Behavior Research Methods, Instruments, & Computers
1988, 20 (6), 566-572

TIMEX: A simple IBM AT C language timer with extended resolution

PHILLIP L. EMERSON
Cleveland State University, Cleveland, Ohio

The simple timing method of reading the IBM-BIOS tick counter is extended from 55-msec to 54-μsec resolution by reading the 8254 hardware counter. Subroutine timex( ) implements the timer in Borland Turbo C on an IBM AT and on a PC or XT if the 8253 timer chip is replaced with an 8254 chip. A more limited version operates on an unmodified PC or XT.

The C language (Kernighan & Ritchie, 1988) is becoming popular for low-level programming tasks associated with real time laboratory experiments conducted on small computers. Although the C language itself contains no primitive commands for input and output, most popular implementations for MS-DOS and the IBM PC, XT, and AT include large libraries of routines that make it easy to do low-level input and output.

The timer presented here could probably be programmed in BASIC, except that two Borland Turbo C routines are needed for good reliability. These are disable( ) and enable( ), which disable and enable the 8086/88 hardware interrupts. Otherwise, the only special routines used are inp( ), outp( ), and peek( ), which are essentially the same as those with similar names in BASIC. However, standard versions of BASIC execute too slowly for intensive real-time programming.

This timer, implemented as the timex( ) routine shown in Listing 1, shares most of the advantages and disadvantages of the simpler clock( ) routine (Datalight C Library; Emerson, in press). The main difference is that timex( ) gives about 54-μsec resolution, whereas clock( ) gives 55-msec resolution on IBM small computers and clones. Another difference is that timex( ) can be used only on the IBM AT or a closely compatible clone, whereas clock( ) works properly on other machines that use MS-DOS, whether IBM compatible or not. However, timex( ) will also work properly on an IBM PC or XT if the Intel 8253 timer-counter chip is replaced by an 8254, and a more limited version is presented that operates with the 8253 on an unmodified PC or XT. A minor difference is the convenience of the time unit, which is .01 sec with clock( ) but 53.6381 μsec with timex( ). With timex( ), a user would probably convert to some convenient standard unit for data analysis using floating point arithmetic.

Both timex( ) and clock( ) have three main limitations, but two could be overcome by simple additional programming if it seemed worthwhile. The first is that there is no direct way to initialize a beginning time count to zero, so a time interval must be measured by taking the difference between readings made at the beginning and the end of the interval. The second is that a time interval straddling midnight will not be measured correctly, because the operating system reinitializes the counters to zero every 24 h. The third limitation is the assumption that concurrently running user programs do nothing to interfere with the normal operation of the time-of-day clock, such as locking out interrupts for unusually long periods of time. A corollary to this is that the user must not change the counting rate of the 8254 hardware counter that drives the time-of-day clock, and must not alter the standard interrupt priorities as programmed by the BIOS at bootup.

Offsetting these limitations are several features of both clock( ) and timex( ) that are advantageous for most purposes. These routines do not interfere with the system time-of-day clock in any way, and there is no need to restore any system information at the termination of use. They are driven by the system time-of-day interrupts, which are of the highest priority in the standard interrupt configuration, so keyboard, disk, serial, and printer operations do not affect the timing performance. For the same reason, these timers do not consume any additional interrupt overhead time of their own.

How to Use timex( )

The timex( ) routine returns a long (32-bit) integer, which is the elapsed time since midnight in units of 53.6381 μsec (64/1.1931817). This implied precision may be greater than that actually achievable among various ATs whose system clock oscillators may not be calibrated to this accuracy. However, it is a useful figure based on the nominal 1.1931817-MHz clock signal specified by IBM. It has become common in the context of timing on IBM small computers to refer to the interval between updates of the software counter used by the time-of-day clock as a "tick," so 1 tick = 54.9254 msec. Because "tick" already has a meaning, "tock" will be used in this paper to refer to timex( )'s elementary time unit of 53.6381 μsec. Thus, 1 tick = 1,024 tocks. At this counting rate (about 18,643 tocks/second), the signed 32-bit integer would wrap to a negative value in 32 h, but the midnight reset

The main ideas of this timing method were suggested to the author by Howard L. Kaplan in unpublished communications. The author's address is Department of Psychology, Cleveland State University, Cleveland, OH 44115.

by the operating system prevents measurement of durations longer than 24 h.

An elementary test program is shown in Listing 2. It merely measures the times between successive presses of the ENTER key on the keyboard, while making a test for timing errors on each cycle. If a negative time value is returned, the indication is that timex( ) is not operating properly. That could happen, for example, if the user program modified the standard bootup interrupt configuration. It is important in a calling program to declare the variables ztime and etime, or other variables serving the same purposes, to be of type long. If the intervals to be measured all are less than 3.5 sec, the obtained differences between the longs can be cast to type unsigned int for more compact 16-bit storage. The timex.c module of Listing 1 can be #included with a calling program, as is done with the test program of Listing 2, or it can be compiled separately and linked in with the calling program.

It is possible to use timex( ) to measure time intervals as short as 1 msec, but its own execution time is a nontrivial fraction of 1 msec and there are only about 18.6 tocks/millisecond. On an AT, samples of execution times of timex( ) had means near 123 μsec and standard deviations of about .34 μsec. Accuracy tests of timex( ) in the measurement of time intervals produced by the serial interface timer (Emerson, 1988b) gave standard deviations near 27 μsec, which is near the theoretical limit of 22 μsec (1 tock/$\sqrt{6}$). This theoretical limit assumes uniform and independent distributions of the starting and ending positions within the 1-tock interval of uncertainty.

How timex( ) Works

It is well known (Sargent & Shoemaker, 1986) that the IBM BIOS uses a pair of 16-bit memory registers to maintain a count of clock ticks that provides the basis for the system time-of-day clock. This tick count is incremented 18.2 times/second (1193818.7/65536) by interrupts from counter 0 of an Intel 8254 (or 8253) module. Tick count information is available to the user via several different BIOS and DOS calls, or by direct access to memory locations 6ch and 6eh of segment 40h, and these facilities have provided the basis for simple timing methods; however, the elementary tick unit is too crude for some purposes. For an extra 10 bits of resolution, it is possible to read the fine-grained 8254 hardware counter and combine that reading with the BIOS tick count. It would be possible to use all 16 bits of the hardware count, but the resulting count would wrap to a negative value in about a half hour, using a signed 32-bit long variable for the counter. On the other hand, there is no point in using fewer than 10 bits, because the wrap to a negative value does not occur in 24 h.

To get the additional 10 bits of resolution, timex( ) begins by reading the tick counter twice, with READ_BACK and read commands to the 8254 counter sandwiched between. The READ_BACK command instantly causes a latching of 3 bytes of information from the 8254 into buffer registers that can be examined later. The first of these is a status byte, and only its highest order bit is used by timex( ). The other two are the contents of the low and high bytes of the 8254 16-bit counter. The READ_BACK sequence has no effect at all on the running counter or on the interrupts that drive the BIOS tick counter.

After the two readings of the 18.2-Hz tick registers, with the intervening READ_BACK sequence, the high and low words of the tick readings are joined to form long integers: ticks1 for the first reading and ticks2 for the second. The two different tick readings are necessary to determine whether or not the tick count changed near the time when the READ_BACK latch operation occurred. The simplest case is no change. In any case, the following operations are performed on the 10-bit number that is constructed from the 8254 latch buffer: This number is subtracted from 1024 to convert the down count to an up count, and the resulting difference is divided by 2. Then the high-order bit of the status byte is tested to see whether the constant 512 should be added into the result of the other operations.

The reason for the division by 2 and the conditional addition of 512 is that 8254 counter 0 counts down twice by 2s from 65536 during each tick period between interrupts. The state of the high-order status bit indicates which of these two countdowns was in progress when the latch operation occurred. This high bit is referred to as "out" in the 8254 data sheets, and it indicates the logic level on the electronic output pin of the 8254 chip. The 8254 counter 0 is programmed by the BIOS to operate in Mode 3, which means that the output signal is a periodic square wave. The output signal triggers the interrupt that increments the BIOS tick counter, but only on the rising edge, whereas the 8254 counter counts down once while the output is high and once again while it is low, during the 50% duty cycle.

With these facts in mind, it is not difficult to understand the remainder of the logic in Listing 1. If the two BIOS tick counts differ by one unit, a second-half countdown by the 8254 must have been completed between the two times when the interrupts were first disabled. Then, the question is whether the READ_BACK latch operation occurred before or after the countdown was completed. If before, then the first tick reading is the base to which the modified 8254 count should be added. If after, then the second of the two tick readings is the correct choice. The question is answered effectively by determining whether the 8254 count was latched in the first or second half of the 55-msec cycle, and that information is given by the high-order status bit. If the latch occurred in the first half, then it occurred after the termination of the second-half countdown. If in the second half, then it occurred before. In either case, it must have occurred very near the end of the countdown, because the two tick readings are obtained in a very short time relative to the 55-msec period, so the decision is quite definite unless the assumptions stated earlier are violated.

If the two tick counts differ by more than one unit, then more than one 8254 interrupt must have occurred, and that should not happen because these interrupts occur only

once every 55 msec, and the two tick readings are made within less than 1 msec of each other unless the user has disturbed the normal interrupt configuration, written to the 8254, or is running a monstrous piece of software that locks out interrupts for unusually long periods of time.

Clearly, the 8254 high-order status bit is crucial for the proper identification of the half of the double countdown that contained the READ_BACK latch operation. The ability to read the status is a feature of the 8254, which is standard on the AT, but not of the 8253 (or 8253-5), which is standard on the PC and XT. At the cost of a further limitation, it is possible to obtain this crucial bit of information when using the 8253, but some users might prefer the better performance obtained by buying an 8254 to replace the 8253 on a PC or XT. The 8254 is upward compatible with the 8253, in that the 8254 recognizes all the 8253 programming commands. An 8254 can be purchased from mail-order distributors such as B.G. Micro (P.O. Box 280298, Dallas, TX 75288) for less than $10. However, the 8253 is likely to be soldered to the motherboard, as it was on an available IBM XT that was used for testing timex( ). Some expert technical help was needed to remove the 8253 and install a socket for the 8254, which took about an hour, not including the time to remove and reinstall the motherboard. Measured execution times of timex( ) on the XT with the 8254 had means near 540 μsec and standard deviations near 1 μsec. Accuracy tests gave essentially the same results as with the AT.

Using the 8253 on a PC or XT

For an unmodified PC or XT with the standard 8253, a different version of this timer, timex3( ), is used (see Listing 3). The additional limitation of this version is that it waits for a period that can be as long as 28 msec after reading the hardware counter, in order to determine whether that reading was made during the first or the second of the two countdowns of the 55-msec tick period. The resolution is still strictly 1 tock, but the execution time is distributed unpredictably between about .5 and 28 msec except under special circumstances. For some purposes, this results in the loss of much of the added resolution, but for others it does not.

For many applications, the full 1-tock resolution is achieved by timex3( ) if the intervals to be measured all are appreciably longer than 28 msec. This is true in the measurement of time intervals between external events, and of an interval that is initiated by the computer and ended by an external event such as a keypress in a reaction time experiment. In these cases, the intervals to be measured must be longer than half a tick and the measurement of a second interval cannot begin until at least half a tick after the end of the previous interval.

A more problematic situation is that in which the computer is to produce a time interval, such as turning a tone on for 200 msec and then off again. The beginning of the time interval can be ascertained accurately enough by turning the tone on and then calling timex3( ). For the end, however, the timer must be called repeatedly in a program loop until the count reaches or exceeds the prescribed value. Thus, a spurious delay of up to half a tick in duration is added into the time interval.

With qualifications, a solution to the problem of producing time intervals is as follows: The technique is to (1) call timex3( ) once, (2) turn on the stimulus signal, (3) call timex3( ) *n* times in a for( ) loop, and (4) turn off the stimulus signal. The timex3( ) readings need not be recorded with this technique, except possibly for test purposes. Under these circumstances, all delays except the first are exactly half a tick long, and the stimulus is produced for a duration of *n* half ticks with tock accuracy. However, the durations that can be produced are restricted to integral multiples of half a tick, and there is half a tick of uncertainty about the clock time of the stimulus onset. A redeeming feature of this technique is that it can be compounded to produce a whole train of intervals, possibly unequal, because no time is lost by interspersed brief operations, such as turning a signal on or off or initiating a new for( ) loop. Unless such an interspersed operation requires nearly half a tick to execute, the 8253 hardware counter maintains the timing continuity, and the time steps remain synchronized to the half-tick increments. Likewise, occasional brief hardware interrupts, such as those serving keyboard and disk operations, should not affect the intervals produced.

If, immediately at the end of a train of intervals produced by the above method, the time until the occurrence of an external event is to be measured, the initial time reading should be taken as the returned value of timex3( ) called after the produced interval. The final reading is made when the external event occurs, and the latency of the external event is given with tock accuracy by the difference between the final and initial readings. However, that final reading most likely is not in synchrony with the half-tick periods established in the production of the preceding train of intervals. Thus, if a following train is to be produced, it, too, must be initiated with the half tick of uncertainty at the beginning. For many kinds of reaction time experiments, it should be possible to relegate that half tick of uncertainty to the intertrial interval, where it would be of little consequence.

How timex3( ) Works

The workings of timex3( ) are similar to those of timex( ). The differences are caused by the fact that the latching operation preserves only the 8253 count, and not the out status. Therefore, the out status at the time of latching must be inferred. The general tactic is to make the initial tick and hardware 8253 readings and then sit in a program loop until the termination of the current 8253 hardware countdown to zero. Then another tick reading is made and compared with the initial one to see if this countdown resulted in an increment of the tick count. If so, the implication is that this countdown was the second of the two that are performed during each tick period, and that status = 0. If not, then status ≠ 0.

However, there are some complications to this scheme due to the fact that the out status that is inferred by this simple method is not necessarily that that existed precisely at the instant that the hardware counter contents were latched. This is not a problem with the 8254 because the status and counter contents are latched simultaneously. To resolve this ambiguity with the 8253, timex3( ) makes a sequence of three hardware counter readings alternating with three BIOS tick readings, and includes a moderately complicated if-else structure as shown in Listing 3.

Let h_1, h_2, and h_3 be the successive readings of the 8253 hardware counter, and t_1, t_2, and t_3 be the readings of the tick counter. The sequence of events is in the order [h_1, t_1, h_2, t_2, h_3, wait for countdown, t_3]. The h_1-h_3 subsequence is executed in a fraction of a millisecond, but t_3 may be delayed by as much as half a tick period. In some cases, the delay and the t_3 reading would not be necessary to determine the out status, but they are always included by timex3( ) because the interval-production technique described earlier would not work properly otherwise.

Analysis and verification of the if-else status logic in Listing 3 are aided by the use of timing diagrams, which can be constructed for the various cases that can arise. For such analysis, note these points: (1) h_2 is the 10-bit down count that later is changed to an up count and added to one of the tick readings, (2) h_1 and h_3 are used only to resolve the status ambiguity, (3) there can be no more than one terminal countdown in the interval from h_1 to h_3, and (4) there is exactly one terminal countdown between h_3 and t_3.

Conclusion

The timex( ) routine was designed to be as flexible as possible within the limitations that have been noted. For example, no DOS or BIOS calls are used, so it is not out of the question to call timex( ) from within an interrupt handler. However, it should not be called with interrupts disabled. Also, timex3( ) in particular consumes more time than is good practice in an interrupt handler.

There is one further assumption, but it would not often present a problem. A simple but overly general form of the assumption is that no other program running concurrently is also using the READ_BACK, LATCH, or related operations with counter 0 of the 8254 or 8253. There would be no harm if simultaneous running is done correctly and in such a way that one latch-read sequence never overlaps with another. For correct programming, the 8254 or 8253 data sheets must be consulted, and the prevention of overlap must be ensured by performing each latch-read sequence while interrupts are disabled. IBM MS-DOS makes no use of the capabilities to read the hardware counter registers, so programs that use only BIOS and DOS calls for input and output should be perfectly safe. Such programs would include those written in C if use is made only of the standard input and output functions, such as printf( ), scanf( ), getchar( ), and the like (see Kernighan & Ritchie, 1988).

REFERENCES

EMERSON, P. L. (1988). The best crude timer for MS-DOS implementations of C. *Behavior Research Methods, Instruments, & Computers*, **20**, 583-584.

EMERSON, P. L. (1988). Using serial interfaces and the C language for real time experiments. *Behavior Research Methods, Instruments, & Computers*, **20**, 330-336.

KERNIGHAN, B. W., & RITCHIE, D. M. (1988). *The C programming language* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall.

SARGENT, M., & SHOEMAKER, R. L. (1986). *The IBM PC from inside out*. Reading, MA: Addison-Wesley.

LISTING 1
The timex( ) Subroutine

```
/* timex.c */
#include <dos.h>                        /* Turbo header for low level I/O. */
#define TIMER0 0x40                     /* Port for 8254 ctr0.             */
#define CNTRL 0x43                      /* Port for 8254 control byte.     */
#define READ_BACK 0xc2                  /* Command to latch status & ctr.  */
#define BIOSSEG 0x40                    /* Memory segment, BIOS.           */
#define TICS_LO 0x6c                    /* Address of low tick count.      */
#define TICS_HI 0x6e                    /* Address of high tick count.     */

long timex()                               /* The timer subroutine.        */
{ unsigned tmp1,tmp2,tmp3,tmp4,hi,lo ;     /* Working variables.           */
  int status ;                             /* Status byte from 8254.       */
  long ticks1,ticks2 ;                     /* Ticks at times 1 & 2.        */
  disable() ;                              /* Disable interrupts.          */
  outp(CNTRL,READ_BACK) ;                  /* Execute latch command.       */
  status = inp(TIMER0) ;                   /* Get 8254 status, ctr0.       */
  lo = (unsigned) inp(TIMER0) ;            /* Get count, low byte.         */
```

LISTING 1 (Continued)

```
    hi = (unsigned) inp(TIMER0) ;              /* Get count, high byte.       */
    tmp1 = (unsigned) peek(BIOSSEG,TICS_LO) ;  /* Read BIOS ticks, low        */
    tmp2 = (unsigned) peek(BIOSSEG,TICS_HI) ;  /* and high words.             */
    enable() ;                                 /* Re-enable interrupts.       */
    hi = hi << 2 | lo >> 6 ;                   /* Use 10 bits of the 16.      */
    disable() ;                                /* Disable interrupts.         */
    tmp3 = (unsigned) peek(BIOSSEG,TICS_LO) ;  /* Read BIOS ticks again,      */
    tmp4 = (unsigned) peek(BIOSSEG,TICS_HI) ;  /* which are used below.       */
    enable() ;                                 /* Re-enable interrupts.       */
    if( !(lo & 63) ) hi-- ;                    /* Adjust hi for easy logic.   */
    hi = (~hi & 1023) >> 1 ;                   /* Change to up-count & halve. */
    if( !(status & 128) ) hi |= 512 ;          /* Add 512 if 2nd-half latch.  */
    ticks1 = (long) tmp2 << 16 | tmp1 ;        /* Construct the two tick      */
    ticks2 = (long) tmp4 << 16 | tmp3 ;        /* readings as longs.          */
    if(ticks2 > ticks1 + 1) return(-1) ;       /* Error return.               */
    if(hi & 512) return(ticks1 << 10 | hi) ;   /* Use ticks1 if later latch,  */
    return(ticks2 << 10 | hi) ;                /* ticks2 if earlier latch.    */
}
```

LISTING 2
Test Program Measuring Intervals Between Presses of the ENTER Key on the Keyboard

```
/* ttimex.c */
#include <stdio.h>                             /* Standard header.            */
#include "timex.c"                             /* The timex file.             */
long timex() ;                                 /* Redeclare in case of        */
                                               /* combining by linking.       */
main()                                         /* Program to measure times    */
{ long ztime,etime ;                           /* between presses of ENTER.   */
  while(1)                                     /* Infinite loop, use ^C.      */
   { printf("press ENTER to start timing\n") ; /* Message to screen.          */
     getchar() ;                               /* Wait for ENTER.             */
     ztime = timex() ;                         /* Get beginning time.         */
     printf("press ENTER to read timer\n") ;   /* Message to screen.          */
     getchar() ;                               /* Wait for ENTER.             */
     etime = timex() ;                         /* Get terminal time.          */
     if( ztime < 0 || etime < 0 ) break ;      /* Stop if error.              */
     etime -= ztime ;                          /* Take the difference.        */
     printf("time was %ld\n",etime) ;          /* Print it to screen.         */
   }
  printf("timing error\n") ;                   /* Error message to screen.    */
}
```

LISTING 3
The timex3( ) Subroutine

```
/* timex3.c */
#include <dos.h>                        /* Turbo header for low level I/O. */
#define TIMER0 0x40                     /* Port for 8253 ctr0.             */
#define CNTRL 0x43                      /* Port for 8253 control byte.     */
#define LATCH 0                         /* Command to latch  ctr.          */
#define BIOSSEG 0x40                    /* Memory segment, BIOS.           */
#define TICS_LO 0x6c                    /* Address of low tick count.      */
#define TICS_HI 0x6e                    /* Address of high tick count.     */

long timex3()                             /* The timer subroutine.       */
{ unsigned thi1,tlo1,thi2,tlo2,tlo3 ;     /* Working variables.          */
  unsigned lo,hi1,hi2,hi3,hi4,hi5,lo2 ;
  int status ;
  long ticks1,ticks2 ;                     /* Ticks at times 1 & 2.       */
  disable() ;                              /* Disable interrupts.         */
  outp(CNTRL,LATCH) ;                      /* Execute latch command.      */
  lo = (unsigned) inp(TIMER0) ;            /* Get count, low byte.        */
  hi1 = (unsigned) inp(TIMER0) ;           /* Get count, high byte.       */
  enable() ;                               /* Re-enable interrupts.       */
  hi1 = hi1 << 2 | lo >> 6 ;               /* 10-bit downcount.           */
  disable() ;                              /* Disable interrupts.         */
  outp(CNTRL,LATCH) ;                      /* 1st tick reading &          */
  lo2 = (unsigned) inp(TIMER0) ;           /* 2nd 8253 reading.           */
  hi2 = (unsigned) inp(TIMER0) ;
  tlo1 = (unsigned) peek(BIOSSEG,TICS_LO) ;
  thi1 = (unsigned) peek(BIOSSEG,TICS_HI) ;
  enable() ;
  hi2 = hi2 << 2 | lo2 >> 6 ;
  disable() ;
  outp(CNTRL,LATCH) ;                      /* 2nd tick reading &          */
  lo = (unsigned) inp(TIMER0) ;            /* 3rd 8253 reading.           */
  hi3 = (unsigned) inp(TIMER0) ;
  tlo2 = (unsigned) peek(BIOSSEG,TICS_LO) ;
  thi2 = (unsigned) peek(BIOSSEG,TICS_HI) ;
  enable() ;
  hi3 = hi3 << 2 | lo >> 6 ;
  for(hi5=hi4=hi3 ; hi4 <= hi5 ; )         /* Now wait for end of         */
    { hi5 = hi4 ;                          /*    countdown.               */
      disable() ;
      outp(CNTRL,LATCH) ;
      lo = (unsigned) inp(TIMER0) ;
      hi4 = (unsigned) inp(TIMER0) ;
      enable() ;
      hi4 = hi4 << 2 | lo >> 6 ;
    }
  tlo3 = (unsigned) peek(BIOSSEG,TICS_LO) ;  /* 3rd tick reading.         */
  if(tlo1 != tlo2)                         /* Status logic.               */
    { if(hi1 > hi2) status = 0 ;
      else status = -1 ;
    }
```

LISTING 3 (Continued)

```
  else
    { if(hi1 < hi2) status = -1 ;
      else if(hi2 < hi3) status = 0 ;
           else
             { if(tlo2 == tlo3) status = -1 ;
               else status = 0 ;
             }
    }
  if( !(lo2 & 63) ) hi2-- ;                    /* See comments in timex().   */
  hi2 = ( ~hi2 & 1023) >> 1 ;
  if( !(status & 128) ) hi2 |= 512 ;
  ticks1 = (long) thi1 << 16 | tlo1 ;
  ticks2 = (long) thi2 << 16 | tlo2 ;
  if( ticks2 > ticks1 + 1 ) return(-1) ;
  if( hi2 & 512 ) return(ticks1 << 10 | hi2) ;
  return(ticks2 << 10 | hi2) ;
}
```

(Revised manuscript accepted for publication October 15, 1988.)