny free to do other tasks in the main-loop of the program. Here I have inserted a NOP command:

```
MAIN_LOOP:
      NOP                ;DO NOTHING
       RJMP MAIN_LOOP
```

When an interrupt is generated on the AVRs, the system looks to an interrupt vector table located at the bottom of memory (.ORG 0000). This table is composed of a series of jumps to routines, that you must supply, to handle the interrupts.

The first interrupt vector is for power-on & chip-reset. So we plug in a jump to the main body of our code (RJMP ON_RESET).

The interrupt vector for timer-overflow is located at ORG $0003, so we put a jump to our interrupt handler there:

```
.ORG 0000
    RJMP ON_RESET        ;RESET VECTOR
.ORG 0003
    RJMP TIM0_OVF        ;TIMER OVERFLOW VECT
```

Next we set the pre-scaler to divide by 1024 again and we enable the timer/counter over-flow interrupt, and also enable interrupts globally with the SEI command:

```
    LDI A,0b0000_0101    ;SET PRESCALER TO /1
    OUT TCCR0B,A         ;TIMER/COUNTER CONTR
    LDI A,0b0000_0010    ;ENABLE TIMER-OVERFL
    OUT TIMSK0,A
    SEI                  ;ENABLE INTERUPTS GL
```

We can eliminate the old PAUSE subroutine because it is no longer needed. The ATtiny automatically clears the Over-Flow Flag for us if an interrupt is generated, so we don't need to clear it in our interrupt routine. Because this is an interrupt, we end with a Return-from_Interrupt command (RETI):

```
TIM0_OVF:
      SBI   PINB,0       ;FLIP THE 0 BIT
       RETI
```

After we make these changes the program would like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG 0000
    RJMP ON_RESET        ;RESET VECTOR
.ORG 0003
    RJMP TIM0_OVF ; Timer0 Overflow Handler

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0000_0101    ;SET PRESCALER TO /1
    OUT TCCR0B,A         ;TIMER/COUNTER CONTR
    LDI A,0b0000_0010    ;ENABLE TIMER-OVERFL
    OUT TIMSK0,A
     SEI                 ;ENABLE INTERUPTS GL

;--------------;
; MAIN ROUTINE ;
;--------------;
MAIN_LOOP:
```

```
      NOP                ;DO NOTHING
       RJMP MAIN_LOOP

;----------------------------------;
; TIMER OVER-FLOW INTERUPT ROUTINE ;
;----------------------------------;
TIM0_OVF:
      SBI   PINB,0       ;FLIP THE 0 BIT
       RETI
```

CHAPTER 6: THE MODIFIED TIMER OVER-FLOW INTERUPT METHOD:

Since the "normal" mode for the timer is to start counting at zero and trip a flag and interrupt when it over-flows past 255. We can fine-tune the speed of our program again by pre-loading the counter register with a value during the interrupt so it starts counting at our own value instead of starting at zero.

```
                      INT         INT
                128   255   128   255
      ON:       +-----+     +-----+
                |/////|     |/////|
                |/////|     |/////|
      OFF: -----+/////+-----+/////+---> ETC.
           128  255   128   255   128
                INT         INT
```

To pre-load the counter, the following two lines would be added to the start of our program:

```
    LDI A,128           ;PRELOAD THE TIMER
    OUT TCNT0,A
```

We must also remember to re-load it again inside our interrupt routine for every time the timer/counter rolls past 255 and back to zero.

With these changes made the entire program should now look like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC
```

```
.ORG 0000
    RJMP ON_RESET        ;RESET VECTOR
.ORG 0003
    RJMP TIM0_OVF ; Timer0 Overflow Handler

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0000_0101    ;SET PRESCALER TO /1
    OUT TCCR0B,A         ;TIMER/COUNTER CONTR
    LDI A,0b0000_0010    ;ENABLE TIMER-OVERFL
    OUT TIMSK0,A
    LDI A,128            ;PRELOAD THE TIMER
    OUT TCNT0,A
     SEI                 ;ENABLE INTERUPTS GL

MAIN_LOOP:
      NOP                ;DO NOTHING
       RJMP MAIN_LOOP

TIM0_OVF:
      SBI   PINB,0       ;FLIP THE 0 BIT
      LDI A,128          ;RELOAD THE TIMER
      OUT TCNT0,A
       RETI
```

CHAPTER 7: THE TIMER COMPARE-MATCH INTERUPT METHOD:

This time we have the ATtiny generate an interrupt when the counter is equal to a “compare” register, instead of calling an interrupt when it hits 255 ($FF). The timer/counter starts at zero and counts up towards 255 but when it equal the compare register, it generates an interrupt and starts counting from zero again. Since the value of the compare register never changes, we don't have to re-load the timer/counter inside our interrupt routine each time.

```
                        MATCH       MATCH
                        INTRPT      INTRPT
                  0     128   0     128
```

```
ON:       +-----+     +-----+
          |/////|     |/////|
          |/////|     |/////|
OFF: -----+/////+-----+/////+---> ETC.
     0    128   0     128   0     128
          MATCH       MATCH       MATCH
          INTRPT      INTRPT      INTRP
```

Note that the location of the Compare-Timer Interrupt is in a different location than the last program. Our vector table looks like this now:

```
.ORG $0000
    RJMP ON_RESET        ;RESET VECTOR
.ORG $0006;<============= NOTE: DIFFERENT ADD
    RJMP TIM0_COMPA      ;TIMER-COMPARE INTER
```

First we tell the AVR to activate CTC Mode, then we set the pre-scaler to 1024:

```
    LDI A,0b0100_0010    ;SET TO CTC MODE
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
```

We enable the timer-compare interrupt by setting the correct bit in Timer-Mask Register (TIMSK0):

```
    LDI A,0b0000_0100    ;ENABLE TIMER-COMPAR
    OUT TIMSK0,A
```

Then we set the compare register to 128 and enable interrupts globally:

```
    LDI A,128            ;SET THE COMPARE REG
    OUT OCR0A,A          ;TO 128
     SEI                 ;ENABLE INTERUPTS GL
```

Our interrupt routine simply toggles the Port B0 output line:

```
      SBI   PINB,0       ;FLIP THE 0 BIT
       RETI
```

Our new program looks like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000
    RJMP ON_RESET        ;RESET VECTOR
.ORG $0006
    RJMP TIM0_COMPA      ;TIMER-COMPARE INTER

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0100_0010    ;SET TO CTC MODE
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,0b0000_0100    ;ENABLE TIMER-COMPAR
    OUT TIMSK0,A
    LDI A,128            ;SET THE COMPARE REG
    OUT OCR0A,A          ;TO 128
     SEI                 ;ENABLE INTERUPTS GL

MAIN_LOOP:
       RJMP MAIN_LOOP

TIM0_COMPA:
      SBI   PINB,0       ;FLIP THE 0 BIT
       RETI
```

CHAPTER 8: THE CTC "AUTOMATIC" WAVE-FORM MODE:

This time we have the timer/counter automatically toggle the output pin so we don't need an interrupt routine for that. With the CTC Wave-Form Method the timer/counter starts counting up. When it reaches the compare register, not only does it restart at zero and generate an interrupt, but we can also tell the system to toggle our output pin automatically. The timer/counter flips the OCA0/PortB0 line when the timer/counter equals the value (128) in the compare register OCR0A .

MATCH MATCH

```
                    TOGGLE      TOGGLE
              0     128   0     128
ON:           +-----+     +-----+
              |/////|     |/////|
              |/////|     |/////|
OFF: -----+/////+-----+/////+---> ETC.
     0    128   0     128   0     128
          MATCH       MATCH       MATCH
          TOGGLE      TOGGLE      TOGGLE
```

As before we setup Port B pin 0 as output. Then we setup the timer for CTC mode to toggle the Pin0 line, with a pre-scaler of /1024 and the compare register is set to 128. Since the timer automatically toggles the output, we can eliminate the need to use an interrupt.

The final program looks like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000
ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0100_0010    ;SET TO CTC MODE
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,128            ;SET THE COMPARE REG
    OUT OCR0A,A          ;TO 128

MAIN_LOOP: RJMP MAIN_LOOP: ;A DO-NOTHING LOOP
```

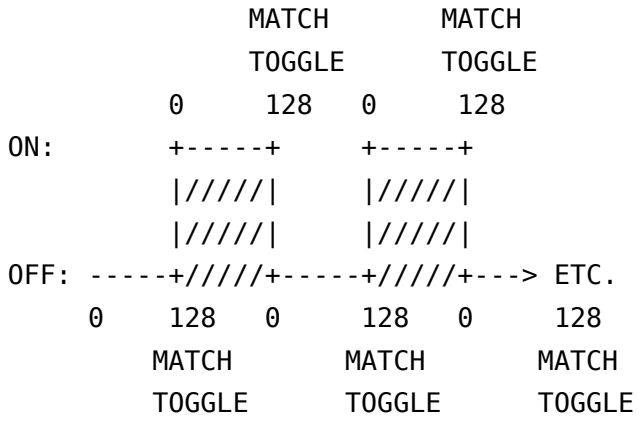
This method of producing a pulse on the OCA0 pin is almost identical to our next method: The Fast PWM mode.

CHAPTER 9: FAST PWM MODE 7:

Pulse Wave Modulation (PWM) is a way to generate waves using a timer/counter. In the “Fast Mode” the counter goes only in the up direction from zero as we've seen in our previous methods. (The other PWM mode counts up & down). This means the processor can create

faster maximum frequencies than other PWM mode, hence the “Fast” name.

The Fast PWM wave generation mode is very similar to the previous CTC mode. The timer/counter starts at zero and increments until it reaches the value in the compare register OCR0A then the OCA0/PortB0 output pin is toggled and the count resets back to zero. So our LED will flip between on and off every time the timer/counter reaches 128 just as it did in the CTC method.



To configure our timer/counter into PWM mode we need to set the WGM00,WGM01 bit of the Timer/Counter Control Registers TCCR0A and we also set the COMP0A0 bit to instruct the timer that we want it to toggle the output on a compare-match:

```
LDI A,0b0100_0011    ;SET TO FAST PWM MOD
OUT TCCR0A,A         ;[-,COMP0A0,-,-,-,-,
```

To get our timer/counter into the PWM Mode 7, we must also set the WGM02 bit which is located in TCCR0B the register, the same register we use to set the pre-scaler to 1024:

```
LDI A,0b0000_1101    ;SET PRESCALER/DIVID
OUT TCCR0B,A         ;[-,-,-,-,WGM02,CS02
```

Once again we set the compare register to 128:

```
LDI A,128            ;SET COMPARE TO 128
OUT OCR0A,A
```

Our Fast PWM program looks like this:

```
.INCLUDE "TN13DEF.INC"   ;(ATTINY13 DEFINITIO
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000
ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b0100_0011    ;SET TO FAST PWM MOD
    OUT TCCR0A,A
    LDI A,0b0000_1101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,128            ;SET COMPARE TO 128
    OUT OCR0A,A

MAIN_LOOP: RJMP MAIN_LOOP;A DO-NOTHING LOOP
```

By varying the value of the compare register OCROA it is possible to vary the frequency of our output wave.

```
    LDI A,64             ;SET COMPARE TO 64
    OUT OCR0A,A
```

CHAPTER 10: THE PWM FAST MODE 3 WITH DUTY CYCLE:

In our last program, we could vary the frequency but not the duty cycle. Using PWM Mode 3 the timer/counter starts at zero and counts all the way to 255, but the output bit on PortB0 is turned on when the counter is at zero and shuts it off when it matches our compare register. If we change the value in the compare register, we have the effect of controlling how long our output bit is on while the counter goes from zero to 255. If the compare value is low we send out short blips. If the compare value is larger it sends out a long wave.

For example if the compare value is 8 (OCR0A=8) we get short waves:

```
      0 8     0 8     0 8     0 8     0
ON:  +-+     +-+     +-+     +-+     +-
     |/|     |/|     |/|     |/|     |/
     |/|     |/|     |/|     |/|     |/
```

```
OFF: +/+-----+/+-----+/+-----+/+-----+/
     0     255 0   255 0   255 0   255
```

If the compare value is 200 (OCR0A=200) we get a long wave each time:

```
     0   200     200 0   200 0       0
ON:  +-----+ +-----+ +-----+ +-----+ +-
     |/////| |/////| |/////| |/////| |/
     |/////| |/////| |/////| |/////| |/
OFF: +/////+-+/////+-+/////+-+/////+-+/
     0     255     255     255     255
```

If we were to connect our circuit to a motor controller, when OCR0A = 8 the motor would go slow, but if the value of the compare register is 200 the motor would go faster. The reason is the difference in the amount of energy over the same period of time each sends out when the OCA0 pin is on. If each “/” in the diagrams above represent one unit of energy, you can see that when OCR0A=200, there is a lot more electical current coming from the output pin then there is when OCR0A=8. This represents a changing of the duty cycle.

To setup for this method, we first tell the system we want to use PWM Mode and we want it to turn on the output port when the counter rolls over to zero and to shut off when it matches the value we have in the compare register.

```
LDI A,0b1000_0011    ;[COM0A1,-,-,-,-,-,W
OUT TCCR0A,A         ;SET TO FAST PWM MOD
```

Since we are using PWM Mode 3 we do not set the WGM02 bit but we still need to set the prescaler/divider to /1024:

```
LDI A,0b0000_0101    ;[-,-,-,-,WGM02,CS02
OUT TCCR0B,A         ;SET PRESCALER/DIVID
```

Here is where we set the compare register, you can try different values here. My LED is barely visible when OCR0A is 8 and is bright when it is 200:

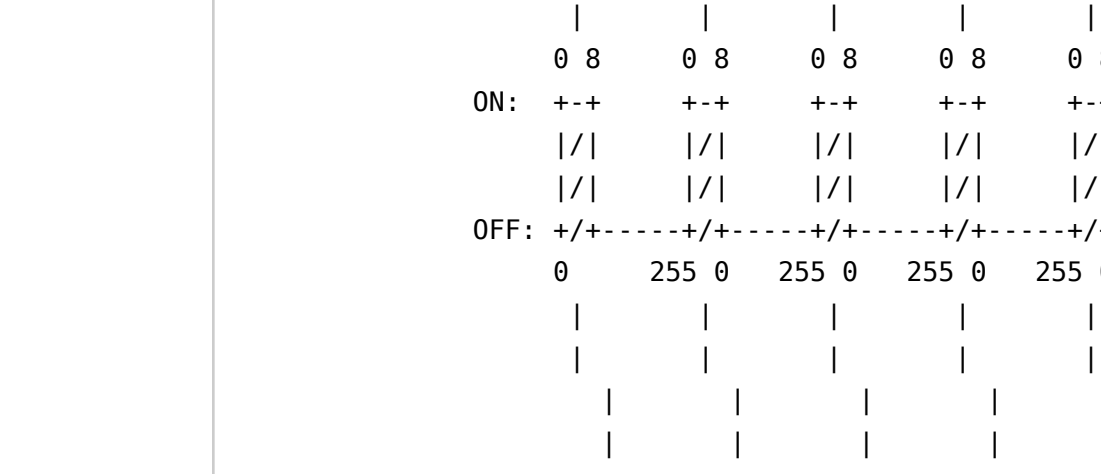
```
LDI A,8              ;DIFFERENT VALUE
OUT OCR0A,A          ;FOR COMPARE
```

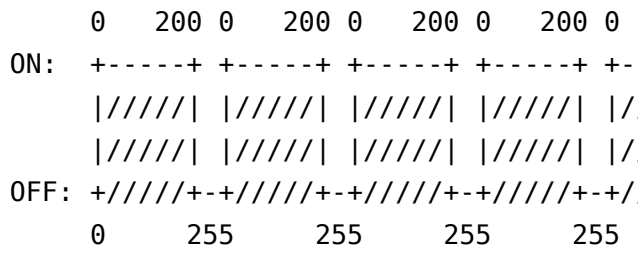
After we make those changes our program becomes:

```
;-------------------------------;
; ANTY_BLINKY_#10 FAST PWM MODE ;
;-------------------------------;
.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b1000_0011    ;SET TO FAST PWM MOD
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,8              ;DIFFERENT VALUE
    OUT OCR0A,A          ;FOR COMPARE

MAIN_LOOP: RJMP MAIN_LOOP;A DO-NOTHING LOOP
```

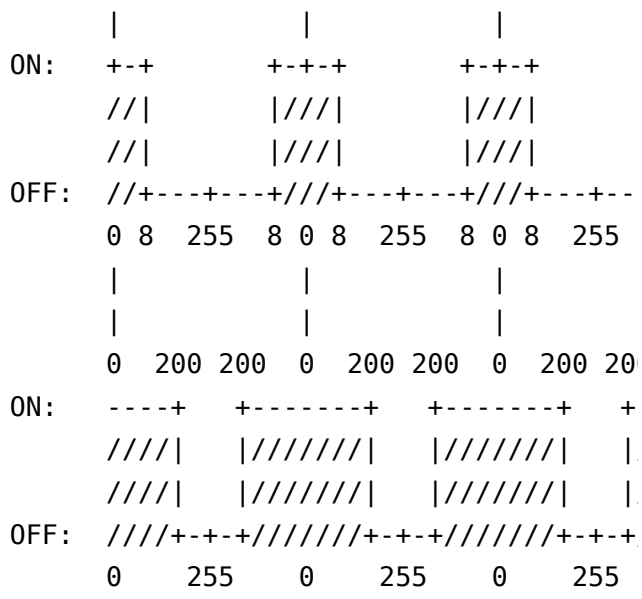
CHAPTER 11: THE PWM PHASE CORRECT MODE:

Using the previous method, we could vary the duty cycle but we were changing the “phase” of our wave. If we look at the centre of each wave from our last example, notice that the centre of our wave, moved forward when we changed the value of our compare value from eight to 200. The centres of the two waves are out-of-sync and out-of-phase:





Many motor driver applications require a wave with a fixed phase, one that stays in sync with the motor turning. To create a “phase correct” wave the timer/counter starts at zero and counts up just like with the Fast PWM Mode, but this time when the counter gets to the top it reverses and counts back down. When the timer matches the compare register on the way up it turns off our output bit, but on the way down when it hits the compare value, it turns back on our output bit. The result is a symmetrical wave that stays in sync (a phase correct output) even when you vary the duty cycle:



Using this mode you can create phase corrected waves but the maximum frequency is lower than the “Fast” mode because the timer goes through twice as many steps to complete a cycle compared to the “Fast” mode. In fast mode it only counts from zero to 255, but in the Phase Correct Mode the timer/counter goes from zero to 255 then back to zero again.

To change to PWM phase correct mode we need to set the WGM00 in

the TCCR0A register:

```
        LDI A,0b1000_0001    ;SET TO PWM MODE 1
        OUT TCCR0A,A
```

Note that this time we don't set the WGM02 bit so we only have to set the pre-scaler/divider in the TCCR0B Register:

```
        LDI A,0b0000_0101    ;SET PRESCALER/DIVID
        OUT TCCR0B,A
```

Your eyes might not see much difference this time, but the phase of the output is now symmetrical:

Here is the completed program:

```
;----------------------------------------;
; ANTY_BLINKY_#11 PHASE CORRECT PWM MODE ;
;----------------------------------------;

.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b1000_0001    ;SET TO PWM MODE 1
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,200            ;DIFFERENT VALUE
        OUT OCR0A,A        ;FOR COMPARE

MAIN_LOOP: RJMP MAIN_LOOP;A DO-NOTHING LOOP
```

CHAPTER 12: TIMER MATH:

Timer mathematics might seem like voodoo to you, but all you really need to remember is:

```
TIME-PERIOD = 1/FREQUENCY
```

```
- or -

FREQUENCY = 1/TIME-PERIOD
```

For example, if you have a 1MHz clock and you want a routine that goes off 50 times a second (50Hz), how long should your routine wait between pulses?

```
TIME-PERIOD = 1/FREQUENCY
TIME-PERIOD = 1/(50Hz)
TIME-PERIOD = 0.02 Sec
```

Therefore you would need to setup a counter that goes off every 20 mSec. How high does our timer have to count to create this 0.02 second delay? In other words, what do you have to divide the clock frequency of 1MHz to get a 0.02 second delay?

```
  FREQUENCY = 1/TIME-PERIOD
FREQUENCY/X = 1/TIME-PERIOD
1,000,000/X = 1/0.02
          X = 1,000,000 x 0.02
          X = 20,000
```

So a timer that counts 20,000 times would produce a 50Hz signal. Since the first count is usually zero and not one, we subtract a one from our answer to get a compare value of 19,999. Counting from zero to 19,999 actually takes 20,000 steps.

For the next example let us assume a clock speed of 1.2MHz with a pre-scaler/divider of 64. How long would it be between clock ticks?

```
     FREQUENCY = 1/TIME-PERIOD
CLOCK/PRESCALER = 1/X
  1,200,000/64  = 1/X
     1,200,000X = 64
              X = 64/1,200,000
              X = 0.0000533 Sec (5.33 uSec)
```

Lets say you hook a small speaker up to an output pin of your AVR and you want to play the tuning note A (440Hz). If your clock speed is 2MHz and your pre-scaler/divider is set to 64. How long do I need to wait between pulses?

```
TIME-PERIOD = 1/FREQUENCY
          X = 1/440
          X = 0.002273 (2.273 mSec)
```

What is the value you should set your compare register to? Your source frequency is 2Mz divided by a pre-scaler of 64:

```
SOURCE FREQUENCY = 2,000,000/64 Hz
                 = 31,250 Hz (31.25 KHz)
```

Now what value do you have to divide 31.25 KHz by to achieve a delay of 2.273 mSec?

```
FREQUENCY = 1/TIME-PERIOD
 31,250/X = 1/0.002273
        X = 31,250 x 0.002273
        X = 71.03
```

So we would use a counter value of 70, remember we subtract one because most systems start the count at zero not one. That would yield a frequency of about 440Hz.

If we use a counter value of 70, what would be the exact theoretical frequency?

```
TONE-FREQUENCY = (CLOCK/PRESCALER)/71
               = (2,000,000/64)/71
               = 31,250/71
               = 440.141 Hz
```

I hope the above two examples show you how easy it is to work with frequencies and timers if you remember three things:

```
1. FREQUENCY   = 1/TIME-PERIOD
2. TIME-PERIOD = 1/FREQUENCY
3. To subtract one for your compare values si
```

CHAPTER 13: FINAL FAST PWM PROJECT:

As our final project we are going to use the same circuit as above to blink an LED, but we are going to slowly change its duty cycle so it goes from off, then to blinking, then to steady-on state.

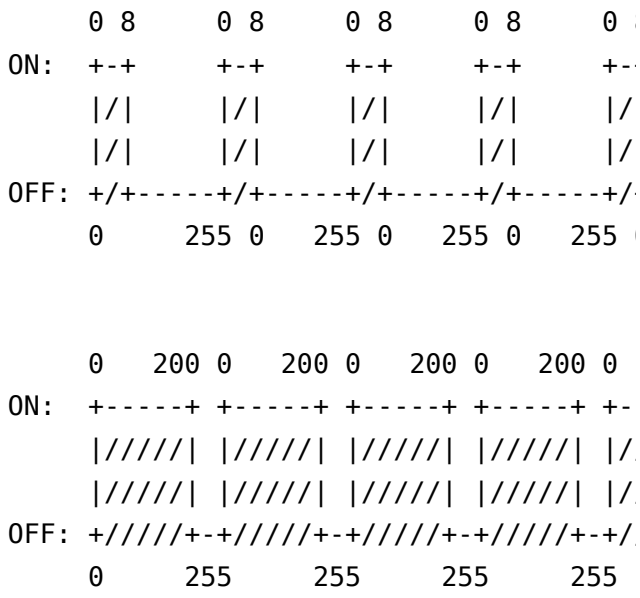
To do this we are going to use PWM Mode 3 (FAST PWM). In this mode the timer/counter starts at zero and counts to 255 then rolls over back to zero again. The system turns on our output pin OCRA (PORTB0) every time the count is zero.

The counter starts at zero with the output on, then as it counts up to 255, when it matches the value we have stored in the OCRA0 register, it shuts off our output but continues counting up to 255. Then it rolls back to zero again turn back on the output pin and starts over.

So if we have a small value in our compare register like eight, the output is only on for the time it takes to count from zero to eight as the counter goes from zero to 255. So our output is off most of the time, and our LED is barely visible, if at all.

If, on the other hand, OCRA0 is set to 200. Then the output pin is on for the values from zero to 200, as it counts to 255, then our output pin is on for most of the time and our LED appears to be steady-on.

This diagram illustrates the two different wave patterns:



We examine the data sheet for the Attiny13 and find the chart of the various timer/counter modes. We want Fast PWM since we are not concerned with Phase and we want our counter to go from zero to 255. This is the fourth entry and it indicates that we need to set the WGM00 & WGM01 bits of the timer/counter control register

(TCCR0A):

```
WGM2 WGM1 WGM0
 0    0    0   - Normal Mode
 0    0    1   - PWM Phase Correct (Top=255)
 0    1    0   - CTC Mode
 0    1    1   - Fast PWM (Top=255)
 1    0    0   - Unused
 1    0    1   - PWM Phase Correct (Top=OCRA0
 1    1    0   - Unused
 1    1    1   - Fast PWM (Top=OCRA0)
```

We consult the data-sheet again for the TCCR0A register and see that we need to set the two lowest bits to one.

```
TCCR0A [COM0A1,COM0A0,COM0B1,COM0B1,---,---,W
```

Next we need to instruct the timer/counter on how we want our output. We consult the data-sheet and see that to start with output on, then to shut off on a compare-match we need to set COM01 to one and COM00 to zero.

```
COM01 COM00
  0     0   - Output Disabled
  0     1   - Output Disabled
  1     0   - Clear Output on a Match
  1     1   - Set OCRA on Match (Inverted Out
```

The two bits are also found in the TCCR0A register:

```
TCCR0A: [COM0A1,COM0A0,COM0B1,COM0B1,---,---,
```

So to get Fast PWM Mode (non-inverted) we need to set bits COM0A1, WGM01 & WGM00

```
    LDI A,0b1000_0011    ;PWM MODE 3
    OUT TCCR0A,A
```

Since we want to be able to see our output, we need to slow the timer/counter way down. We consult the data sheet and find the largest pre-scaler/divider available is 1024:

```
CS02 CS01 CS00
  0    0    0   - Timer Stopped
```

```
0    0    1   - No Pre-Scaler
0    1    0   - Divide by 8
0    1    1   - Divide by 64
1    0    0   - Divide by 256
1    0    1   - Divide by 1024
1    1    0   - External Clock Falling Edge
1    1    1   - External Clock Rising Edge

TCCR0B: [FOC0A,FOC0B,---,---,WGM02,CS02,CS01,
```

To select a pre-scaler/divider of 1024 we need to set the CS02 & CS00 bits to one:

```
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
```

The Attiny13 is shipped with a 9.6MHz internal oscillator selected and a pre-scaler of 8. This means the chip runs at 1.2Mhz (9.6Mhz/8). If we further select a pre-scaler/divider of 1024 that gives us a base frequency of about 1.2Khz (1.2Mhz / 1024), and since our counter is going from zero to 255 we divide 1.2Khz by 256 and we are left with 4.6Hz, so our LED should flash about 5 times per second, easy for the human eye to see. The exact theoretical frequency would be:

```
FREQUENCY =  (((9,600,000 / 8) / 1024 ) / 25
```

To cycle through all the different values for our compare register we are going to increment it each time we get a match. To do this we need to use the timer-compare interrupt. This interrupt will be called every time the timer/counter equals the value in our compare register. We tell the system where our interrupt is with the following commands:

```
.ORG $0006               ;COMPARE MATCH VECTO
    RJMP TIM0_COMPA
```

In our interrupt routine we will read the current value of the compare register (OCR0A) increment it by one and write the result back out.

```
TIM0_COMPA:
    IN   A,OCR0A
    INC  A
    OUT  OCR0A,A
     RETI
```

We consult the data-sheet an find that we need to set the OCIE0A bit of the Timer Mask Register (TIMSK) to one to activate a compare-match interrupt:

```
TIMSK0: [---,---,---,---,OCIE0B,OCIE0A,TOIE0,
```

We see that the Output Compare Interrupt Enable (OCIE0A) is the 3rd bit so we set it with:

```
    LDI A,0b0000_0100    ;ENABLE COMPARE INTE
    OUT TIMSK0,A
```

Then we give the system permission to run interrupts, so we enable them globally with the command SEI.

This is what our finished program looks like:

```
.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000               ;STARTUP VECTOR
    RJMP ON_RESET
.ORG $0006               ;COMPARE MATCH VECTO
    RJMP TIM0_COMPA

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b1000_0011    ;PWM MODE 3
    OUT TCCR0A,A
    LDI A,0b0000_0101    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,0b0000_0100    ;ENABLE COMPARE INTE
    OUT TIMSK0,A
     SEI                 ;ENABLE INTERUPTS GL

MAIN_LOOP: RJMP MAIN_LOOP;A DO-NOTHING LOOP

TIM0_COMPA:
    IN   A,OCR0A
        INC  A
        OUT  OCR0A,A
         RETI
```

If we make a few small changes we can see one of the many applications of PWM, to dim an LED.

First we select a lower pre-scaler/divider to increase the frequency. Here we set it to divide by 64 instead of 1024:

```
    LDI A,0b0000_0011    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
```

What would be the frequency of the our PWM wave be? It is our system clock divided by the pre-scaler of 64, divided by 256:

```
FREQUENCY = ((1.2Mhz / 64) / 256) = 73 Hz
```

Since we are going to slowly dim our LED, we need to decrement the compare register instead if incrementing it.

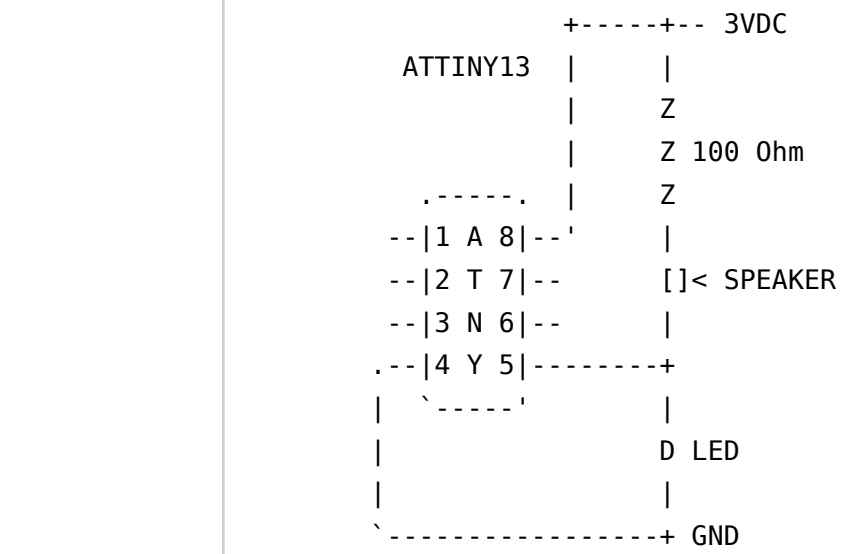
```
   IN   A,OCR0A
   DEC  A
   OUT  OCR0A,A
```

The second version of our final program looks like this:

```
.INCLUDE "TN13DEF.INC"   ;ATTINY13 DEFINITION
.DEF A = R16             ;GENERAL PURPOSE ACC

.ORG $0000               ;STARTUP VECTOR
    RJMP ON_RESET
.ORG $0006               ;COMPARE MATCH VECTO
    RJMP TIM0_COMPA

ON_RESET:
    SBI DDRB,0           ;SET PORTB0 FOR OUTP
    LDI A,0b1000_0011    ;PWM MODE 3
    OUT TCCR0A,A
    LDI A,0b0000_0011    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
    LDI A,0b0000_0100    ;ENABLE COMPARE INTE
    OUT TIMSK0,A
     SEI                 ;ENABLE INTERUPTS GL

MAIN_LOOP: RJMP MAIN_LOOP;A DO-NOTHING LOOP
```

```
TIM0_COMPA:
    IN   A,OCR0A
    DEC  A
    OUT  OCR0A,A
     RETI
```

Let us hook-up a speaker to our circuit so we can hear and see our results. Hook a small 8 Ohm computer speaker to our output pin. The other end of the speaker to our positive 3 volts through a 100 Ohm speaker:



As a final exercise let us increase the frequency and observe the results. Lets change the pre-scaler/divider from 64 to 8 by making the following change to our program:

```
    LDI A,0b0000_0010    ;SET PRESCALER/DIVID
    OUT TCCR0B,A
```

What is the base frequency of our project now?

```
FREQUENCY = ((CLOCK-SPEED/PRE-SCALER)/256)
          = ((1,200,000/8)/256
          = 150,000/256
          = 586 Hz
```

Subpages (1): 3. TIMER COUNTERS and PWM

Comments

You do not have permission to add comments.