Accurate measurement of code execution time

Z. Gingl, Department of Technical Informatics, University of Szeged

It can be important in an Arduino UNO project, how much time is needed to execute some lines of code or a function. If you can measure the execution time accurately, you can optimize your code, speed up loops and you can be sure that the code, function or interrupt service routine can complete its job in time. Or you can just learn more about the code execution properties, compiler performance and you can see what you can expect from library functions.

The typical way to measure time is to use the millis() or micros() functions [1]. However, they have limited resolution and accuracy, much worse than the resolution of the system clock. Is it possible to determine reliably, how many system clocks periods are needed to execute some source code lines or even a single instruction? Fortunately, the answer is yes and in addition, it is quite simple.

The 16-bit Timer/counter1 of the UNO's Atmega328 microcontroller can be clocked by the system clock [2]. So, every increment of the counter takes 62.5 ns, the time of a single cycle at 16 MHz. Therefore, if we start the timer at the beginning and stop at the end of a code fragment, the counter value will contain the number of system clock periods needed to execute the code plus the overhead of the timer handling. If this latter is known, we can determine the code execution time at the highest possible resolution, 62.5 ns. Let's see the following code example:

```
TCNT1 = 0;       // clear the counter value
TCCR1B = 1;      // start timer1 using the system clock
digitalRead(9);  // execute some code
TCCR1B = 0;      // stop timer1
```

Figure 1 shows the time diagram of the code execution and related timer operation. At the end the counter value is N+2, so the number of system clock cycles required to execute the code is N=TCNT1-2. The 16-bit register TCNT1 contains the counter value.

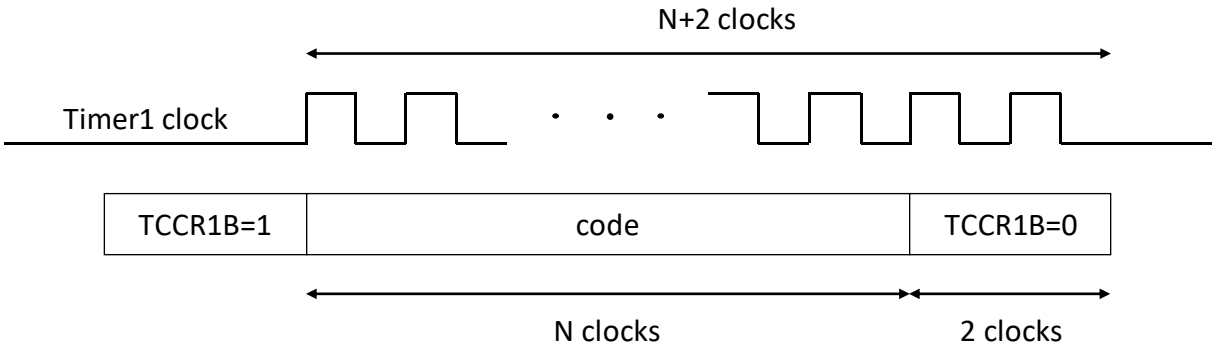

Figure 1. Time diagram of the code execution and timer operation.

The Timer/counter1 needs some simple additional setup instructions and it is more convenient to use more readable macros or functions like this:

```
clearAndStartTimer();
digitalRead(9);
stopTimer();
Serial.print(getTimer());
```

This study was funded by the Content Pedagogy Research Program of the Hungarian Academy of Sciences.

If the functions are forced to be inline functions [3] by adding function prototypes, then no function call and return overheads will be added:

```
inline void clearAndStartTimer(void) __attribute__((always_inline));
inline void stopTimer(void) __attribute__((always_inline));
```

The functions doing the job are the following:

```
inline void clearAndStartTimer(void)
{
  TCCR1A = 0;     // set timer1 mode: normal
  TCCR1B = 0;     // disable timer1 clock to be sure it is not running
  TCNT1  = 0;     // clear timer1 value
  TCCR1B = 1;     // start timer1 with input clock 16MHz
}

inline void stopTimer(void)
{
  TCCR1B = 0;     // stop timer1
}

uint16_t getTimer(void)
{
  return TCNT1 - 2; // remove overhead clocks
}
```

If you don't use inline functions, then more and possibly uncertain overhead will be experienced. In other hand, it is quite useful to learn and use inline functions in an Arduino application.

You can also use macros to keep the code more compact, and to be a bit more straightforward (even though macros are not recommended in general):

```
#define CLEAR_AND_START_TIMER {TCCR1A=0; TCCR1B = 0; TCNT1 = 0; TCCR1B = 1;}
#define STOP_TIMER {TCCR1B = 0;}
```

Note, that reading the TCNT1 value by getTimer() is not time critical, since the timer is not running during reading its final value. Therefore, it is not needed to use an inline function or a macro.

You can find example codes showing that even a single cycle 'nop' instruction execution time can be reliably measured.

Note that in an Arduino application another timer is running in the background by default and generates interrupts periodically (at about every millisecond) to update the system time. If this happens during the code execution to be inspected, an additional time might be observed. You can avoid this by disabling the interrupts temporarily using noInterrupts() or cli() and re-enable them by calling interrupts() or sei() [1]. This may affect the system time of course. Note also that some rare Arduino libraries (e.g. the Servo library) may use Timer/counter1.

The maximum time what can be measured is 2^{16} system clock periods, a bit more than 4 ms at 16 MHz system clock frequency. Longer time can be measured in the usual way, by using micros() or millis(), they provide enough accuracy in such cases. Timer overflow interrupts can also be used to count the number of overflows and to count more than 2^{16} system clock periods, but the code will be more complex and the accuracy won't be really better. Keep in mind that the system clock accuracy is also limited. You may find some other methods too with various pros and cons.

References

1. Arduino function reference, https://www.arduino.cc/reference/en/#functions
2. ATmega328P datasheet, https://www.microchip.com/wwwproducts/en/ATmega328P
3. Inline functions, GCC compiler, https://gcc.gnu.org/onlinedocs/gcc/Inline.html