Behavior Research Methods, Instruments, & Computers
2001, 33 (4), 532-539

Object-oriented millisecond timers for the PC

JEFF P. HAMM
University of Auckland, Auckland, New Zealand

Object-oriented programming provides a useful structure for designing reusable code. Accurate millisecond timing is essential for many areas of research. With this in mind, this paper provides a Turbo Pascal unit containing an object-oriented millisecond timer. This approach allows for multiple timers to be running independently. The timers may also be set at different levels of temporal precision, such as 10^{-3} (milliseconds) or 10^{-5} sec. The object also is able to store the time of a flagged event for later examination without interrupting the ongoing timing operation.

Object-oriented programming allows one to develop utilities that may easily be reused for developing future applications. Object-oriented utilities, such as the timing functions presented here, can simplify code development and can make the debugging process faster. This allows for a shorter period between the idea and the experiment. Millisecond timing is a common requirement in many experiments. Accurate timing is required for both stimulus control and as a response measure.

Myors (1999) has shown that millisecond timing routines will be adversely affected when run under a multitasking operating system, such as the various incarnations of Windows. Myors recommends that experiments that require millisecond accuracy be run under a pure DOS environment. Although DOS is an antiquated operating system, there are some advantages for its continued use on data collection machines. Myors has pointed out the difficulties of obtaining reliable millisecond timing under multitasking operating systems such as Windows. Accurate timing underlies the reliable control of external equipment, stimuli, and response measurement, all of which are necessary for the conducting of well-controlled experiments. Until consistent and reliable timing routines are available under Windows and other multitasking operating systems, these operating systems will not be as well suited for data acquisition as DOS.

As an additional benefit, because DOS uses far fewer of the computer's resources than graphical/multitasking operating systems, older equipment may be employed for data acquisition. For example, in Wilson's 1996 paper, the controlling computer is an 8088 running at 12 mHz using software written in Turbo Pascal 3.0.

The ability to use antiquated equipment extends the useful life of laboratory purchases and allows for equipment stability over years of research. Additionally, multiple antiquated machines can usually be purchased for the price of a single new machine. This reduces the costs to a laboratory interested in setting up multiple experimental stations in order to reduce data acquisition time. The new and more expensive computers may then be dedicated to modern data analysis and visualization software packages, which often require up-to-date computer systems.

Use of older machines is a benefit only if the older machines can actually perform the task of controlling external equipment, stimulus presentation, and the recording of response data, with the level of temporal control sufficient for scientific interpretation. Although beyond the scope of this paper, under DOS the parallel port can be used to control and monitor external devices (Wilson, 1996), responses may be collected from the mouse, keyboard, or, with the least temporal variation, through the game port (Segalowitz & Graves, 1990), and a sound card may be used to detect/record vocal responses and to present auditory stimuli (Kello & Kawamoto, 1998). Visual stimuli can be presented on the CRT, though the timing of responses to these stimuli must consider the vertical position of the stimulus on the screen. This will be covered in more detail later.

Because the ability to monitor inputs and control external devices is not limited by use of DOS as the experimental operating system, the improved reliability of event timing under DOS over Windows (Myors, 1999) continues to make DOS the preferable operating system for experimental control. This paper will explain how to obtain millisecond timing accuracy under DOS and presents source code in the Appendix[1] that implements the method described.

With DOS, there are various methods of obtaining reliable millisecond accuracy on the XT/AT family of computers available. Smith and Puckett (1984; see Graves & Bradley, 1991) suggested an algorithm that was presented in assembler form by Graves and Bradley (1987, 1991). The basic idea is as follows. Read the time of day (TOD) counter and the corresponding residual counter (Ticks), and on the basis of the difference between two subsequent readings, millisecond accuracy may be achieved. Millisecond, and better, accuracy is possible because the TOD updates occur every 55 msec. Reading the TOD updates alone is sufficient only for 55-msec time resolution. How-

I thank V. K. Lim at the University of Melbourne, Australia, for her assistance in testing the routines provided. Additionally, I thank editor Jonathan Vaughn and four anonymous reviewers for their helpful comments. Correspondence should be addressed to J. Hamm, Department of Pschology, Private Bag 92019, University of Auckland, Auckland, New Zealand (e-mail: j.hamm@auckland.ac.nz).

ever, the TOD update occurs when a second countdown variable, the Ticks, has reached 0. At the point the Ticks reaches 0, the TOD count increments by 1, interrupt 1C fires, and the Ticks is set back to 65535, where it begins the countdown cycle again. As such, 55-msec bins of time are coded for by the TOD count, and 55/65356 (~0.0008392) msec are coded for each Tick.

The TOD update cycle is independent of computer processing speed, making this timing method highly portable and accurate. Timers based on these routines will work on all machines running under DOS since the XT. One caution with this technique is that if the TOD updates while reading the Ticks value, there is a potential for a $\pm$55-msec error (Bovens & Brysbaert, 1990) depending on the order in which the TOD and Ticks are read. A solution to this error was provided by Graves and Bradley (1991) and is employed in the current timing unit.

Although the TOD and Ticks update rates are independent of processing speed, the rate at which one may sample the TOD and Ticks values will improve with faster processors. In theory, it is possible to obtain a temporal resolution of 55/65536 msec, or ~0.0008392 msec, provided one may read both the TOD and Ticks value at least as fast as the tick countdown. This becomes important when attempting to set the resolution of the timers. Using the current routines on a 486 processor running at 66 mHz, the TOD and Ticks may be sampled every 1.8×10^{-5} seconds, whereas on a Pentium 200-mHz machine, the TOD and Ticks may be sampled at 0.6×10^{-5} sec. Both of these machines, therefore, are easily capable of reading the TOD and Ticks values fast enough for millisecond accuracy. The 486, however, is not quite capable of sampling these values fast enough to provide a resolution of 10^{-5} sec, whereas the Pentium is capable. However, the 486 sampling rate is sufficient for synchronizing millisecond response timing with stimulus presentation on a CRT. A demonstration of how to use multiple timers to perform this synchronization is presented later.

In addition, because the TOD count is reset to 0 at midnight, these timing routines will produce errors if they are started before and stopped after midnight. Adjustments may be made to account for the situation when the start TOD counts is greater than the stop TOD counts; however, this has not been included because the additional overhead of the conditional statement will decrease the timer sampling rate. For those who wish to include this adjustment, if the Stop TOD counts are less than the Start TOD counts, then subtract 1,573,040 from the Start TOD counts value. This is one more than the value of the TOD counter just prior to midnight; one more because the counter is reset to 0 so there is one more TOD update than the maximum value.

To obtain the TOD and Ticks value, the timing chip is set to Mode 2 (Bovens & Brysbaert, 1990). The TOD count is then read directly from its locations in Bios memory. The Bios segment is always \$40 (\$ indicates hexadecimal value) if programs are run in "real" mode, but the Bios segment may not be \$40 if programs are run in "protected" mode. Borland Pascal 7.0, which was used to write the unit presented here, allows for access to the Bios segment location in either real or protected mode through use of Seg0040. If this option is unavailable, add the line

```
Seg0040 = $40;
```

to the constant declaration segment of the unit, and use these timers only in real mode. Note, the version of Turbo Pascal (5.5) available for free download from the internet[2] compiles only in real mode.

With the continued need for millisecond accuracy obtainable under the DOS environment (Myors, 1999), a unit is provided in the Appendix that defines an object "ATimer" that may be incorporated into an experiment. The basics of the timing routine itself, described previously, are the same as those tested extensively under different operating systems by Myors (1999). As indicated in Myors's report, these should be used only under true DOS, and not in a DOS box running under any of the forms of Windows. Declaration of multiple variables of type "ATimer" allows for multiple timers of different temporal resolutions to be running simultaneously during an experiment.

The benefit of multiple timers with different temporal resolutions can be demonstrated in relation to the following example. Experiments that employ the CRT of a standard computer for the presentation of visual stimuli have to consider when in the refresh cycle of the CRT the stimulus actually appears. The CRT is updated from left to right and from top to bottom at a frequency of 60 Hz, or once approximately every 16.7 msec. Stimuli that are presented high on the CRT will appear slightly earlier than those that appear low on the CRT. In order to synchronize response timing with actual stimulus onset, it is necessary to first synchronize with the screen refresh cycle (Heathcote, 1988; routines that do this are supplied in the unit as the procedure WaitForTopOfScreen). Because of the constant refresh rate, it is possible to calculate the amount of time required for the raster to scan from the beginning of the refresh cycle to the horizontal row of the stimulus. A small delay may then be introduced to complete synchronization between stimulus onset and initiation of response timing.

For example, using a graphics mode with a screen resolution of 640×480, each horizontal line of the screen requires approximately 0.035 msec (16.7/480) to refresh. Because the left–right scan time is well below 1 msec, horizontal offsets are generally not of concern. However, stimuli that are vertically separated will have an appreciable onset asynchrony. Stimuli that are vertically separated by only 145 pixels will be presented just over 5 msec apart, with the higher stimuli presented sooner. The measuring of response time should begin at stimulus onset rather than at the beginning of the refresh cycle. This may be accomplished by using one timer object set at a resolution of 1/100000 sec to enforce a small delay after the detection of a new refresh cycle; these tiny delays may be incorporated in Listing 1.

Note that the screen refresh rate of 16.7 msec has been multiplied by 100 because of the higher temporal resolution of the delay timer. Furthermore, the topmost line of

Listing 1

```
DelayTimer.Init ;                                  { Initialize to the default 10−3 seconds }
DelayTimer.SetResolution (100000) ;                { change to 10−5 seconds }
RT_Timer.Init ;                                    { default 10−3 seconds }
{ Now calculate, in 10−5 seconds, the delay to get to the                   }
{ stimulus                                                                  }
DelayTime : = StimYCoor * 1670/ScreenYResolution;
WaitForTopOfScreen; {wait for beginning of screen refresh                   }
DelayTimer.Start ;                                 { Start timing the delay }
REPEAT UNTIL DelayTimer.GetTime >= DelayTime ;     { Wait }
RT_Timer.Start ;                                   { Now start the response timer }
REPEAT UNTIL ResponseMade ;                        { Wait for a response }
RT_Timer.Stop ;                                    { Stop the timer }
RemoveStimlus ;                                    { Clear the CRT }
ResponseTime := RT_Timer.GetLastTime ;             { Save response time }
DelayTimer.Done ;                                  { Shut down the timers }
RT_Timer.Done ;
```

the screen is assumed to have a *y*-coordinate of 0. A higher resolution timer is useful in this situation in order to reduce the synchronization error between stimulus onset and the starting of the response timer to be less than the resolution of the response time measurement. As noted earlier, this second high-resolution timer can be obtained with sufficient resolution for this purpose on a 486 machine running at 66 mHz. The response time will be measured in the default units of milliseconds because setting the resolution of the delay timer object does not affect the second response timer object. Even though both timers operate on the same internal clock, because of the object-oriented approach employed, the timers function independently.

Finally, for the experimental researcher the main benefit of the object-oriented approach of programming is primarily one of pragmatics. When reliable routines have been developed, encapsulating them in an object structure, such as the msTimer object presented in the Appendix, development time of new experimental programs can be greatly reduced as the function of the object may be referred to through logically informative method calls. Provided one is careful to address only internal variables of the object through method calls, bug-free code may be used with confidence from one program to the next. The use of well-named methods that indicate the function of the routine, such as Timer.Start, improve the readability of the code, easing modification for follow-up experiments. With a well-tested library of only few objects—for example, one that monitors the game port for input responses, one that controls the parallel port for device control and monitoring, one that monitors the sound card for microphone and sound generation, and one that provides millisecond timing routines—programming in a high-level language such as Pascal need be no more complicated than implementation of the script language provided with many commercial experimental software packages. An additional benefit beyond that of commercial packages is that experimenters can always modify and recompile their source code to account for the latest developments or to produce code that suits their specific and novel experimental needs while gaining an appreciable knowledge as to the workings of one of their most common research tools.

REFERENCES

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.

Graves, R. E., & 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. E., & Bradley, R. (1991). Millisecond timing on the IBM PC/XT/AT and PS/2: A review of the options and corrections for the Graves and Bradley algorithm. *Behavior Research Methods, Instruments, & Computers*, **23**, 377-379.

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.

Kello, C. T., & Kawamoto, A. H. (1998). Runword: An IBM-PC software package for the collection and acoustic analysis of speeded naming responses. *Behavior Research Methods, Instruments, & Computers*, **30**, 371-383.

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.

Smith, B., & Puckett, T. (1984, April). Life in the fast lane. *PC Technical Journal*, pp. 63-74.

Wilson, W. J. (1996). The ϕ-maze: A versatile automated T-maze for learning and memory experiments in the rat. *Behavior Research Methods, Instruments, & Computers*, **28**, 360-364.

NOTES

1. The source code, in text file format, is also available for free download on the World-Wide Web at http://www.psych.auckland.ac.nz/psych/research/jeff

2. Turbo Pascal 5.5, which supports object-oriented programming, is, at the time of writing, available for free from Borland website: www.borland.com by following the Community/Museum link to find Turbo Pascal 5.5.

APPENDIX

```
UNIT msTime ;
{$q-,r-}
INTERFACE
TYPE ATimer = OBJECT
    xPerSecond,
    NumCaught,
    StoredTODCalls,
    StoredTicks,
    StartTODCalls,
    StopTODCalls,
    StartTicks,
    StopTicks : LONGINT;
    TimeResolution : DOUBLE;
    NoneStored : BOOLEAN;
    CONSTRUCTOR Init ;
    DESTRUCTOR Done ;
    PROCEDURE Start ;
    PROCEDURE Stop ;
    PROCEDURE TakeReading (VAR TODCalls,Ticks: LONGINT) ;
    PROCEDURE SetResolution (ixPerSec : LONGINT) ;
    PROCEDURE Store (OnFirstTrue : BOOLEAN) ;
    FUNCTION GetTime : LONGINT ;
    FUNCTION GetLastTime : LONGINT ;
    FUNCTION GetStoredTime: LONGINT ;
    FUNCTION NumRepeats : LONGINT ;
    END;
FUNCTION NumberOfTimers : INTEGER;
PROCEDURE Calibrate (AtRes: LONGINT) ;
PROCEDURE WaitForTopOfScreen ;
IMPLEMENTATION
CONST TimerCt1          = $43;{$xx indicates xx is a hexidecimal value}
    Timer0              = $40;
    TimerSet            = $34;
    TimerReset          = $36;
    TimerLatch          = $00;
    TODOffset           = $6c;
    CountsPerSec        = 1193182;
    CountsPerTODCall    = 65536;
VAR NumTimersActive : LONGINT;
    ExitSave: POINTER;
{ Procedures and Functions ---------- }
PROCEDURE WaitForTopOfScreen ;
{ This procedure detects the beginning of a refresh cycle of the CRT }
{ It should be used to synchronize response timing with presentations }
{ in conjunction with a high-resolution timer }
BEGIN
    REPEAT UNTIL PORT[MEMW[Seg0040:$63]+6] AND 8 = 8;
END;
{ ---------- SetTimerMode2 ---------- }
PROCEDURE SetTimerMode2 ;
{ Sets the timer chip into Mode 2, which allows the accessing of the }
{ TOD countdown values. }
VAR TempTOD : LONGINT;
BEGIN
    TempTOD := MEML[Seg0040:TODOffset];
    REPEAT UNTIL TempTOD <> MEML[Seg0040:TODOffset];{Wait for update }
    PORT [TimerCt1] := TimerSet;                    {Set Mode 2 }
    PORT [Timer0] := 0;
    PORT [Timer0]:= 0;
END;
{ ----------SetTimerMode3code ---------- }
```

APPENDIX (Continued)

```
PROCEDURE SetTimerMode3 ;
{ Returns the timer chip to Mode 3, its normal setting                                    }
VAR TempTOD : LONGINT;
BEGIN
    TempTOD := MEML[Seg0040:TODOffset];
    REPEAT UNTIL TempTOD <> MEML[Seg0040:TODOffset];{Wait for update                      }
    PORT[TimerCT1] := TimerReset ;                  { Set Mode 3                          }
    PORT[Timer0] := 0;
    PORT[Timer0] := 0;
END;
{ ----------------EmergencyShutDown ---------------------------------------------------- }
PROCEDURE EmergencyShutDown ; FAR;
{ Ensures that the timer chip is returned to Mode3 upon exiting the                       }
{ program                                                                                 }
BEGIN
    ExitProc := ExitSave;
    SetTimerMode3;
END;
{ --------------------GetTicks ------------------------------------------------------------ }
PROCEDURE GetTicks (VAR NTicks: WORD);
var Temp : RECORD                               { This data structure is used to      }
                CASE Byte OF                    { as a simple quick way to combine    }
                0 : (b1,b2: BYTE);              { two 1-byte values into              }
                1 : (w: WORD);                  { a 2-byte unsigned value             }
END;
BEGIN
    PORT [TimerCt1] := TimerLatch;              { Store the tickes                    }
    Temp.b1:= PORT[Timer0];                     { Read the low byte                   }
    Temp.b2:= PORT[Timer0];                     { Read the high byte                  }
    NTicks:= Temp.W;                            { Return unsighted 2-byte value       }
END;
{ ----------------Calibrate ------------------------------------------------------------ }
PROCEDURE Calibrate (AtRes: LONGINT);
{ This procedure provides a simple test to determine if a given                           }
{ temporal resolution may be achieved. The output provides two                            }
{ values of major interest. The first is the Reading time, which                          }
{ is the number of units of time that pass between subsequent calls to                    }
{ sampling the timer. (AtRes = 1,000 makes the units milliseconds)                        }
{ The sampling rate is a more conservative value because it also                          }
{ includes processing time for the various FOR Loops and other                            }
{ operations not associated with the reading of the Timer object                          }
{ Sampling and/or Reading rates less than 1 indicate that these values                    }
{ may be read fast enough to provide timing information at the given                      }
{ temporal resolution                                                                     }
VAR Sam,Tods,
    Time,
    Ticks : LONGINT;
    T1,T2 : WORD;
    L1,L2 : WORD;
    Ave : DOUBLE;
    t: ATimer;
BEGIN
    Ave := 0.0;                                 { Initialize variable                 }
    t.Init ;                                    { Resolution will be milliseconds     }
    t.SetResolution (AtRes) ;                   { Return time at the given resolution }
    t.Start ;                                   { Start the timer                     }
    FOR L1 := 1 TO 1000 DO                      { Do 1,000 repetitions                }
BEGIN
    IF L1 MOD 100 = 1 THEN                      { To indicate how far along we are    }
    WRITE ('.');
    Sam := 0;                                   { Starting a new repetition           }
    FOR L2 := 1 TO 1000 DO                      { 1,000 readings                      }
```

APPENDIX (Continued)

```
BEGIN
    GetTicks (T1);                                  { Use the ticks to calculate time          }
    Time := t.GetTime;                              { Now read the timer                       }
    GetTicks (T2);                                  { To determine how many tics passed        }
    IF T2 < T1 THEN                                 { If it's not wrapped around               }
BEGIN
    Sam := Sam + (T1 - T2);                         { Sum the number of tics                   }
END
ELSE
BEGIN
    Sam := Sam + (T1 + NOT WORD(T2)) + 1;           { Number of tics for wrap                  }
                                                    { around                                   }
    END;
END;
Ave := Ave + Sam/1000;                              { Sum average number of ticks for reading  }
END;
t.Stop ;
Ave := Ave / 1000;                                  { Mean of 1,000 average ticks              }
                                                    { Now write the information to the screen  }
WRITELN ('Readings take ',Ave*AtRes/CountsPerSec:0:6,' units');
WRITELN (t.NumRepeats,' double readings were necessary in 1,000,000 samples');
WRITELN ('Sampling rate: ',t.GetLastTime/(1000*1000):0:6,' units');
t.Done;                                             { Shut down the timer                      }
END;
{ ----------------NumberOfTimers ---------------------------------------------------------------- }
FUNCTION NumberOfTimers : INTEGER;
{ Primarily for diagnostic purposes. Every timer that is initialized                             }
{ is counted, and every timer that is shut down is removed from the                              }
{ count. At the end of a program, this should return 0                                           }
BEGIN
    NumberOfTimers := msTime.NumTimersActive;
END;
{ ------------------------------------------------------------------------------------------------ }
{ ------------ Methods for ATimer ------------------------------------------------------------- }
{ ------------------------------------------------------------------------------------------------ }
{ -------------------ATimer.Init ------------------------------------------------------------- }
CONSTRUCTOR ATimer.Init;
{ This call must be made prior to use of subsequent methods                                      }
{ It is used to initialize internal variables and should be                                      }
{ called only once. However, if a call is made to the                                            }
{ Destructor Done, then Init must be called again prior to                                       }
{ reuse.                                                                                         }
BEGIN
    NumCaught : = 0;                                { Initialize internal variable             }
    SetResolution (1000);                           { Default is milliseconds                  }
    INC (NumTimersActive);                          { Count the timer as active                }
    NoneStored : = TRUE;                            { Initialize internal variable             }
END;
{ ----------------ATimer.Done ------------------------------------------------------------------- }
DESTRUCTOR ATimer.Done;
{ The destructor cleans up the timer and decrements the global timer                             }
{ count. The global timer count, NumTimersActive, can be used during                             }
{ program development to ensure that the timer objects are properly                              }
{ initialized and shut down. When all timers have been deactivated,                              }
{ NumTimersActive equals 0.                                                                      }
BEGIN
    DEC (NumTimersActive);                          { Produces faster code than x := x - 1;    }
END;
{ -------------------ATimer.SetResolution ---------------------------------------------------- }
```

APPENDIX (Continued)

```
PROCEDURE ATimer.SetResolution (ixPerSec: LONGINT);
{  This sets the timer's GetTime (and related functions) to return at                    }
{  the specified resolution. The units are 1 sec divided by the                          }
{  passed value. For example SetResolution (1,000) returns milliseconds                  }
{  and SetResolution (10,000) returns tenths of a millisecond. This only                 }
{  needs to be called directly if units other than milliseconds are                      }
{  desired.                                                                              }
BEGIN
    xPerSecond  := ixPerSec;
    TimeResolution  := xPerSecond/CountsPerSec;
END;
{  ---------- ATimer.NumRepeats ----------  ----------}
FUNCTION ATimer.NumRepeats  :  LONGINT;
{  Returns the number of double samples in Taking readings to                            }
{  avoid the ± 55-msec error.                                                            }
{  This function is primarily for diagnostic purposes.                                   }
BEGIN
    NumRepeats  : =  NumCaught;
END;
{  ---------- ATimer.TakeReading ----------  ----------}
PROCEDURE ATimer.TakeReading (VAR TODCalls,Ticks: LONGINT);
{  This reads the current TOD value, and the countdown value                             }
{  The TOD is read twice, and if the value has changed, then                             }
{  The values are resampled, otherwise there is the potential                            }
{  of a ± 55 ms error; see Bovens and Brysbaert, 1990                                    }
VAR Loop : INTEGER;
    t1,t2: LONGINT;
    Temp: record
                        case byte of                    {  This structure is to convert  }
                        0: (b1,b2: byte);               {  Two single byte values        }
                        1: (w  : WORD);                 {  to 1 unsigned 2-byte value    }
    END;
BEGIN
    Loop:= -1;          {  First time through this increments to 0                       }
REPEAT                  {  and if it must repeat, gets incremented to 1                  }
    t1  := MEML[Seg0040:TODOffset];     {  Get TOD updates                               }
    PORT [TimerCt1]  : = TimerLatch;    {  Store the current ticks                       }
    Temp.b1  : = PORT[Timer0];          {  Read the low byte of ticks                    }
    Temp.b2  : = PORT[Timer0];          {  Read the high byte of ticks                   }
    t2  : = MEML[Seg0040:TODOffset];    {  Resample TOD                                  }
    INC (Loop);
UNTIL  (t2 = t1);       {  Loop if a TOD has occured                                     }
Ticks  := Temp.w;       {  This combines t.b1 and t.b2 into a WORD                       }
TODCalls  := t1;
INC  (NumCaught,Loop); {  For diagnositic purposes                                       }
END;
{  ---------- ATimer.Start ----------  }
PROCEDURE ATimer.Start  ;
{  Takes the initial readings of the TOD and Ticks and clears the                        }
{  storage flag.                                                                         }
BEGIN
    TakeReading (StartTODCalls,StartTicks);
    NoneStored  := TRUE;
END;
{  ---------- ATimer.Stop ----------  ----------}
PROCEDURE ATimer.Stop;
{  This doesn't actually stop the timer per se, but the time between                     }
{  Start and Stop calls may be retrieved from a call to                                  }
{  Timer.GetLastTime. Note, Stop is called from the GetTime routine, so                  }
{  GetLastTime will reflect the time between Start and Stop or                           }
{  GetTime, which ever of the latter occurred most recently.                             }
BEGIN
    TakeReading (StopTODCalls,StopTicks);
END;
```

APPENDIX (Continued)

```
{ -----------------ATimer.Store --------------------------------------------------------------------------------------------------- }
PROCEDURE ATimer.Store (OnFirstTrue: BOOLEAN);
{ The time at which OnFirstTrue initially becomes True is stored. To          }
{ use reuse store, it must be flushed by a call to GetStoredTime or           }
{ by a new call to Start.                                                     }
BEGIN
IF (OnFirstTrue) AND (NoneStored) THEN
BEGIN
   TakeReading (StoredTODCalls,StoredTicks);
   NoneStored:= FALSE;                       { Prevents overwrites of Stored value      }
   END;
END;
{ --------------------ATimer.GetStoredTime ---------------------------------------------------------------------------------------}
FUNCTION ATimer.GetStoredTime : LONGINT;
{ Returns the time stored via Store. If no time was stored returns − 1       }
{ and clears the storage flag to allow for a new time to be stored           }
VAR NumTicks: LONGINT;
BEGIN
   IF NOT (NoneStored) THEN
BEGIN
   NumTicks := StartTicks − StoredTicks;
   GetStoredTime := TRUNC ((((StoredTODCalls-
                      StartTODCalls) * CountsPerTodCall)+
                      NumTicks) * TimeResolution);
   NoneStored := TRUE;       { Allows for a new value to be stored                 }
END
ELSE
BEGIN
   GetStoredTime := − 1;                     { no value stored                          }
   END;
END;
{ -------------- ATimer.GetTime ----------------------------------------------------------------------------------------------- }
FUNCTION ATimer.GetTime: LONGINT;
{ This returns the amount of time since the timer was started                }
VAR NumTicks : LONGINT;
BEGIN
   Stop;              { Loads values into StopTODCalls and StopTicks                  }
   NumTicks := StartTicks − StopTicks;
   GetTime := TRUNC ((((StopTODCalls−
                      StartTODCalls) * CountsPerTodCall)+
                      NumTicks) * TimeResolution);
END;
{ -----------------ATimer.GetLastTime ------------------------------------------------------------------------------------------ }
FUNCTION ATimer.GetLastTime : LONGINT;
{ This returns the time at the last GetTime or Stop call                     }
{ If neither have been called, then the result is meaningless               }
VAR NumTicks : LONGINT;
BEGIN
   NumTicks := StartTicks − StopTicks;
   GetLastTime := TRUNC ((((StopTODCalls-
                      StartTODCalls)*CountsPerTodCall)+
                      NumTicks) * TimeResolution);
END;
{ -------------- Unit Initialization Code ------------------------------------------------------------------------------------ }
BEGIN
   msTime.ExitSave : = ExitProc;             { Ensures the return to Mode 3             }
   ExitProc := @EmergencyShutDown;           { Links in the Exit procedure              }
   msTime.NumTimersActive := 0;              { Initialize global variable               }
   SetTimerMode2 ;                           { Ensures the timer chip is in Mode 2      }
END.
```

(Manuscript received July 16, 2000;
revision accepted for publication July 3, 2001.)