µC eXperiment

Examination of the Arduino millis() Function

8 . 87 1 0

First, some background information. The typical Arduino has a 16MHz oscillator. A 16MHz oscillator results in the microcontroller having a clock cycle once every 16-millionth of a second. A clock cycle is roughly the time it takes for one instruction cycle (there are exceptions). Therefore, on the Arduino, each clock cycle is 1/16,000,000 of a second, which is:

- 0.0000000625 seconds
- 0.0000625 milliseconds (ms)
- 0.0625 microseconds (µs)

The Atmel ATmega168/328 based Arduino has 3 timers, of which the millis function uses the microcontroller's Timer #0. Before utilizing any timer, several registers must be set. If we examine how the Arduino sets up Timer #0, we find it sets a prescale factor of 64 (from the wiring.c file):

ATMEL ATMega328 Datasheet page 111 specifies the Timer #0 prescale bits:

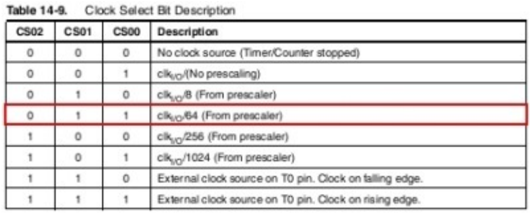


Table 14-9. Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/(No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

While technically a prescaler divides the timer, I like to think of it as a multiplier. For example, with an Arduino clock cycle occurring every 1/16,000,000 of a second, applying a 64 prescale means Timer #0 is ticking at 64 times the base oscillator rate, or 64*1/16,000,000, which is every:

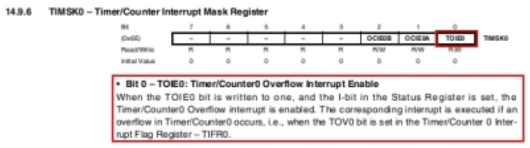
- 0.000004 seconds

- 0.004 ms
- 4 µs

Timer #0 has an 8-bit counter register (TCNT0) which is incremented by 1 every 0.004 milliseconds. The maximum 8-bit value the timer can hold is 255 (hexadecimal 0xFF). Therefore, when the timer attempts to count to 256, it "rolls over" to 0. This "roll over" is called an "overflow" in microcontroller parlance. Take note the counter ticks off 256 times, not 255 times (because 0 + 255 = 256).

The Timer #0 overflow can trigger an . When an interrupt occurs, the Arduino halts execution of the running program and then calls the specific interrupt subroutine. This subroutine is typically called an "interrupt handler." The interrupt handler that we are interested in is called the . The Arduino code enables the Timer #0 overflow interrupt (again, found inside the wiring.c file):

ATMEL ATMega328 Datasheet page 112 specifies the Timer #0 interrupt enable bit:



14.9.6 TIMSK0 – Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0	
(0x6E)	–	–	–	–	–	OCIE0B	OCIE0A	TOIE0	TIMSK0
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 0 – TOIE0: Timer/Counter0 Overflow Interrupt Enable**

When the TOIE0 bit is written to one, and the I-bit in the Status Register is set, the Timer/Counter0 Overflow interrupt is enabled. The corresponding interrupt is executed if an overflow in Timer/Counter0 occurs, i.e., when the TOV0 bit is set in the Timer/Counter 0 Interrupt Flag Register – TIFR0.

Therefore a Timer #0 "overflow interrupt" occurs each time the timer's counter (TCNT0) rolls over. Taking the above math into account, this occurs every 1/16,000,000(oscillator) * 64(prescale) * 256(roll over) = 0.001024 seconds, or every 1.024 ms. It's important to realize the interrupt doesn't occur exactly each millisecond.

Next we need to examine the Timer #0 interrupt handler, which is made slightly more complicated because of the decimal part of the 1.024ms roll-over period. But first, let's do some math so we can substitute these numbers for the following MACROs (these macros are embedded inside several Arduino hardware files):

F_CPU is the oscillator frequency, and is defined during sketch compilation. We already know this is 16MHz or 16,000,000, which makes:

Here is the Timer #0 overflow interrupt handler with the above substitutions made and a little cleanup just to make the code easier to understand (the original code is listed at the bottom of the post):

Therefore, this overflow handler increments the value of timer0_millis every 1.024ms and then adds another increment to timer0_millis (catches up, if you will) every time timer0_fract is greater than 125. Hence, timer0_millis accounts for the missing 0.024ms from every “timer overflow” at intervals of 125/3 = 41.67ms. Which means timer0_millis accumulates an error of 0.024ms each time it executes (every overflow), until the error approaches 1ms. At which time, timer0_millis jumps by 2 and corrects itself.

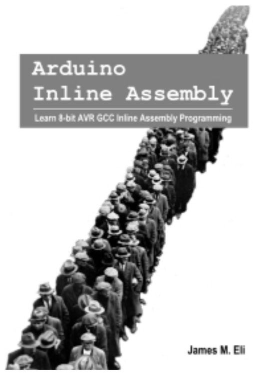
To assist in understanding how this works, the following table demonstrates the internal values of the overflow handler during each iteration:

Finally, the function merely returns the value of timer0_millis (again, I cleaned up the function to make it easier to understand, the actual function is listed below):

Next we'll look at the micros() function.

If you are concerned about millis overflow when using millis to time intervals, please see this.

Do you wish you could read and write inline assembly code for the Arduino? Check out the book with greatly expanded coverage!



[click on the image]

Cascading Timers to Create a Long Delay
In "arduino"

Examination of the Arduino micros() Function
In "arduino"

Arduino Inline Assembly Port & Pin Compendium
In "arduino"

J
µC experimenter
View all posts by Jim Eli →

12 . 7 B 98 . 7 7 . 8 C 7 . 780 0 3 4 B 5 0 7 . 9 0 6 1 0 6 2 0 8 . 3 0 6 . 0 6 . 8 8 4 6 4 1 . 9 6 5 4

26 Responses to *F)*

7 40

. . 6 . 0 0 6

Hello everyone,
I read on arduino.cc/en/reference/micros that I can know the number of microseconds since the Arduino began running the program.
On my 16 MHz Arduino Uno board, the function has a resolution of four microseconds and less with a 8 MHz board. I found the arduino Due (arduino.cc/en/Main/ArduinoBoardDue) and it has a 84 MHz clock speed.

Could I have a better resolution than 4 µs ? I would like to have a precision with nanos seconds.

. 9 5

9 5 0 0 9 6

you didn't mention one important aspect: how many clock cycles it takes?
i am calling micros() from the external interrupt handler and trying to calculate how many ticks the whole interrupt would require in the worst scenario. any ideas?

. 9 5

J

9 5 0 0 6

Convert your code to run on a generic AVR uC then run it inside the simulator in AVR Studio. The AVR Studio simulator has provisions that easily allow you to time sections of your code in either uSec and/or clock cycles.

. 9 5

9 5 0 0 6

good hint probably, but I am using MacOS, so dealing with objcopy and simulavr…

J

9 5 0 0 6

Running in my simulator micros() round trip is 49 clock cycles/3.06uS @ 16MHz.

8 . 0 0 6

The logic of how it calculates times is all fine. Could you tell us how it calculates the time since the program began running? Does it use the WDT? Also according to the Arduino reference page, millis() holds duration of upto 50 DAYS!!! could you please tell us how many registers it uses to store such large durations? What all registers should I involve if I want to port this function to other AVR chips?

. 95

J

8 . 0 0 6

1.) millis runs off the AVR timer #1. The timer is a function of the AVR clock, which in the case of the Arduino is typically running at 16MHz. When the Arduino starts, it takes a few moments for this system clock to stabilize, then the initialization code completes and millis starts counting. It does not use the WDT. To actually time this interval would be difficult.

2.) millis is not stored in a register, it is stored in 4 bytes (32 bits) of memory. 4 bytes can hold a number as large as:
(2^32-1) = 4294967295
A value this large will roll over to zero every:
4294967295 / 1000 / 60 / 60 / 24 = 49.71 days

3.) It would be easy to port millis or a function similar to another AVR chip since the timers in the AVR family are similar. You would need to read the datasheet for the new chip to account for differences (8 vs. 16 bit timer, names of timer registers, etc.). Good luck.

. 95

8 . 0 0 6

your are awesome! thanks!

J

8 . 0 0 96

Checkout my post here to see an example of how to implement millis on a ATtiny24 AVR chip.

J

1 0 0 96

Why this complicated stupid way to compensate error each cca 40ms ? We just can use TIMER0_COMPA_vect and set OCR0A to 250. Instead of TIMER0_OVF_vect. Timer in CTC mode (TCCR0A). So then counter counts not to 255 and overflow, but counts to 250 and reset itself to zero and again counts to 250. And 0.004ms times 250 is exactly ONE milisecond. 🙂

. 95

7 B 0 0 96

I just started looking at Timer0 to see if I could hook onto it for an ADC function I need.
I have the SAME question. WHY are they doing 1 msec timing the HARD/SLOW way?
With COMPA0 reset the only overhead is the ISR in/out and the increment of timer0_millis.
What is the rationale for having an accumulating error, when one can instead create a timer interrupt/value that is EXACTLY 1 msec?
And there is also the option of just adding 50 to TCNT0 every interrupt if you want to use TOV0 instead (to trigger A/D in my case). It messes up micro() a bit but one can correct this post measurement by looking at the values and adjusting (e.g. any micro value between 0-50 needs 50 usec subtracted from the difference calculation.

. 95

J

7 B 0 0 96

Arduino was originally derived from the Wiring project created by Hernando Barragan (hence the "wiring" based file names used for the Arduino core code). Wiring was developed around 2003 and evolved over a series of disparate microcontrollers. Therefore, the millis function is an artifact of a scheme designed to run at multiple (mostly slower) frequencies, which admittedly is not the most efficient method for the current hardware.

1 0 0 96

I had the same idea, but OCR0A shhould actually be set to 249, as the timer starts from 0 🙂

. 95B

1 0 0 96

Arduino is intentionally doing it this way in order to keep the [PWM on pins 5 and 6 (controlled by Timer0)]() with a 256-step resolution (0 to 255). So..that requires a bit of trickery in millis().
~http://www.ElectricRCAircraftGuy.com

. 95B

J

1 0 0 96

For example this is section of code for ATtiny13 copied from eclipse from my project:

```
#include <util/atomic.h>

unsigned int millis()           // millis function
{
        unsigned int millis_return;
        // Ensure this cannot be disrupted
        ATOMIC_BLOCK(ATOMIC_FORCEON)millis_return = timer0_millis;
        return millis_return;
}

ISR (TIM0_COMPA_vect)           // timer0 interrupt service routine
{
        timer0_millis++;
}

int main(void) {
// this is setup
        CLKPR = 128;            // Clock Prescaler Change Enable
        CLKPR = 6;              // 4.8MHz/64 = 75kHz
        TCCR0A = 2;             // CTC timer mode
        TCCR0B = 1;             // timer prescaling 1
        OCR0A = 75;             // 75kHz, period = 13.333us, 1ms = 13.3333us * 75
        sbi(TIMSK0, OCIE0A);    // enable timer interrupt
while (1)
{
// this is loop
}
}
```

It will be slightly different in arduino code and also for ATMega328P. But same point.
For arduino it will be better just to edit source files for millis function and not creating your own named differently.

. 95B

J

1 0 0 96

I forgot this:

```
//define function cbi – Clear Bit in I/O Register
//define function sbi – Set Bit in I/O Register
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~(1<<bit))
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= 1<<bit)
```

_ . 95B

J

1 0 0 96

Jerry,
I think your approach is not possible within arduino environment, because arduino also uses the output compare register for analogWrite function for PWM pins. And for this the OCRxx is used for holding the PWM values. I am sure the designers of arduino are not stupid and they have managed to compromise here and there a little bit in order to get all the great features of an arduino.
Regards,
Jo

_ . 95B

---

1 0 0 96

Thanks! This really helped.
Just because I like to check things out, I verified millis() and micros() really produced the same result. Here's the code:

```
/*
Gabriel Staples
http://electricrcaircraftguy.blogspot.com/
11 March 2015
*/
#include //<–see this file located here! C:\Program Files (x86)\Arduino\hardware\arduino\avr\cores\arduino
void setup() {
// put your setup code here, to run once:
Serial.begin(115200);
Serial.println(MICROSECONDS_PER_TIMER0_OVERFLOW);
Serial.println(MILLIS_INC);
Serial.println(FRACT_INC);
Serial.println(FRACT_MAX);
}
void loop() {
// put your main code here, to run repeatedly:
Serial.println();
static unsigned long t_ms = millis();
static unsigned long t_us = micros();
delay(5000);
unsigned long dt_ms = millis() – t_ms;
unsigned long dt_us = micros() – t_us;
Serial.print("dt_ms; Serial.println(dt_ms);
Serial.print("dt_us; Serial.println(dt_us);
}
```

_ . 95B

1 0 0 96

And the output:

```
1024
1
3
125
```

```
dt_ms = 5000
dt_us = 5000012
dt_ms = 10001
dt_us = 10000840
dt_ms = 15001
dt_us = 15001764
dt_ms = 20002
dt_us = 20002696
dt_ms = 25004
dt_us = 25003628
dt_ms = 30004
dt_us = 30004560
```

. 95B

. B 0 0 96

Thank you very much for the nice explanation!

. 95B

8 . 0 0 96

where did you define timer0_overflow_count ????

. 95B

J

8 . 0 0 96

Slow down there QuestionMarkBoy. An interrupt routine is defined with ISR(). This macro registers and marks the routine as an interrupt handler. SIGNAL() is just the old way of doing it, and ISR is the new way. See the AVR GCC documentation for

. 95B

. 9 . 6 . 0 0 96

Hi Jim!

This is very nice article!

In your “Prescaler” section, did you forget to add “sei()” before “sbi(TCCR0B, CS01)”?

. 95B

J

. 9 . 6 . 0 0 96

George, You’re right, sei() occurs before the two sbi calls. But I didn’t include I thought it wasn’t pertinent to the discussion.

. 95B

1 0 0 6

This website is just gret. I’ve search these informations a great deal and I view
it that is good written, easy to understand. I congratulate you because of this article that
I’ll recommend to people friends. I request you to recommend the gpa-calculator.co site where each college student or pupil can calculate ratings gpa
marks. Thank you!

. 95B