Behavior Research Methods, Instruments, & Computers
2003, 35 (1), 96-108

Timing accuracy under Microsoft Windows revealed through external chronometry

CHRISTOPHER D. CHAMBERS
University of Melbourne, Melbourne, Victoria, Australia

and

MEL BROWN
Monash University, Melbourne, Victoria, Australia

Recent studies by Myors (1998, 1999) have concluded that the Microsoft Windows operating system is unable to support sufficient timing precision and resolution for use in psychological research. In the present study, we reexamined the timing accuracy of Windows 95/98, using (1) external chronometry, (2) methods to maximize the system priority of timing software, and (3) timing functions with a theoretical resolution of 1 msec or better. The suitability of various peripheral response devices and the relative timing accuracy of computers with microprocessors with different speeds were also explored. The results indicate that if software is properly controlled, submillisecond timing resolution is achievable under Windows with both old and new computers alike. Of the computer input devices tested, the standard parallel port was revealed as the most precise, and the serial mouse also exhibited sufficient timing precision for use in single-interval reaction time experiments.

The accuracy of computer-based timing is a central methodological issue in experimental psychology. Under the MS-DOS operating system, several studies have shown that millisecond precision timing is achievable by accessing a low-level system timer through various high-level programming languages, such as QuickBasic (Graves & Bradley, 1987, 1988), Borland C (Dlhopolsky, 1988; Emerson, 1988; Warner & Martin, 1999), and Turbo Pascal (Bovens & Brysbaert, 1990; Creeger, Miller, & Paredes, 1990; Hamm, 2001; Heathcote, 1988). The timing resolution of various input devices, including keyboard, mouse, and game port is also well established under MS-DOS (Beringer, 1992; Crosbie, 1990; Segalowitz & Graves, 1990). Newer operating systems, such as Microsoft Windows, have now largely superceded the MS-DOS architecture. The Windows interface provides aesthetic and computational advantages, but opinion is divided over the obtainable timing precision. Recent evidence has suggested that the multitasking environment of Windows may be unsuitable for psychological experiments because the operating system controls the system access necessary for precise timing (Myors, 1998, 1999). Alternatively, McKinney, MacCormac, and Welsh-Bohmer (1999) have proposed a method for achieving 0.1-msec timing precision under Windows 3.1 with the addition of extra hardware. In addition, a number of software packages are available either commercially (e.g., E-Prime) or at no expense (e.g., DMDX) that claim to provide millisecond precision timing under Windows 95/98 but either remain untested.[1]

The studies by Myors (1998, 1999) have provided the first evidence that the Windows environment, without added hardware, may be unable to support sufficient timing accuracy for psychological research. Myors (1999) took an internal time stamp before and after an event that was expected to take 500 msec. In one condition, the event was a keypress triggered every 500 msec; in the other, the event was 35 screen refreshes at a refresh rate of 70 Hz. Under different versions of the Windows operating system, the error variance of the elapsed time increased to as much as 3,036 msec^2, as compared with submicrosecond variance under MS-DOS. Although these results are striking, Myors's method and interpretation contain several potential flaws. First, all timing measurements were internally generated and were not compared with an external chronometer. Therefore, it is not possible to determine what proportion of the measured variance under Windows originated from the system timer and what proportion was due to variability in the actual duration of the specified event. For example, added variability in the time required to register a keypress or perform a screen refresh would increase the variability of the elapsed time independently of any actual variance in the system timer. Second, during the experimental conditions employing Windows, it is unclear

Parts of this paper were presented at the 28th Annual Australian Experimental Psychology Conference, Melbourne, April 2001. We thank Mark Williams, Jeff Hamm, Richard Plant, and Jon Vaughan for helpful comments on an earlier draft of this manuscript. Correspondence concerning this article should be addressed to C. D. Chambers, Cognitive Neuroscience Laboratory, Department of Psychology, University of Melbourne, Victoria, Australia 3010 (e-mail: c.chambers@psych.unimelb.edu.au).

whether background applications were closed down prior to testing and whether the timing program was afforded maximum priority in the stream of operations continually undertaken by a multitasking operating system. Third, the program was written in C under MS-DOS and executed through an MS-DOS prompt under Windows, so the applicability of the interpretation to a pure Windows environment may be limited.

We sought to clarify the best achievable timing precision of the Windows operating system over two experiments. In Experiment 1, the precision of the system timer was examined independently of other internal processes. In Experiment 2, the timing accuracy of input devices was investigated, including the standard serial mouse, the PS/2 mouse, and the parallel port.

EXPERIMENT 1
Windows Timing Precision

It has been argued that accurate timing under Windows 95/98 is complicated by software interruptions as a consequence of multitasking and an inability to access a low-level system timer (McKinney et al., 1999; Myors, 1999). However, the Windows programming environment provides potential solutions to these problems. First, the priority of a program within the multitasking stream can be adjusted to prevent or minimize interruptions. Second, a number of timing functions are available that, theoretically, enable access to high-resolution timing.

In this experiment, we tested two timing functions that run under the Windows 95/98 operating systems: *timeGet Time*, which accesses a tick counter with a maximum theoretical resolution of 1 msec; and *QueryPerformance Counter*, which accesses a system timer with a maximum theoretical resolution of 0.8381 μsec. In addition, the priority of the test program was maximized. Both timers were tested in each of two computers: a low-end 486 DX and a high-end Pentium III, to examine whether the accuracy of timing under Windows is affected by computer configuration.

Method

Hardware

Test computers. The test machines were IBM-compatible PCs: a 486 DX, with 66-MHz CPU, a 33-MHz Bus speed, running Windows 95 with 16 MB SD RAM; and a Pentium III, with 666-MHz CPU, 133-MHz Bus speed, running Windows 98 with 256 MB SD RAM.

Preliminary verification of external chronometry. The external timer was a Tucker-Davis Technology System-II signal generation and analysis system (ExT). This apparatus enables real-time recording of an input voltage at a maximum analog-to-digital (A/D) sampling rate of 100 kHz. The ExT was controlled by a dedicated hardware card (AP2) in the test computer, which was controlled from the test program. Timing information was relayed to the ExT A/D input by switching the voltage of the parallel port on the test computer. The ExT digitized the parallel port voltage at a 100-kHz sampling rate and, therefore, served as an external timer with a resolution of 10 μsec.

Prior to Experiment 1, a preliminary investigation was undertaken to examine the latency associated with writing to the parallel port from software and, hence, the suitability of using the parallel port as an external time stamp. The initial test program was written and compiled under Visual Basic 5 (VB5) for reasons outlined below in the Software section. Under VB5, read and write operations on the parallel port are supported by the *dollx8.dll* dynamic link library.[2] The write subroutine within *dollx8.dll* is declared as

```
Declare Sub WriteDOLLx8 Lib “dollx8.dll” (ByVal x8Port As Integer,
ByVal value As Integer)
```

The x8Port argument of WriteDOLLx8 is an 8-bit number that identifies the port address. For these experiments, we used the Control port, which occupies Pins 1–7 of the parallel port. A constant called ControlIOPort was assigned as the address for the Control port:

```
Const ControlIOPort As Integer = 890
```

Clearly, any variance of the delay associated with switching the parallel port from software must be low if the Control port voltage is to be regarded as a suitable time stamp. To examine this variance, a series of write statements were executed serially while the ExT recorded the voltage on Pin 1. The code statements were organized so that the voltage was repeatedly switched between a high state (+5 V) and a low state (0 V), as shown below.

```
‘--- Define loop variable
Dim i as long
For i = 1 to 100000
        ‘--- Switch Control port high(+5v)
        WriteDOLLx8 ControlIOPort, 2
        ‘--- Switch Control port low (0v)
        WriteDOLLx8 ControlIOPort, 3
Next i
```

One loop of 100,000 trials was conducted. Following recording, the number of A/D samples between each voltage maximum and minimum was counted and multiplied by the ExT sampling interval (10 μsec) to calculate the time between successive write executions and, hence, the parallel port write delay (see Figure 1). Across the entire sample of trials, the average write delay was 21.6 μsec (SD = 0.05 μsec) for the 486 and a constant 10 μsec (SD = 0 μsec) for the PIII.[3] The low magnitude and, particularly, the low variance of the write delay indicate that switching the parallel port voltage provides a suitable time stamp for external analysis.

Software

Programming language. The test program was written in VB5. This programming language was chosen for several reasons, many of which are cited by McKinney et al. (1999). The language is native to the Windows environment and specifically engineered for Windows programming. In addition, the software is user friendly and enables easy handling of subsidiary packages that are convenient for automated data analysis.

Thread priority code. The program or *thread* priority was manipulated in two phases. First, the base priority of the system resources, or process, in which the thread operates was optimized. Second, the priority of the thread was maximized relative to any other threads within the same process.[4]

The procedure for implementing these changes is as follows. Initially, the application programming interface (API) functions, *GetCurrentProcess*, *GetCurrentThread*, *SetPriorityClass*, and *SetThreadPriority* must be declared, along with constants for maximizing the priority class (REALTIME_PRIORITY_CLASS) and thread priority (THREAD_PRIORITY_TIME_CRITICAL):

```
Declare Function GetCurrentProcess Lib “kernel32” () As Long
Declare Function GetCurrentThread Lib “kernel32” () As Long
Declare Function SetPriorityClass Lib “kernel32” (ByVal hProcess As
Long, ByVal dwPriorityClass As Long) As Long
```

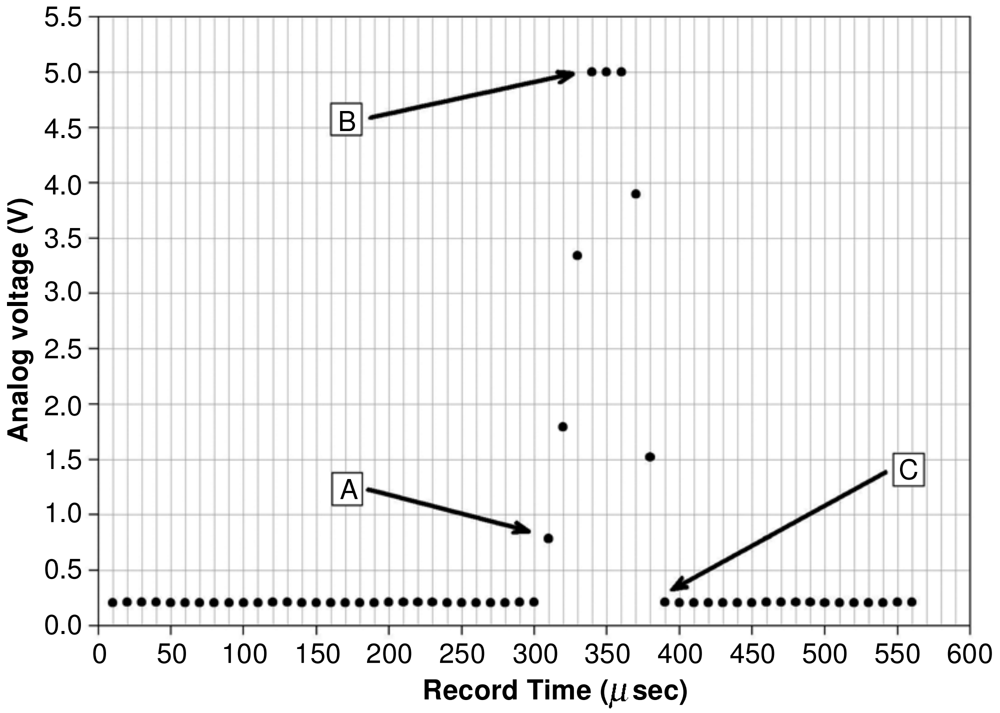

Figure 1. A hypothetical ExT output from which the write delay on the Control port was calculated. Recording begins at Record Time 0, with a sample taken every 10 µsec. At Point A, the initial write statement is being implemented (WriteDOLLx8 ControlIOPort, 2), which sets the Control port to a high state (+5 V). At Point B, this write statement has been implemented, which signifies the beginning of the delay period for the next WriteDOLLx8 statement. At Point C, the second write statement has completed, and the Control port has been returned to a low state (WriteDOLLx8 ControlIOPort, 3). The write delay is calculated as the time difference between the completion of the first write execution at Point B (350 µsec) and the completion of the second write execution at Point C (390 µsec). Hence, the write delay in this example is 40 µsec.

```
Declare Function SetThreadPriority Lib "kernel32" (ByVal hThread
As Long, ByVal nPriority As Long) As Long
Const REALTIME_PRIORITY_CLASS = &H100
Const THREAD_PRIORITY_TIME_CRITICAL = 15
```

The first stage in adjusting the priority then requires the handles for the current process and current thread to be retrieved:

```
'---Define the thread handling variables
Dim CurrentProcess As Long, CurrentThread As Long
Dim SetPC As Long, SetTP As Long
'---Retrieve handles for the current process and current thread
CurrentProcess = GetCurrentProcess()
CurrentThread = GetCurrentThread()
```

The priority class and thread priority are then maximized by assigning the constants REALTIME_PRIORITY_CLASS and THREAD_PRIORITY_TIME_CRITICAL to the function arguments. Both *SetPriorityClass* and *SetThreadPriority* functions return zero if the assignment failed, which allows an error trapping routine to be added:

```
SetPC = SetPriorityClass(CurrentProcess, REALTIME_PRIORITY_
CLASS)
If SetPC = 0 Then
        '---SetPriorityClass returned zero so raise an error
        Err.Raise 1, 0, "Cannot set the priority class!"
End If
SetTP = SetThreadPriority(CurrentThread, THREAD_PRIORITY_
TIME_CRITICAL)
If SetTP = 0 Then
        '---SetThreadPriority returned zero so raise an error
        Err.Raise 1, 0, "Cannot set the thread priority!"
End If
```

Timing code: timeGetTime. The *timeGetTime* function accesses a system clock that registers the number of elapsed milliseconds since Windows was last initialized. The procedure for accessing *timeGetTime* is as follows. The function must first be declared through the dynamic link library *winmm.dll*:

```
Declare Function timeGetTime Lib "winmm.dll" () As Long
```

The variables that receive time stamps must then be defined as long integers, to receive the 32-bit number returned by the *timeGetTime* function:

```
Dim StartTime as Long, CurrentTime as Long, ElapsedTime as Long
```

To use *timeGetTime*, the clock count is simply taken before and after a critical event, and the difference between the two time stamps is calculated as the elapsed time:[5]

```
StartTime = timeGetTime()
'** PERFORM CRITICAL EVENT HERE**
CurrentTime = timeGetTime()
ElapsedTime = CurrentTime – StartTime
```

Timing code: QueryPerformanceCounter. The *QueryPerformanceCounter* function accesses a system tick counter that increments at a constant frequency. To convert ticks into time intervals, the frequency of the counter must be obtained using the *QueryPer-*

formanceFrequency function. For most machines, the tick frequency is 1193180 Hz. The maximum theoretical resolution of this timer (in μsec) is, therefore, 1000000/1193180, or 0.8381 μsec.[6]

The procedure for accessing the high-performance timer is as follows. Initially, the appropriate API functions must be declared, which use the 64-bit Currency data type to return the current tick count and an Integer to return the operative status of the high performance timer:[7]

```
Declare Function QueryPerformanceCounter Lib "Kernel32" (X As
Currency) As Integer
Declare Function QueryPerformanceFrequency Lib "Kernel32" (X As
Currency) As Integer
```

The variables that receive the tick frequency, the start tick count, and the stop tick count are then defined, and the status of the high performance timer is retrieved. The *QueryPerformanceCounter* function returns zero if the high-performance timer is inaccessible or nonexistent:

```
'---Define the timing variables
Dim Freq as Currency
Dim StartCount As Currency, CurrentCount As Currency,
Elapsed Time as Currency
'---Retrieve the status and frequency of the high performance timer
If QueryPerformanceCounter(StartCount) = 0 Then Exit Sub
QueryPerformanceFrequency Freq
```

The procedure for calculating the elapsed time is similar to the *timeGetTime* method. A tick stamp is taken before and after a critical event, with the time interval calculated as the tick difference divided by the tick frequency and multiplied by 1,000 to convert to milliseconds:

```
QueryPerformanceCounter StartCount
'** PERFORM CRITICAL EVENT HERE**
QueryPerformanceCounter CurrentCount
ElapsedTime = (CurrentCount - StartCount) / Freq * 1000
```

Procedure

For both timers, the ExT began recording the voltage on the Control port following the software trigger "DD1Go 1". An initial time stamp was then registered, and the Control port voltage was set high, which was recorded by the A/D input on the ExT as a +5-V rise and plateau. The code then entered a loop in which the elapsed time was continuously measured until it was equal to the preset delay (SetDelay). After exiting the loop, the Control port was immediately set low. The number of recorded A/D samples between ascending and descending voltage spikes was then counted and multiplied by the A/D sampling interval to calculate the observed duration in microseconds (see Figure 2). The two timing loops, as implemented, are presented below.

Millisecond Timer Loop (*timeGetTime*):

```
'--- Start Recording on ExT
DD1Go 1
'--- Set the Control port voltage high
WriteDOLLx8 ControlIOPort, 2
'--- Register the initial Time Stamp
StartTime = timeGetTime()
'--- Loop until elapsed time equals the preset delay
Do While ElapsedTime < SetDelay
        CurrentTime = timeGetTime()
        ElapsedTime = CurrentTime – StartTime
Loop
```

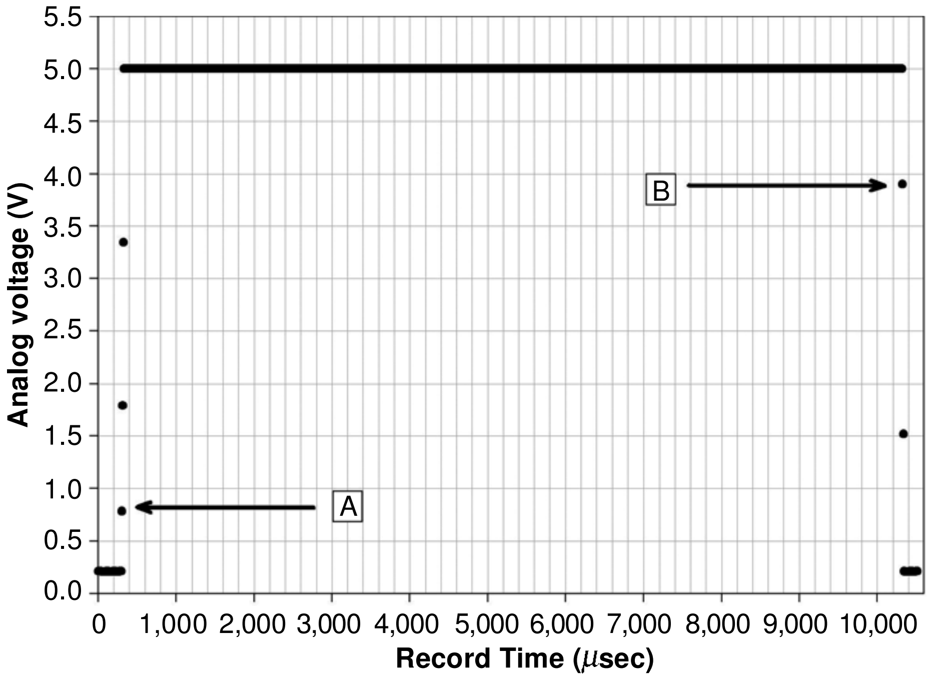

Figure 2. A hypothetical ExT output at a sampling rate of 100 kHz, from which the actual elapsed time was calculated. The time stamp set by the first WriteDOLLx8 statement is recorded at Point A, which signifies the start of the measured duration. The Control port remains high (+5 V) until the preset delay on the Windows timer is reached, at which point the voltage is returned to ground. Point B is the first observed return from the maximum plateau and is the finishing stamp of the duration. The elapsed time is calculated by subtracting the time address at Point B (10,330 µsec) from the time address at Point A (310 µsec). This difference (10,020) is then divided by 1,000 to convert the elapsed time into milliseconds. For this example, the elapsed time is therefore 10.02 msec.

```
'--- Set the Control port voltage low
WriteDOLLx8 ControlIOPort, 3
'--- Stop recording on ExT
DD1Stop 1
```

High-Performance Timer Loop (*QueryPerformanceCounter*):

```
'--- Start Recording on ExT
DD1Go 1
'--- Set the Control port voltage high
WriteDOLLx8 ControlIOPort, 2
'--- Register the initial Tick Stamp
QueryPerformanceCounter StartCount
'--- Loop until elapsed time equals the preset delay
Do While ElapsedTime < SetDelay
        QueryPerformanceCounter CurrentCount
        ElapsedTime = (CurrentCount - StartCount) / Freq * 1000
Loop
'--- Set the Control port voltage low
WriteDOLLx8 ControlIOPort, 3
'--- Stop recording on ExT
DD1Stop 1
```

For each of the two timing functions, 24 preset delays ranging from 5 to 50 msec in 5-msec increments, from 100 to 1,000 msec in 100-msec increments, and from 1,250 to 2,000 msec in 250-msec increments, were tested on each computer. One hundred thousand trials were conducted at each preset delay, generating 4.8×10^6 timing estimates for each machine. Network cards were removed from the test machines during testing. In addition, all memory-resident programs, as visible in the Close Program dialog box (obtained by pressing Ctrl–Alt–Del), were closed, except for Explorer and the test program.

Results and Discussion

The dependent variables of interest in this experiment were the precision and resolution of computer timing across the various conditions, as measured externally by the ExT. The precision of the timer can be regarded as the maximum difference between the obtained duration and the expected duration. Consequently, to achieve *millisecond precision*, 100% of obtained durations within each block of 100,000 trials must fall within 1 msec of the corresponding preset duration (i.e., preset duration ± 1 msec). The resolution of the timer is calculated as the total range of obtained durations within each block and includes the upper and lower limits of the obtained frequency distribution. To achieve *millisecond resolution*, the difference between maximum and minimum obtained durations within each block must not exceed 1 msec.

Timing Precision

Figure 3 presents the average deviation of the obtained durations from the corresponding preset (or expected) durations as a function of the preset delay magnitude, timer function, and test machine. The two errors bars surrounding each data point are the standard deviation (thick error bar with larger cap width) and the range (thin error bar with smaller cap width). The upper panels of Figures 3A and 3B, respectively, report data obtained with the *timeGetTime* function in the 486 and the PIII. Across all preset durations in the 486 condition, it is apparent that the average difference between observed and expected durations is consistently negative ($M = -0.437$ msec, $SD = 0.089$ msec). This indicates that the *timeGetTime* function underestimates the true elapsed time and is effectively running fast. It can also be seen that this systematic timing error increases as the preset delay is increased, in accordance with a gradual accumulation of error, or timing drift (see also Figure 4A). For preset durations of 800 msec and greater, the expansion of systematic timing error exceeds the critical limit for millisecond precision timing, as is shown by the range error bars in Figure 3A, which cross the lower reference line. As also is indicated by the filled circles of Figure 4A, the poorest precision was obtained for the preset duration of 2,000 msec, at which the maximum difference between observed and expected durations was 1.19 msec. However, the standard deviation of the estimates remained consistent ($M = 0.291$ msec, $SD = 0.013$ msec), indicating that the accumulation of systematic error was not paralleled by an increase in random error.

For the PIII, timing error associated with the *timeGetTime* function was substantially reduced (Figures 3B and 4A), with all obtained durations falling within 1 msec of the corresponding preset durations (maximum deviation = 0.72 msec). However, it can be seen that even under the PIII, the *timeGetTime* function underestimated the elapsed time by an average of 0.184 msec ($SD = 0.019$ msec). The consistent direction of this error across both machines suggests that systematic underestimation of elapsed time may be a feature of the *timeGetTime* function. Although these effects are minor in the present context, it should be noted that for much longer preset durations than those reported here, cumulative timing drift resulting from systematic timing error would be expected to increasingly widen the difference between preset and actual durations. This problem would become salient if the computer was required to synchronize events with another machine or external apparatus over an extended period.

Figures 3C and 3D report data obtained with the *Query PerformanceCounter* function. For both the 486 and the PIII, all obtained durations fell within 1 msec of the preset durations and thus satisfied the requirements of millisecond precision. The standard deviation of each estimate was substantially reduced relative to the *timeGetTime* function and averaged 0.021 msec ($SD = 0.007$ msec) for the 486 and 0.009 msec ($SD = 0.001$ msec) for the PIII. From Figure 4A, it can be seen that the maximum departure from perfect precision was also much reduced relative to the *timeGetTime* function, peaking at 0.61 msec for the 2,000-msec duration on the PIII. At a finer level, the *QueryPerformanceCounter* function exhibited systematic timing error in both machines. For the 486, all average deviations slightly exceeded zero ($M = 0.064$ msec, $SD = 0.002$ msec), whereas for the PIII the same trend was apparent at a greater magnitude ($M = 0.374$ msec, $SD = 0.077$ msec). The *QueryPerformanceCounter* function therefore appears to overestimate the elapsed time, in contrast to the direction of systematic timing error observed for the *timeGetTime* function. Consistent with the *timeGetTime* results, however, is the fact that the computer

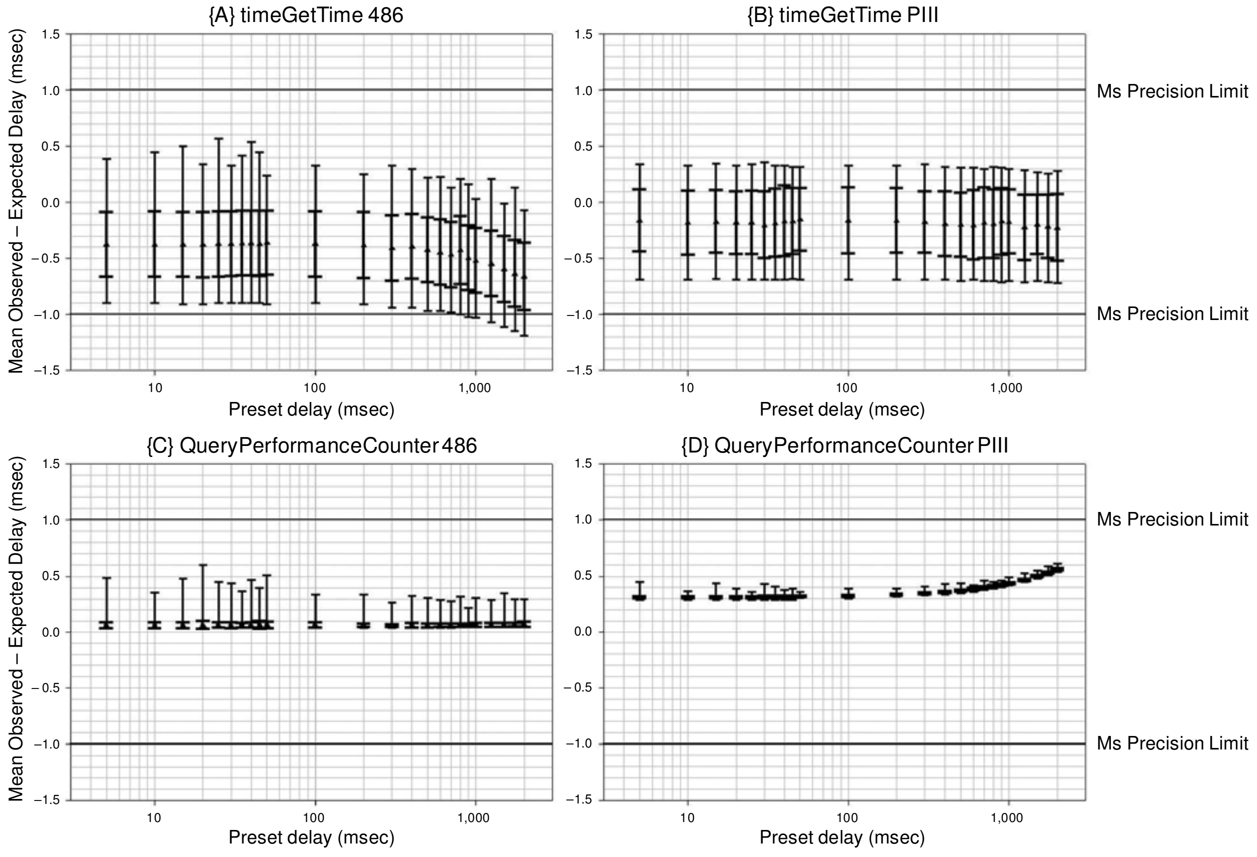

Figure 3. Timing results for the 486 and the PIII, using the *timeGetTime* (A, B) and the *QueryPerformanceCounter* (C, D) functions. In each panel, the average difference between the observed and the expected durations is plotted as a function of the preset delay magnitude. The thin error bars with smaller caps represent the range of obtained durations within each block, and the thick error bars with larger caps are ±1 *SD*. The horizontal reference lines in each panel indicate the thresholds for millisecond precision timing, defined as ±1 msec from the expected duration.

displaying the greater overall systematic error also demonstrated a more accelerated cumulative timing drift.

In summary, the timing precision for most conditions was better than 1 msec. Both timers under the PIII met the criteria for millisecond precision, as did the *Query PerformanceCounter* function for the 486. The precision of the *timeGetTime* function on the 486 was within 1 msec until the preset delay exceeded 800 msec. The absence of significant random timing errors indicates that the thread priority manipulations were successful in preventing software interruptions across all 9.8×10^6 trials. Under these conditions, the precision of timing across a range of low- and high-performance computers is sufficient for most psychology experiments.

Timing Resolution

The resolution of the timer provides an indication of the spread or variability of the obtained durations. The resolution results are plotted in Figure 4B against the preset duration, for both test machines and both timing functions. Each resolution data point is calculated as the range of the deviation between observed and expected delays and is equivalent to the length of the range error bars presented in Figure 3. For the *timeGetTime* function, the average resolution, collapsed across preset durations, was 1.25 msec for the 486 (*SD* = 0.107 msec, maximum = 1.47 msec) and 1.01 msec for the PIII (*SD* = 0.018 msec, maximum = 1.05 msec). For the *QueryPerformanceCounter* function, the resolution significantly improved, averaging 0.325 msec for the 486 (*SD* = 0.095 msec, maximum = 0.56 msec) and 0.088 msec for the PIII (*SD* = 0.028 msec, maximum = 0.16 msec). For the purposes of psychological experimentation, the resolution of both timers on both machines is well within requirements.

Recommendations

The results of this experiment suggest that precise, high-resolution timing is achievable under Windows in both low- and high-end computers, contrary to the conclusion of Myors (1998, 1999). Three important caveats must be noted, however, with respect to the present results. First, the precision and resolution of timing measurements could conceivably be affected by added operations occurring during a timing loop (and within the same programming thread), particularly if these opera-

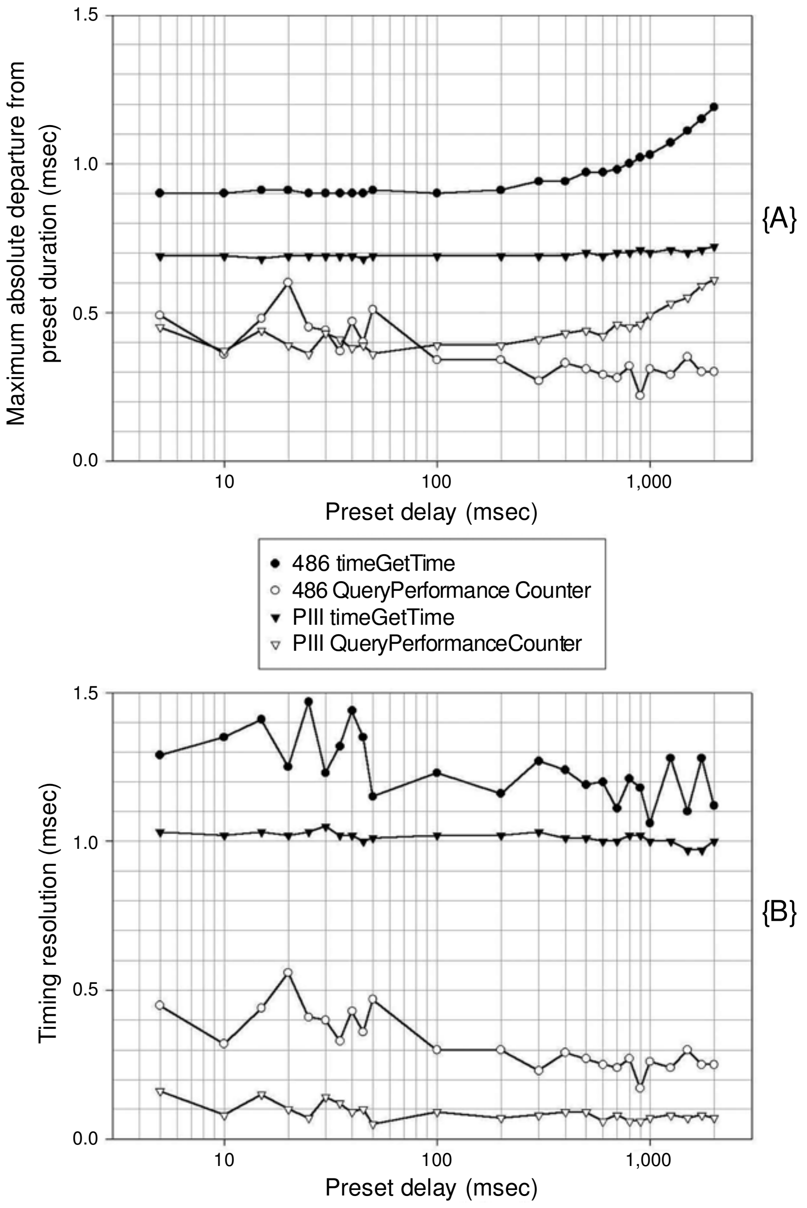

Figure 4. Summaries of timing precision (A) and resolution (B) across both timers and test machines. In panel A, the maximum disparity between observed and expected durations is plotted as a function of the preset duration for each combination of computer and timer. In panel B, the resolution ordinate is calculated as the range of obtained values within each block and is plotted against the magnitude of the preset delay.

tions place high demand on the operating system. Second, as was mentioned in the Method section, all trials in the present experiment were conducted with network cards and drivers removed and with all memory-resident programs closed, except for Explorer and the testing software. Experimenters are therefore advised to replicate these conditions closely to minimize the likelihood of incurring timing interruptions. Finally, the present experiment did not examine whether the precision and resolution of timing measurements under Windows are preserved when the effects of input devices are included in the examination. This question was explored in Experiment 2.

EXPERIMENT 2
Timing Accuracy of Peripheral Input Registration

Many experimental paradigms in psychological research adopt reaction time (RT) as a dependent variable. At a broad level, the error variance associated with RT measurements can be attributed to the sum of subject variance and measurement variance. Measurement variance, in turn, can be partitioned into the variability associated with timing and the variability associated with the computer registration of the response.

Experiment 1 indicated that if the Windows environment is properly controlled, the timing variance component of measurement variance is relatively low, peaking at 0.084 msec^2 for the *timeGetTime* function in the 486. With the addition of an input device, an added variable delay is introduced between the initiation of a response and computer registration of the response. Unless the variance of the registration delay is zero, the true measurement variance will be higher than Experiment 1 suggested, and the timing resolution will be concomitantly lower. The purpose of Experiment 2 was to quantify this added variance for various auxiliary input devices and, thus, provide an estimate of the timing accuracy of Windows directly applicable to RT experiments.

Several studies have examined the performance of the keyboard, mouse, and game port under MS-DOS, rather than Windows. Segalowitz and Graves (1990) reported a combination of systematic and random timing error when these devices were used on IBM XT and AT computers. For the keyboard, a systematic 10-msec delay was observed, along with a random variation of ±7.5 msec on an XT and ±5 msec on an AT. For the serial mouse, a constant delay of 31 msec was observed with a ±2-msec error, which increased to 45 msec (±15 msec) when the mouse ball was moved prior to a buttonpress (see also Crosbie, 1990). The PS/2 mouse was found to perform less accurately, with a change in status reported only every 10 msec. Overall, the game port was found to be the most accurate input device, with a precision of 1 msec and negligible variability (see also Graves & Bradley, 1987).

The mouse timing results of Segalowitz and Graves (1990) and Crosbie (1990) have since been confirmed and expanded by Beringer (1992). Beringer demonstrated that the transmission rate of 1,200 baud on the serial mouse results in a fixed minimum response delay of 22.5 msec, to which is added a 7-msec delay to overcome bouncing button contacts. In addition, Beringer confirmed the higher variability of the PS/2 mouse and noted that the status of the PS/2 interface is scanned at a rate of ~60 Hz.

The results of these studies may, of course, have few implications for the magnitude and variability of registration delays encountered on newer computers under the considerably more complex and developed Windows operating system. However, these experiments do suggest that significant delays are to be expected in most devices and that the variability of these delays should not be ignored. In Experiment 2, we examined the response delays associated with the serial mouse, the PS/2 mouse, and the parallel port under Windows. We chose to examine the parallel port because of its widespread implementation as a printer interface in IBM-compatible computers and ease of software control. We also included four extensions on earlier work. First, all previous studies have investigated only one mouse model from each type. In the present experiment, multiples of the same model of serial mouse were tested in order to examine within-model performance variability. Second, all previous studies have triggered mice electronically, so the obtained registration delays do not take into account mechanical influences of the apparatus. In the present experiment, all the mice were tested manually, thus including electronic variability resulting from mechanical closing of the button switch (such as electrical bounce from key contacts). Third, registration delays associated with both left and right buttonpresses were compared in the present experiment, unlike in previous reports. Finally, there has been no investigation into the interaction between registration delays and computer configuration. This comparison was satisfied in the present experiment by conducting timing tests on both a 486 and a PIII.

Method

Hardware

Test computers. The same test machines as those in Experiment 1 were used.

Peripheral devices. A total of eight two-button mice were tested. These included five Microsoft 2.0A serial mice (four with Federal Communications Commission [FCC] ID No. C3KSS1, one with FCC ID No. C3KKS2), one Microsoft 2.1A serial mouse (FCC ID No. C3KKS8), one non-Microsoft serial mouse (Digicor brand, Serial No. 9827267), and one Microsoft 2.1A PS/2 port compatible mouse (FCC ID No. C3KKMP1). All the mice were tested on both test machines, with the exception of the PS/2 mouse, which was tested only on the PIII, owing to the absence of a compatible PS/2 port on the 486.

External chronometry. The design was similar to that in Experiment 1, with the ExT used for all external timing. For the mouse-timing tests, the rising edge of a TTL pulse resulting from a manual buttonpress triggered the ExT to begin recording. Immediately following computer registration of the buttonpress, the Control port voltage was set high. The number of samples on the ExT record buffer prior to the +5V edge was then counted and multiplied by the sampling interval (10 μsec) to calculate the duration of the mouse registration delay. This design is presented schematically in Figure 5.

For tests of the parallel port, a design was adopted similar to the preliminary verification of the external time stamp conducted in Experiment 1. However, in this experiment, the time taken to read the status of the parallel port was included in the tests.

Software

Programming language. As in Experiment 1, the testing program was written and compiled in VB5.

Thread priority code. The thread priority was maximized with the same priority-class and priority-level procedures as those implemented in Experiment 1.

Mouse registration code and Procedure. The buttonpress of the mouse was coded in VB5 through the MouseDown subroutine. The stage of code executions leading to measurement of the delay was as follows. First, the ExT was configured to receive the TTL pulse from the voltage change across the mouse switch. The mouse button was then depressed, and the ExT commenced monitoring of the Control port. The code then cycled continuously while the ExT was recording. This code, as implemented, is shown below:

```
'--- Set up the ExT to begin recording on an external TTL trigger
DD1strig 1
DD1arm 1
'--- Wait for Mouse Button-Press then loop until recording is finished
Do While DD1status(1) <> 0
```

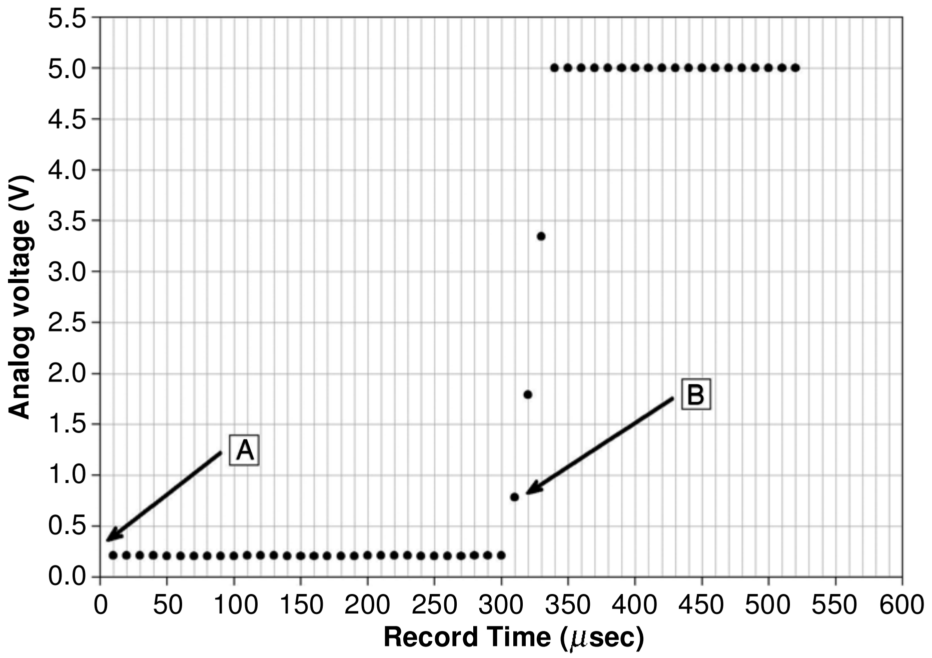

Figure 5. A hypothetical ExT output at 100 kHz, used to calculate the mouse registration delay. The TTL pulse resulting from the manual mouse buttonpress triggers the ExT to start recording at Point A. When the computer registers the buttonpress, the Control port is set high (+5 V) through the WriteDOLLx8 command. The delay between the buttonpress and the computer registration of the buttonpress is calculated as the time address at which the first evidence of a change in port status is apparent (Point B). Thus, in the above example, the registration delay would be 310 μsec.

```
        '--- Include DoEvents statement to allow the program to process
        the MouseDown subroutine
        DoEvents
Loop
```

The DoEvents command within the DD1Status(1) loop allowed the program to register the buttonpress of the mouse through the MouseDown subroutine. Immediately following registration of the mouse event, the Control port was switched to a high state. The resulting voltage edge was recorded on the ExT, providing a time stamp that signaled the termination of the registration delay period. A total of 1,000 trials were conducted on each button of each mouse, for each of the two test machines. All the mice were tested with the ball removed, but with the cursor active and the drivers loaded. During testing, all memory-resident programs, as visible in the Close Program dialog box, were closed, except for Explorer and the test program.

Parallel port registration code and Procedure. Registration of a response on the parallel port was detected with the ReadDOLLx8 function. Like the write operation, the read function is supported and declared through the *dollx8.dll* dynamic link library as

```
Declare Function ReadDOLLx8 Lib "dollx8.dll" (ByVal x8Port As Integer) As Integer
```

The x8Port argument of ReadDOLLx8 is assigned depending on which pin(s) are to be read. As with Experiment 1, we used the Control port and the ControlIOPort address of 890 as the x8Port argument. The procedure for estimating the time required to detect a change on the parallel port was as follows. The ExT initially began monitoring the voltage on the Control port. A buttonpress was then simulated by setting the Control port high. The first 0.5-V increment above 0 V was taken as the start stamp of the read delay period, thus including the rise time of the simulated buttonpress in the registration delay estimate. Once the status of the Control port had been retrieved, the voltage was returned to ground. The termination stamp of the read delay was taken as the first recorded sample after the start of the delay period at which the voltage on the Control port dropped more than 0.5 V below the maximum plateau (see Figure 6). The start and termination stamps for the read delay thus bordered, as closely as possible, the initiation of the simulated buttonpress with the termination of the port status retrieval accomplished by ReadDOLLx8. The calculated registration delay therefore represents the time from the closing of the switch on a parallel port button box to the registration of change on the parallel port through software.

To confirm that the voltage on the port had been switched by the time the ReadDOLLx8 function had executed, the status of the Control port was entered into an array on each trial and was output to file at the conclusion of testing. The primary code, as described and implemented, is presented below:

```
'--- Start recording on Control port
DD1go 1
'--- Simulate button press on Control port
WriteDOLLx8 ControlIOPort, 2
'--- Immediately read value on Control port and assign to variable
ResponseIO = ReadDOLLx8(ControlIOPort)
'--- Simulate button release on Control port
WriteDOLLx8 ControlIOPort, 3
```

A total of 100,000 trials were conducted on each machine. During data collection, all memory-resident programs, as visible in the Close Program dialog box, were closed, except for Explorer and the testing program.

Results and Discussion

Mouse Timing Results

Descriptive statistics for mouse registration delays in the 486 are presented in Tables 1 and 2, and for the PIII in Tables 3 and 4. Across both test machines and mouse

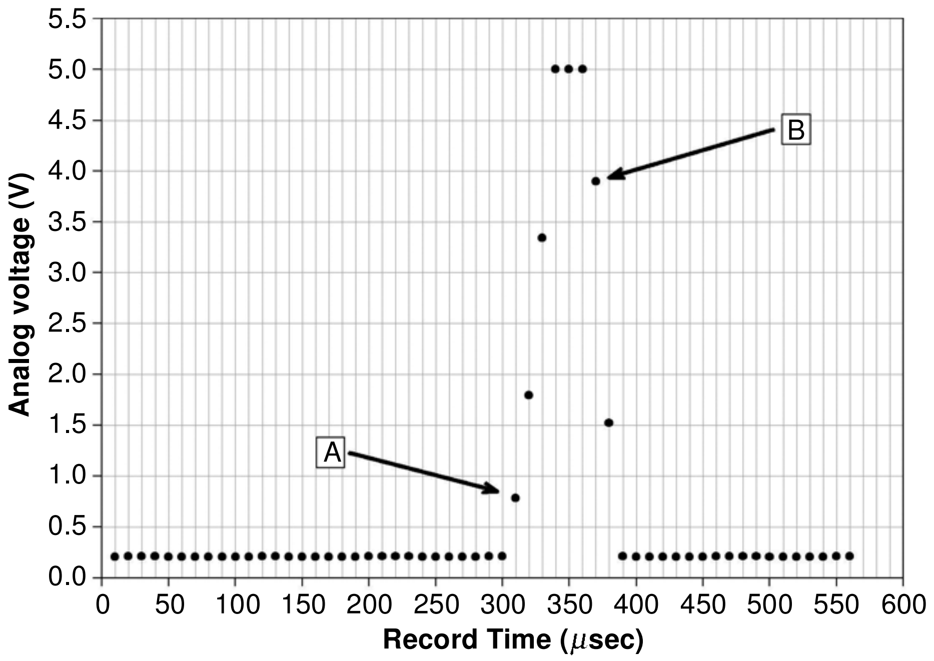

Figure 6. A hypothetical ExT output, similar in many respects to that presented in Figure 1, from which the read delay on the parallel port was calculated. Recording begins at time zero on the *x*-axis. At Point A, the simulated buttonpress is executed, setting the Control port high (+5 V). Once the voltage has reached the maximum plateau, the port status is read through the ReadDOLLx8 function. The second write execution returns the Control port to a LOW state, and the first evidence of a voltage reduction (Point B) signals the termination of the read delay period. The read delay is calculated as the time difference between Points A and B and, therefore, includes the rise time associated with the simulated buttonpress. In the above example, the delay is 370 μsec − 310 μsec, or 60 μsec.

buttons, the minimum and maximum registration delays for a serial mouse were 28.93 msec [right buttonpress, PIII, M-Serial 2.0A(2); see Table 4] and 46.60 msec [left buttonpress, 486, M-Serial 2.1A; see Table 1], respectively. Within serial mice, the maximum range of registration delays was 9.63 msec for the right button of the M-Serial 2.1A model in the 486 (see Table 1). The maximum variance of the registration delay across serial mice was 2.42 msec^2 for the left button of the M-Serial 2.0A(2) model in the 486 (see Table 1).

From the serial mouse results across both test machines, four general observations may be made. First, the registration delay is not constant across multiples of the same mouse model, with left and right buttonpresses of the M-Serial 2.0A(2) mouse registered consistently faster and with greater variability than the other 2.0A serial mice. Although nondistinctive in external appearance, M-Serial 2.0A(2) had a different FCC ID number and circuit board configuration from the remaining four 2.0A serial mice. Caution must therefore be exercised in drawing generalizations across mice with the same model number. Second, the presence of comparable registration delays in the alternative brand serial mouse indicates that the registration delay is not unique to Microsoft hardware.

Table 1
Mean, Maximum, Minimum, and Variance of Registration Delays for Left Mouse Buttonpresses on the 486 Test Computer, as a Function of the Various Models of Serial Mouse

Model	Mean Delay (msec)	Max. Delay (msec)	Min. Delay (msec)	Variance (msec^2)
M-Serial 2.0A (1)	41.33	42.96	39.57	0.38
M-Serial 2.0A (2)	34.07	37.99	30.41	2.42
M-Serial 2.0A (3)	41.21	43.17	39.62	0.39
M-Serial 2.0A (4)	40.96	42.53	39.38	0.35
M-Serial 2.0A (5)	41.25	42.98	39.78	0.36
M-Serial 2.1A	41.86	46.60	38.00	2.02
Serial Alt Brand	41.16	43.12	39.66	0.37

Table 2
Mean, Maximum, Minimum, and Variance of Registration Delays for Right Mouse Buttonpresses on the 486 Test Computer, as a Function of the Various Models of Serial Mouse

Model	Mean Delay (msec)	Max. Delay (msec)	Min. Delay (msec)	Variance (msec^2)
M-Serial 2.0A (1)	41.08	42.67	39.43	0.37
M-Serial 2.0A (2)	34.14	38.21	29.93	2.40
M-Serial 2.0A (3)	41.18	42.94	39.63	0.39
M-Serial 2.0A (4)	41.15	43.01	39.47	0.38
M-Serial 2.0A (5)	41.21	42.87	39.66	0.37
M-Serial 2.1A	41.34	45.76	36.13	1.96
Serial Alt Brand	41.22	43.18	39.58	0.39

Third, there appears to be little difference in the registration delays between left and right buttonpresses, with a maximum difference across all mice of 0.52 msec for M-Serial 2.1A in the 486 and an overall mean difference of 0.06 msec (*SD* = 0.24 msec). Fourth, the registration delay for serial mice (collapsed across both buttons) was consistently shorter in the PIII (*M* = 38.30 msec, *SD* = 2.57 msec) than in the 486 (*M* = 40.22 msec, *SD* = 2.6 msec), perhaps reflecting faster execution of the mouse driver. Overall, the serial mouse registration delays are similar to those reported under the MS-DOS operating system, with the exception that under Windows, there appears to be an added component to the delay of approximately 5–10 msec (see Beringer, 1992; Crosbie, 1990; Segalowitz & Graves, 1990).

Registration delays for the PS/2 mouse were substantially shorter and more variable than the delays associated with serial mice (see Tables 3 and 4). These results are consistent with the data reported by Segalowitz and Graves (1990) and Beringer (1992) and reflect the different registration detection mechanisms for PS/2, as compared with serial, hardware.

Parallel Port Timing Results

For the 486, the mean registration delay for the parallel port was 22.7 μsec (*SD* = 0.08 μsec), and for the PIII, it was a constant 10 μsec (*SD* = 0 μsec). In addition, the ReadDOLLx8 function was 100% successful in correctly detecting a change in the port status across all 200,000 trials. These results indicate that a button box connected to the parallel port is an ideal response device for RT experiments.

Recommendations

Across all the tested input devices, the parallel port appears to be the only apparatus capable of preserving the precision and resolution of the *timeGetTime* and *QueryPerformanceCounter* timers tested in Experiment 1. The added timing variance of the mice, however, is not substantial and can easily be corrected. For example, the highest input variance within serial mice was 2.42 msec^2. The loss of statistical power associated with added error variance rises as the proportion of added variance, relative to the remaining error variance, is increased (see Ulrich & Giray, 1989, for a review). Hence, even for a simple RT experiment with a low subject variance of 225 msec^2 (*SD* of 15 msec), the experimenter would need add only [2.42/225 × 100], or 1.08%, more independent observations to maintain statistical power. In an experiment with much higher subject variance, such as choice RT, the correction is so minor as to be ignored (e.g., 2.42/6400 × 100 = 0.04%). The serial mouse, with ball removed, can therefore be regarded as an adequate input device for single-interval RT experiments. If absolute RTs are required, individual testing of hardware is recommended so that the magnitude of the registration delay can be subtracted from obtained RTs.

The added error variance of the PS/2 mouse we tested is capable of substantially reducing statistical power in paradigms with low subject error variance. However, if

Table 3
Mean, Maximum, Minimum, and Variance of Registration Delays for Left Mouse Buttonpresses on the PIII Test Computer, as a Function of the Various Models of Serial Mouse

Model	Mean Delay (msec)	Max. Delay (msec)	Min. Delay (msec)	Variance (msec^2)
M-Serial 2.0A (1)	39.34	40.63	38.09	0.24
M-Serial 2.0A (2)	32.23	35.50	29.35	2.08
M-Serial 2.0A (3)	39.27	40.84	38.16	0.28
M-Serial 2.0A (4)	39.11	40.23	37.97	0.23
M-Serial 2.0A (5)	39.46	41.60	38.10	0.32
M-Serial 2.1A	39.74	43.48	35.96	1.85
Serial Alt Brand	39.38	40.88	38.11	0.31
PS/2	27.24	39.88	15.62	47.97

Table 4
Mean, Maximum, Minimum, and Variance of Registration Delays for Right Mouse Buttonpresses on the PIII Test Computer, as a Function of the Various Models of Serial Mouse

Model	Mean Delay (msec)	Max. Delay (msec)	Min. Delay (msec)	Variance (msec^2)
M-Serial 2.0A (1)	39.22	40.9	38.16	0.21
M-Serial 2.0A (2)	32.24	35.41	28.93	2.39
M-Serial 2.0A (3)	39.25	40.64	38.17	0.27
M-Serial 2.0A (4)	39.15	40.37	38.04	0.23
M-Serial 2.0A (5)	39.21	40.40	38.12	0.23
M-Serial 2.1A	39.32	43.01	35.56	1.78
Serial Alt Brand	39.28	40.42	38.02	0.26
PS/2	27.73	40.71	15.04	53.29

the predicted subject *SD* is equal to or less than 73 msec, the power correction for the PS/2 mouse we tested is only 1% or less. Consequently, even this input device need not seriously impact statistical comparisons. We recommend individual testing of PS/2 mice prior to testing, since only one model was examined in the present experiment.

CONCLUSIONS

The principal objective of the present study was to determine the achievable timing precision and resolution of the Windows operating system, the impact of various response devices on timing accuracy, and the nature of any interaction between timing accuracy and computer configuration. Across two experiments, our main results may be summarized as follows. First, the external timing tests of Experiment 1 disproved the assertions of Myors (1998, 1999) that Microsoft Windows is incapable of supporting reliable high-resolution timing. The disparity between our results and those of Myors perhaps highlights the importance of employing external chronometry in computer timing experiments, thus avoiding a sole reliance on internal timer consistency checks. In Experiment 1, both the *QueryPerformanceCounter* and the *timeGetTime* functions were found to support a baseline precision and resolution that is adequate for psychological research, when combined with methods that maximize the thread priority of the experimental program. In addition, no substantial difference was observed between the timing accuracy of a 486 test machine and that of a PIII. In Experiment 2, the parallel port was revealed as the most accurate device for receiving responses, with registration delays that preserve the precision and resolution of the timer across both test machines. Serial mice also performed consistently and can be regarded as adequate RT instruments under Windows. The most accurate combination of system and response timing is obtained by using the *QueryPerformanceCounter* timer with an input device on the parallel port.

REFERENCES

Beringer, J. (1992). Timing accuracy of mouse response registration on the IBM microcomputer family. *Behavior Research Methods, Instruments, & Computers*, **24**, 486-490.

Bovens, N., & Brysbaert, M. (1990). IBM PC/XT/AT and PS/2 Turbo Pascal timing with extended resolution. *Behavior Research Methods, Instruments, & Computers*, **22**, 332-334.

Creeger, C. P., Miller, K. F., & Paredes, D. R. (1990). Micromanaging time: Measuring and controlling timing errors in computer-controlled experiments. *Behavior Research Methods, Instruments, & Computers*, **22**, 34-79.

Crosbie, J. (1990). The Microsoft mouse as a multipurpose response device for the IBM PC/XT/AT. *Behavior Research Methods, Instruments, & Computers*, **22**, 305-316.

Dlhopolsky, J. G. (1988). C language functions for millisecond timing on the IBM PC. *Behavior Research Methods, Instruments, & Computers*, **20**, 560-565.

Emerson, P. L. (1988). TIMEX: A simple IBM AT C language timer with extended resolution. *Behavior Research Methods, Instruments, & Computers*, **20**, 566-572.

Graves, R., & Bradley, R. (1987). Millisecond interval timer and auditory reaction time programs for the IBM PC. *Behavior Research Methods, Instruments, & Computers*, **19**, 30-35.

Graves, R., & Bradley, R. (1988). More on millisecond timing and tachistoscope applications for the IBM PC. *Behavior Research Methods, Instruments, & Computers*, **20**, 408-412.

Hamm, J. P. (2001). Object-oriented millisecond timers for the PC. *Behavior Research Methods, Instruments, & Computers*, **33**, 532-539.

Heathcote, A. (1988). Screen control and timing routines for the IBM microcomputer family using a high-level language. *Behavior Research Methods, Instruments, & Computers*, **20**, 289-297.

McKinney, C. J., MacCormac, E. R., & Welsh-Bohmer, K. A. (1999). Hardware and software for tachistoscopy: How to make accurate measurements on any PC utilizing the Microsoft Windows operating system. *Behavior Research Methods, Instruments, & Computers*, **31**, 129-136.

Myors, B. (1998). A simple graphical technique for assessing timer accuracy of computer systems. *Behavior Research Methods, Instruments, & Computers*, **30**, 454-456.

Myors, B. (1999). Timing accuracy of PC programs running under DOS and Windows. *Behavior Research Methods, Instruments, & Computers*, **31**, 322-328.

Segalowitz, S. J., & Graves, R. E. (1990). Suitability of the IBM XT, AT, and PS/2 keyboard, mouse, and game port as response devices in reaction time paradigms. *Behavior Research Methods, Instruments, & Computers*, **22**, 283-289.

Ulrich, R., & Giray, M. (1989). Time resolution of clocks: Effects on reaction time measurement—good news for bad clocks. *British Journal of Mathematical & Statistical Psychology*, **42**, 1-12.

Warner, C. B., & Martin, M. K. (1999). eXpTools: A C++ class library for animation, tachistoscopic presentation, and response timing. *Behavior Research Methods, Instruments, & Computers*, **31**, 387-399.

NOTES

1. For further information on E-Prime and DMDX, the reader is directed to the Psychology Software Tools, Inc. Web site at http://www.pstnet.com/

E-Prime/e-prime.htm and the DMDX home page at http://www.u.arizona.edu/~kforster/dmdx.htm.

2. The *dollx8.dll* file is a dynamic link library (DLL) developed by AkiraSoft, and may be downloaded as shareware from the manufacturer's Web site (see http://www.audiotwister.com/more_dollx8.cfm for more information). According to the developers, the DLL is written in Assembly language and compiled in C++. Although fully functional under Windows 95/98, the DLL is not currently supported by Windows NT/2000/XP.

3. A constant 10-μsec delay indicates that the voltage was elevated above ground for only one sample before being reset by the next write statement. This probably indicates that the delay was consistently less than 10 μsec (and, therefore, beyond the resolution of the external timer), rather than equaling exactly 10 μsec each time. The true variance of the write delay is, in all likelihood, a nonzero value, but low enough to be unimportant in the present context.

4. For further details on processes, thread priority classes, and thread priority levels, the reader is directed to on-line information available at http://msdn.microsoft.com/library/en-us/com/htm/aptnthrd_8po3.asp.

5. The *timeGetTime* counter cycles every 2^{32} msec, or approximately 49 days and 17 h. The cyclical nature of the clock can present problems for timing measurements if the operating system runs continuously for longer than this time. If an initial time stamp were taken at the end of the clock cycle and the finishing stamp at the beginning of the next cycle, the magnitude of the calculated elapsed time would be erroneously large. A further problem is that the long integer data type in Visual Basic is signed and cannot store values higher than $2^{31}-1$ (total range spanning -2^{31} to $2^{31}-1$). Thus, if the operating system has been running continuously for longer than $2^{31}-1$ msec, or approximately 24 days and 20 h, the statement StartTime = timeGetTime() might fail, owing to either an overflow error or an arithmetic error. The simplest way to circumvent both these problems is to ensure that the Windows operating system is reinitialized at least once every 24 days, thus preventing rollover of the clock counter and confining the value returned by *timeGetTime* to the positive range of the signed long integer data type.

6. For further information on both timer functions used in the present experiment, the reader is directed to on-line information available at http://support.microsoft.com/support/kb/articles/Q172/3/38.asp.

7. Because it uses a 32-bit Windows API, the high-performance timer should be accessible from all 32-bit Windows interfaces, including Windows 95/98/2000/XP.

(Manuscript received September 25, 2001;
revision accepted for publication May 10, 2002.)