Behavior Research Methods, Instruments, & Computers
1988, 20 (3), 289-297

METHODS & DESIGNS

Screen control and timing routines for the IBM microcomputer family using a high-level language

ANDREW HEATHCOTE
Queen's University at Kingston, Kingston, Ontario, Canada

Procedures are described for the IBM PC/AT and compatibles that measure an event's duration with millisecond accuracy and that synchronize stimulus presentation with the vertical-retrace signal. The software is written in Turbo Pascal (Versions 2.0, 3.0, and 4.0; Borland International, Inc., 1984, 1985, 1987). Difficulties reflecting differences among video-controller cards are also presented.

Most of the routine housekeeping requirements of experiments requiring a subject to make a choice—selection of stimuli, randomization of trials, and the like—can be met by an IBM PC/AT or compatible using a high-level language under MS-DOS. Exact timing, however, is difficult. MS-DOS does not include a standard method by which to measure time to millisecond accuracy. Timing procedures included in the operating system, and hence in languages supported by the operating system, do not provide millisecond accuracy. Moreover, MS-DOS does not include a standard method by which to control the exact onset of displays written to the screen. When material is written to the screen, it may not appear on the screen immediately. The lag depends on the rate at which the video adapter refreshes the screen and on the time in the refresh cycle at which the write command was given. The lag can vary from 0 to 20 msec. If a subject's reaction time is measured by starting a clock immediately after a write statement and stopping it when the subject pushes a button, the measurement will include a random value reflecting the lag between the nominal onset of the material (the write statement) and its actual appearance on the screen.

In many situations, an error of 20 msec is unimportant. Inasmuch as most experiments involve comparisons among conditions, a constant error of 20 msec could be ignored without difficulty. Because the error is random, however, when latencies are averaged across subjects or trials, the error adds variance to the data. To eliminate that variance, that is, to force the same lag on all occasions, the user must synchronize the write operation with the video system's refresh cycle.

The author would like to thank Victor Lee, Andrew Martin, and Stephen Popiel for help in developing these programs. Special thanks to Doug Mewhort and Stephen Popiel for comments on earlier drafts of this paper, and to two anonymous reviewers for helpful suggestions. The work was supported by a Commonwealth Fellowship to the author and by Grant AP-318 from the Natural Sciences and Engineering Research Council of Canada to D. J. K. Mewhort. Reprints may be obtained from the author at the Department of Psychology, Queen's University at Kingston, Kingston, Ontario K7L 3N6, Canada.

Timing

Channel 0 of the Intel 8253 16-bit timer/counter circuit is used to maintain the system's time of day. A 16-bit counter is incremented at a frequency determined by the output of channel 0 (1.1931817 MHz). When the counter reaches its maximum value (65,535), interrupt 8 is called, and the counter resets to zero. Interrupt 8 updates the system's time of day. An update normally occurs every 55 msec. To increase the resolution of the time-of-day clock, the frequency of the time-of-day interrupt must be increased. Fortunately, the value at which the counter initiates the timing interrupt is programmable. It can, therefore, be decreased to obtain the resolution required.

Interrupt 8 does not terminate with a return from interrupt instruction, as is usual, but invokes interrupt 1CH. Interrupt 1CH is serviced by a single return-from-interrupt instruction. It allows the user a point of access for his/her own interrupt routines. When an interrupt occurs, the interrupt handler looks up the starting address of the routine that services the interrupt in the interrupt-vector table. The address in the table can be altered so that a jump will occur to the beginning of code supplied by the user. To create a software clock based on the time-of-day interrupt, the jump address for interrupt 1CH can be changed to point to code that increments a counter or counters. When combined with the appropriate change in frequency of the time-of-day interrupt, this strategy will create an interrupt-driven software clock with the desired resolution. There is a hidden cost, however: the operating system's time-of-day function will be incremented at approximately 55 times its normal rate. To correct the increase in the system's time-of-day value, it must be reset after the clock's interrupt frequency is returned to normal.

Bührer, Sparrer, and Weitkunat (1987) described assembly-language routines with which to build the software clock described above. The same clock can be built using the extensions to Pascal included in Borland International's (1984, 1985, 1987) Turbo Pascal (Versions 2.0, 3.0, and 4.0). Use of Pascal avoids the need to link the assembler code to the high-level language program controlling the overall experiment. Moreover, Pascal is easier to understand than the assembler version, and it is simpler, in that housekeeping functions—such as saving and restoring the CPU registers when calling system interrupts—are performed automatically. Listing 1 shows how to build the clock in Turbo Pascal. The code for Versions 2.0 and 3.0 of Turbo Pascal is identical, but some changes are required for Version 4.0. The changes required are included as comments in the listing.

The code shown in Listing 1 follows the strategy outlined earlier. The procedure Interrupt_Handler will be executed each time interrupt 1CH occurs. The Turbo Pascal procedure Inline is used to enter assembler instructions (represented as hexadecimal numbers) directly into the program. Interrupt_Handler begins by saving the current values of the CPU registers, and then stores the address of Turbo Pascal's data segment in register DS. The address is directly poked into the Inline code by the procedure Timer_On, replacing the pair of dummy instructions $00/$00/. Turbo Pascal assumes that DS contains the base address of its data segment. However, DS is not preserved during the time-of-day interrupt. Therefore, the correct value must be restored by Interrupt_Handler each time it is called. The final instruction in the first call of Inline disables all maskable interrupts. The body of Interrupt_Handler increments an integer counter and can be altered as required. In particular, nested integer counters permit the user to avoid the upper limit of a single 16-bit (integer) counter. Finally, Inline is used to restore the CPU registers, to enable interrupts, and to return from the interrupt.

Version 4.0 Turbo Pascal allows interrupt handlers to be written easily by providing the compiler instruction Interrupt. The instruction Interrupt is placed after a procedure declaration and tells the compiler that the procedure will be used as an interrupt handler. The compiler automatically generates code to save and restore the registers and to store the data segment address in the DS register.

LISTING 1
Timer.Inc

```
{ Uses          VERSION 4: Functions used in this file are included
    Crt;        in separate   units which must be linked with the
    Dos;        statements on the left.}

CONST
   HiFast = $4;                 {Hi byte for 1000 Hz interrupt frequency}
   LoFast = $A9;                {Lo byte for 1000 Hz interrupt frequency}

TYPE
   registers = record                                      {CPU registers}
      case integer of
         0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer);
         1: (AL,AH,BL,BH,CL,CH,DL,DH         : Byte);
      end;
{VERSION 4: This Type is already declared in the DOS unit}


VAR
    Regs         : registers;                        {Used for interrupts}
    Seg1C,Ofs1C,                        {Seg and Offset of interrupt vector 1CH}
    HM,SCS,                                {Time: Hour+Minute, Sec.+CentiSec.}

    Year,MD,                                      {Date: Year, Month+Day }
    t            : integer;                   {General purpose ms. counter}


PROCEDURE INTERRUPT_HANDLER;

{Increments variable 't' when it services interrupt 1CH
 (dummy interrupt normally serviced by a single IRET
 to allow the user access to the clock)}

Begin
   Inline($50/$53/$51/$52/                          {PUSH AX,BX,CX,DX}
          $56/$57/$1E/$06/$55/                   {PUSH SI,DI,DS,ES,BP}
          $B8/$00/$00/                     {MOV AX,turbo dseg address}
          $50/                                              {PUSH AX}
          $1F/                                               {POP DS}
          $FA);                                                 {CLI}
   t := t+1;                                      {Increment Counter}
   Inline($5D/$07/$1F/$5F/$5E/                 {POP BP,ES,DS,DI,SI}
          $5A/$59/$5B/$58/                        {POP DX,CX,BX,AX}
          $FB/                                                  {STI}
          $CF);                                                {IRET}
End;

{VERSION 4: A special procedure type is provided for writing interrupt
            handlers. Registers are saved and returned and the address
            of the data segment stored in DS automatically. Only the
            disabling and enabling of maskable interrupts need be
            carried out using the "Inline" function.
```

LISTING 1 (Continued)

```
PROCEDURE INTERRUPT_HANDLER;

INTERRUPT;

Begin
   Inline($FA);
   t := t+1;
   Inline($FB);
End;}

PROCEDURE CHANGE_TIMING_FREQUENCY(HiByte,LoByte: byte);


{Changes the divisor which determines the frequency of the
 time of day interrupt. When the procedure parameters are
 set to 0,0 this gives the normal frequency of 18.2 Hz}

Begin
   port[$43] := $36;              {Select Mode 3 of timer Channel 0}
   port[$40] := LoByte;             {Output low byte of new divisor}
   port[$40] := HiByte;            {Output high byte of new divisor}
End;


PROCEDURE GET_TIME_AND_DATE(var Year,MonDay,HrMin,SecCentiS: Integer);

{Stores current time of day and date}

Begin
   with Regs do
   Begin
      AX := $2A00;                       {Interrupt 21, function 2C}
      MsDos(Regs);                                 {Call interrupt}
      Year := CX;                         {Store current date: years}
      MonDay := DX;               {Store current date: month and day}
      AX := $2C00;                       {Interrupt 21, function 2C}
      MsDos(Regs);                                 {Call interrupt}
      HrMin := CX;            {Store current time: hours and minutes}
      SecCentiS := DX;  {Store current time: seconds and centiseconds}
   End;
End;

PROCEDURE TIMER_ON;

{Stores the old interrupt vector for 1CH and resets it
 to the address of procedure INTERRUPT_HANDLER. The
 current time is then recorded and the interrupt
 frequency increased to approximately 1000 Hz}

Begin
   MemW[Cseg:Ofs(Interrupt_Handler)+17] := Dseg;
   {Store current turbo data segment address in the Interrupt_Handler
    VERSION 4: the data segment address is stored by the interrupt
    handler procedure. The above line of code MUST NOT be included if
    the Version 4 Interrupt_Handler, described above, is used.}

   with Regs do
   Begin
      AX := $351C;               {Interrupt 21, function 35, on 1CH}
      MsDos(Regs);                   {Gets current interrupt vector}
      Seg1C := ES;                 {Save current segment and offset}
      Ofs1C := BX;
      DS := Cseg;                 {Interrupt_Handler in code segment}
      DX := Ofs(Interrupt_Handler)+7; {Offset of first executable code}
      {VERSION 4: The 7 byte offset is no longer necessary
                i.e., DX := Ofs(Interrupt_Handler);}

      AX := $251C;               {Interrupt 21, function 25, on 1CH}
      MsDos(Regs);                   {Set DS:DX as interrupt vector}
   End;
   Get_Time_and_Date(Year,MD,HM,SCS);
   Change_Timing_Frequency(HiFast,LoFast);
End;


PROCEDURE RESET_TIME;


{Resets time of day clock to correct time. Doesn't take account of
 millennium leap year rule or increases in time greater than one
 year}

Const
   NoDays : array[1..2,1..13] of integer
          = ((0,31,59,90,120,151,181,212,243,273,304,334,365),
             (0,31,60,91,121,152,182,213,244,274,305,335,366));
   {Total days to the beginning of the month corresponding to the
    second array index.  First index: 1 = normal, 2 = leap year}

Var
   YType,Y2Type: 1..2;      {Year Type: 1 = normal year, 2 = leap year}
   DifCS,DifS,DifMi,              {Components of Diff: cs, s and min}
   DifH,DifD,                     {Components of Diff: hrs and days}
   x,Days,                            {Used to calculate days/months}
   HM2,SCS2,               {Fast Time: Hour+Minute, Sec.+CentiSec.}
   Year2,MD2   : Integer;              {Fast Date: Year, Month+Day}
```

LISTING 1 (Continued)

```
   Hrs,Hrs2,                                              {Number of hours}
   Diff        : Real;          {Difference between fast time and the
                                  time recorded by Get_Time, in hours}
Begin
   Get_Time_and_Date(Year2,MD2,HM2,SCS2);

   {Determine whether leap years have occurred}

   If (Year mod 4) = 0 then YType := 2                       {Leap Year}
   else YType := 1;
   If (Year2 mod 4) = 0 then Y2Type := 2                     {Leap Year}
   else Y2Type := 1;

   {Calculate the difference between the fast date/time and the date
    /time recorded by the first call of Get_Time_and_Date, in hours}

   Diff := (Year2 - Year) * NoDays[YType,13] * 24;
   Hrs := (NoDays[YType,Hi(MD)]*24) + (Lo(MD)*24) + Hi(HM)
          + (Lo(HM)/60) + (Hi(SCS)/3600) + (Lo(SCS)/360000.0);
   Hrs2 := (NoDays[Y2Type,Hi(MD2)]*24) + (Lo(MD2)*24) + Hi(HM2)
           + (Lo(HM2)/60) + (Hi(SCS2)/3600) + (Lo(SCS2)/360000.0);
   Diff := (Diff + (Hrs2 - Hrs))/54.9328;

   {Diff divided by the acceleration factor to give the true period
    of time that has passed. Now convert into day/time scales}

   DifCS := Round(Frac(Frac(Diff)*3600.0)*100.0);
   DifS := Trunc(Frac(Diff)*3600) mod 60;
   DifMi := Trunc(Frac(Diff)*60);
   DifH := Trunc(Diff) mod 24;
   DifD := Trunc(Diff) div 24;

   {Calculate correct time and reset system time accordingly}

   SCS := SCS + DifCS;
   SCS := SCS + (Lo(SCS) div 100)*156;
   SCS := SCS + 256*DifS;
   HM := HM + DifMi + (Hi(SCS) div 60);
   SCS := SCS - (Hi(SCS) div 60)*15360;
   HM := HM + (Lo(HM) div 60)*196;
   HM := HM + DifH*256;
   Days := (Hi(HM) div 24) + DifD + Lo(MD) + NoDays[YType,Hi(MD)];
   HM := HM - (Hi(HM) div 24)*6144;
   with Regs do
   Begin
      AX := $2D00;                        {Interrupt 21, function 2D}
      CX := HM;                                   {Load correct time}
      DX := SCS;
      MsDos(Regs);                                   {Call interrupt}
   End;

   {Calculate correct date and reset system date accordingly}

   If Days > NoDays[YType,13] then
   Begin
      Year := Year + 1;
      Days := Days - NoDays[YType,13];
      if (Year mod 4) = 0 then YType := 2
      else YType := 1;
   End;
   x := 0;
   While Days > NoDays[YType,x+1] do x := x + 1;
   with Regs do
   Begin
      MD := x*256 + Days - NoDays[YType,x];
      AX := $2B00;                        {Interrupt 21, function 2B}
      CX := Year;
      DX := MD;
      MsDos(Regs);                                   {Call interrupt}
   End;
End;


PROCEDURE TIMER_OFF;

{Resets the interrupt vector for 1CH to its old value, resets
 interrupt frequency and restores correct time of day}

Begin
   With Regs do
   Begin
      AX := $251C;               {Interrupt 21, Function 25, on 1CH}
      DS := Seg1C;
      DX := Ofs1C;
      MsDos(Regs);                   {Set DS:DX as interrupt vector}
   End;
   Change_Timing_Frequency(0,0);
   Reset_Time;
End;
```

The only functions that still have to be carried out by Inline are the disabling and enabling of maskable interrupts.

Because most BIOS and DOS interrupts are not reentrant, the documentation for Versions 3.0 and 4.0 recommends against I/O, DOS calls, or dynamic memory allocation from within an interrupt handler. However, in testing, I found that some of these are available, particularly when the Version 4.0 code is used. Screen I/O is allowable because interrupt 10H is re-entrant. Writing directly to video memory should not be attempted under Versions 2.0 and 3.0, because the system sometimes hangs. Dynamic memory access appears to be trouble-free under Version 4.0, although caution should be exercised. User-written procedures can be called from within Interrupt_Handler under Version 4.0 but not under Versions 2.0 and 3.0. Excluding cases with interrupts that are not reentrant, Turbo Pascal procedures and functions can be called under all versions. Floating-point arithmetic cannot be used under Versions 2.0 and 3.0 if an 8087 or 80287 coprocessor is present, as the latter's registers are not saved. Under Version 4.0, floating-point arithmetic can be used. Declaration of variables local to the interrupt handling procedure will cause the system to hang under Versions 2.0 and 3.0, but not under Version 4.0. The reasons, and possible solutions, are discussed below. Finally, an overriding restraint on anything carried out within Interrupt_Handler is execution time. The execution time for Interrupt_Handler must be less than the period between time-of-day interrupts. For example, a single Writeln in the body of Interrupt_Handler will cause the system to hang if the time-of-day interrupt occurs every millisecond.

Timer_On begins by poking the base address of Turbo Pascal's data segment into the specially prepared location in Interrupt_Handler. As previously detailed, this is not necessary under Version 4.0. Indeed, it must not be done, as it will cause the program to crash. Timer_On then records the address of the interrupt handler that normally services interrupt 1CH and changes it to the starting address of the procedure Interrupt_Handler. The new address is 7 bytes greater than the actual offset of Interrupt_Handler, because the compiler inserts a standard 7-byte entry code at the beginning of a procedure. The entry instructions are superfluous when the procedure is called as an interrupt, rather than by the main Pascal program, and must be skipped. When the Interrupt instruction is used under Version 4.0, no superfluous entry code will be inserted; hence, the new address will simply be the offset of Interrupt_Handler. If variables local to Interrupt_Handler are declared, the size of the entry code will be increased. The Version 4.0 compiler handles such variations automatically, but under Versions 2.0 and 3.0 the appropriate adjustments to the new address must be made. For instance, if 3 to 32 bytes of local variables are declared, an extra 3 bytes must be added to the new address. When timing is no longer required, the normal jump address for interrupt 1CH is restored using the procedure Timer_Off.

The procedure Change_Timing_Frequency alters the value at which the 16-bit counter for channel 0 initiates an interrupt. The value is entered as a parameter to the procedure, broken into high- and low-byte values. To achieve approximately millisecond resolution, the constants HiFast and LoFast are used as parameters. They result in the occurrence of an interrupt when the counter reaches 4A9H, a clock frequency of approximately 1000.15 Hz. When timing is no longer required, the time-of-day interrupt is returned to its normal frequency by calling Change_Timing_Frequency with parameters (0,0). The appropriate calls to Change_Timing_Frequency are made automatically by Timer_On and Timer_Off.

The parameter 4A9H is derived from hardware specifications. It should give the best approximation to one interrupt per millisecond. The best parameter value was also determined empirically by comparison with an external clock. The estimate was done using an AT clone with a 6-MHz system clock and a PC clone, both at 4.77-MHz and 8-MHz clock settings. For the AT, LoFast = A9H gave the best results; for both settings of the PC, LoFast = AAH was optimal. In both cases, a gain of 0.015% occurred. Bührer et al. (1987) used 4A9H (LoFast = A9H) for a system using a 4.77-MHz clock. They also noted a gain of approximately 0.015%. These conflicting results indicate that calibration against an external clock may be prudent.

The procedure Get_Time_and_Date is called by Timer_On to record the system time before interrupt frequency is increased. Get_Time_and_Date is again called, from the procedure Reset_Time_and_Date, to record the accelerated system time and date after interrupt frequency is returned to normal. Reset_Time_and_Date determines the actual time that has elapsed between the time at which the timer was turned on and the time at which it was turned off by dividing the difference between the two date and time readings by a factor of approximately 55 (that is, the increase in interrupt frequency). The correct time and date is then calculated, and the system clock is reset accordingly. Although this scheme for the restoration of the system's date and time is conceptually simple, the actual code is quite complex, due to the need to take into account the irregular nature of units of time and date and the occurrence of leap years. The algorithm is Reset_Time_and_Date will be reliable as long as the increase in the date does not increment the year counter twice. Reset_Time_and_Date is automatically called by Timer_Off.

The final section of this paper illustrates how the interrupt-driven software timer described above can be integrated into a high-level language program controlling an experiment (Listing 3). Timer_On is called to redirect interrupt 1CH, to record the current time, and to increase the frequency of the time-of-day interrupt. Interrupt_Handler is not called explicitly by the program but must be compiled with it so that it can be accessed by interrupt 1CH. Timing is accomplished by manipulating the integer variable *t*. Once the timer has been set up, *t* will increment every millisecond. When timing is no longer required, the frequency of interrupts is reduced and interrupt 1CH is redirected to its usual interrupt handler by calling Timer_Off. This returns the operating system's time-of-day function to normal, and sets it to its correct value.

Controlling Display Onset

A video display is produced by an electron-beam-exciting screen phosphor. The phosphor must be periodically reexcited or refreshed to maintain the display as phosphor luminance level decays with time. Typically, a video screen is refreshed at a frequency of 60 Hz. Hence, a new screen is written every 16.7 msec. To synchronize timing with the appearance of a stimulus, it is necessary to know the current point in the refresh cycle. A reference point is usually gained by monitoring the vertical synchronization or vertical-retrace signal. The vertical-retrace signal occurs during each screen refresh. If timing is started when the retrace signal is detected, random variance in the measurement of stimulus onset will be removed. When a stimulus is sent to the screen, information about it is first stored in video memory. The latter is scanned sequentially by the video adapter during each refresh. If the user wishes to time stimulus onset (as described above), the stimulus information must be stored in video memory before the critical refresh, but no part of it may be written to the screen during the previous refresh. One method of ensuring this is to disable the video signal. Stimulus information can then be sent to video memory without the stimulus's appearing on the screen. Subsequently, the video signal is enabled and timing is begun when the next vertical retrace signal is detected.

Solutions to the problem of controlling visual stimulus onset have been attempted at a number of levels. Reed (1979) described hardware modifications to the Apple II microcomputer to make the vertical synchronization signal available to timing software and to allow the video signal to be disabled. Bührer et al. (1987) detailed how to perform the same functions for the IBM PC/XT/AT family with assembly language routines. Listing 2 shows how these functions can be performed by Turbo Pascal programs. The code is identical for Versions 2.0 and 3.0 of Turbo Pascal, but some changes are required for Version 4.0. The changes required are included as comments in the listing.

Three generic types of video adapters are commonly used: the monochrome and color adapters (M/CA), the Hercules monochrome adapter (HMA), and the extended graphics adapter (EGA). A number of differences exist between adapter types. The vertical retrace signal is available in a register of the video adapter. The register can be read through a CPU port by software. Both the M/CA and EGA signal a vertical retrace by setting bit 3 of the relevant register, whereas the HMA zeros bits 4 and 6. The address of the register containing vertical retrace information varies with the hardware being used. The appropriate address can be obtained from BIOS for all adapter types. The video signal can be disabled and enabled by writing the appropriate values to another video adapter register, again through a CPU port. The address and values can be obtained, for the M/CA and HMA, from BIOS. The video signal for the EGA can be disabled by turning off all planes of the color plane enable register. The color plane enable register is controlled through index 12H of CPU port 3C0H. The index is selected by first enabling indexing, then reading the EGA's feature control register, and then writing the index to port 3C0H. The address of the feature control register will be different for monochrome and color modes. It can be determined on-line by checking the EGA's current mode and selecting the address accordingly.

The procedure Video_Sync_On should be called at the start of an experiment, before trials have begun. It will query the user as to the type of video adapter being used and then determine all the appropriate addresses and values. The Display procedure can then be used for synchronized presentation of stimuli during trials. The Display procedure requires two parameters: the stimulus (a string of characters) and the *x* and *y* screen coordinates at which it will be presented. Although most video con-

LISTING 2
ScrSync.Inc.

```
{Uses          VERSION 4: Functions used in this file are included
   Crt;        in separate   units which must be linked with the
   Dos;        statements on the left.}

TYPE
   StimType = String[10];                          {Stimulus type definition}

VAR
   Display_Port,      {Write only port controlling video enable/disable}
   VSYNC_Port,               {Read only port, vertical refresh information}
   EGA_FCR         : integer;                  {EGA Feature Control Register}
   Enable,Disable  : byte;                  {Values to write to Display_Port}
   Adapter_Type    : 1..3;               {1=MCA, 2=HMA, 3=EGA : set by user}


PROCEDURE VIDEO_SYNC_ON;

{Sets up variables according to type of CRT controller card
 present, as indicated by the parameter Adapter_Name}

CONST
   BIOS_DSeg    = $40;                                     {BIOS data segment}
```

LISTING 2 (Continued)

```
   Addr_6845    = $63;                  {Base address of 6845 registers}
   Crt_Modset   = $65;                             {Video mode register}

TYPE
   Registers = record                                  {CPU registers}
      case integer of
         0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer);
         1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
      end;
{VERSION 4: This Type is already declared in the DOS unit}


VAR
   Regs            : registers;                {Used for interrupts}
   Port_6845       : integer absolute BIOS_DSeg:Addr_6845;
   Current_Mode    : byte absolute BIOS_DSeg:Crt_Modset;
   {These variables correspond to memory locations in the BIOS
    which contain the address of the base port of the  6845 and
    the value of the current video mode respectively}

BEGIN
   Write('Video Adapter Type (1 = MCA, 2 = HMA, 3 = EGA) : ');
   readln(Adapter_Type);
   Display_Port := Port_6845 + 4;                         {Offset 4}
   VSYNC_Port   := Port_6845 + 6;                         {Offset 6}
   Enable       := Current_Mode;         {Current mode = video enabled}
   Disable      := Current_Mode and 247;   {Bit 3 = 0 disables video}
   If Adapter_Type = 3 then with Regs do                      {EGA}
   Begin
      AH := $F;                          {Interrupt $10, Function $0F}
      intr($10,Regs);                         {Determine current mode}
      if AL in [7,$F] then EGA_FCR := $3BA      {Monochrome Monitor}
         Else EGA_FCR := $3DA;                      {Colour Monitor}
      Display_Port := $3C0;
      Enable := $20;
      Disable := $F;
   End;
   clrscr;
END;


PROCEDURE DISPLAY(stimulus: StimType; x,y: integer);

Var
   Dummy : Byte;                       {Used to reset EGA FCR Flip-flop}

BEGIN
   if Adapter_Type = 3 then                                    {EGA}
   Begin
      Dummy := Port[EGA_FCR];                       {Enable Indexing}
      Port[Display_Port] := $12;                   {Select Index 12H}
   End;
   port[Display_Port] := Disable;                   {Disable refresh}
   gotoxy(x,y);                                {Beginning of stimulus}
   write(stimulus);                               {Store in video RAM}
   if Adapter_Type = 2 then                  {HMA: Retrace when bits}
      repeat until (not(port[VSYNC_Port]) and 80) = 80   {6 and 4 down}
      Else repeat until (port[VSYNC_Port] and 8) = 8;    {EGA and M/CA}
                                               {Retrace when bit 3 up}
   port[Display_Port] := Enable;                     {Enable refresh}
END;
```

trollers are fairly standard, some troublesome variations may be found between brands. For instance, the color graphics adapter supplied by Zenith Data Systems (Z-100 PC Series, 1985) requires write-only port 3DAH to be set to 0 before any of the functions described above will work.

A major advantage of the programs described here over those described by Bührer et al. (1987) is that they do not require the character and attribute values of the elements of the display to be calculated, stored, and loaded into video memory. Instead, the more familiar screen-control functions supplied by the high-level language can be used.

Synchronization with the vertical retrace removes most random error from stimulus onset. However, there will be a constant delay between the enabling of the screen and the appearance of the stimulus. This will vary depending on where on the screen the stimulus is written. The delay can be calculated using the following formula:

$$\text{Delay} = y \text{ position} \times (\text{refresh time}/25).$$

The y position can take on values from 0 to 24 from the top to the bottom of the screen. Not all monitors refresh at the rate indicated by their specifications. Refresh rate depends on the adapter. During testing with various adapters, I found that a number of 60-Hz monitors were driven at 50 Hz. The rate can be determined empirically for an individual system by using the timing program to determine the delay between two vertical retraces. Determining exact stimulus duration is not so easy. Stimulus duration is determined jointly by the refresh time and the decay time of the monitor phosphor. The latter constant can be difficult to determine for the (long-decay) phosphors used in standard monitors, because the slope of the decay function may be shallow in the critical range. One solution is to present displays in inverse video. Because removing the display involves brightening the screen phosphor, a relatively rapid operation, this source of error is minimized.

A Simple Reaction Time Experiment

Listing 3 illustrates how the code described above can be incorporated into a program running an experiment. The experiment begins with a request for adapter type. On each trial the user is asked to enter a target string. The screen is cleared, and a fixation cross is presented for 1 sec. The cross is followed by a synchronized presentation of the stimulus. The user makes a two-choice response (for instance, a lexical decision) using the key f or g. Reaction time and response are displayed, and the user can then request further trials.

The timing and screen control procedures are made available to the program RT_Experiment, running under Version 2.0 or 3.0, using Turbo Pascal's include-file feature. The code presented in Listings 1 and 2 must be present in the files Timer.Inc and ScrSync.Inc, respectively. They are included in the compilation of RT_Experiment by the statements {$I Timer.Inc} and {$I ScrSync.Inc}. If Version 4.0 is being used, the include-file feature is not supported. Instead, the code in Listings 1 and 2 must be compiled into a unit and linked to RT_Experiment with the Uses statement.

The main body of RT_Experiment begins by setting up the software timer and screen synchronization. This is achieved by calling the procedures Video_Sync_On and Timer_On. After a stimulus string is received from the user, the cursor is turned off and a fixation cross is written to the screen. A delay of 1 sec is achieved by zeroing the timer variable *t* and then checking its value until it reaches 1,000. Synchronized stimulus presentation is performed by Display, and *t* is zeroed immediately after. The reaction time to the stimulus is recorded by reading the value of *t* as soon as a valid response is detected. After the experiment is finished, the timer must be disabled using Timer_Off. If this is not done, the system will crash. As can be seen from the example, screen control and timing functions are easy to include and to use in a high-level language. They may be conveniently stored in a program library and used routinely to extend the power of the high-level language.

LISTING 3
Sample Experimental Program

```
PROGRAM RT_EXPERIMENT;

{Uses    VERSION 4: Functions used in this file are included in a
   Dos; unit which must be linked with the statements on the left.}

{$I Timer.Inc}                                      {Timing routines}
{$I ScrSync.Inc}                    {Display routine for adapter in use}

VAR
   Stimulus: StimType;    {Stimulus from user, type defined in MCA.Inc}
   Another,
   Response: char;                                     {Flow control}
   RT       : integer;                  {Reaction time in milliseconds}


PROCEDURE CURSOR(Present: boolean);

{Procedure to turn the cursor on and off}

TYPE
   Registers = record                                  {CPU registers}
      case integer of
         0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer);
         1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
      end;
{VERSION 4: This Type is already declared in the DOS unit}

VAR
   Regs : registers;                               {Used for interrupts}

BEGIN
   With Regs do
   Begin
      if mem[$0000:$0449] = 7 then
      Begin
         if Present then CX := $0B0C
         else CX := $3000;
      End
      Else Begin
         if Present then CX := $0707
         else CX := $2000;
      End;
      AX := $0100;
      intr($10, Regs);
   End;
END;

BEGIN
   Timer_On;                                             {Set Timing}
   Video_Sync_On;                  {Initialize screen synchronization}
   Repeat
      write('Enter Stimulus Word (=<10 characters): ');
      readln(Stimulus);
      Cursor(false);                                     {Cursor off}
```

LISTING 3 (Continued)

```
         clrscr;
         gotoxy(40,12);
         write('x');                                       {Fixation point}
         t := 0;
         repeat until t = 1000;                         {Delay for 1 second}
         clrscr;
         Display(Stimulus,35,12);                     {Synchronized stimulus}
         t := 0;                                              {Zero timer}
         repeat
            read(kbd,Response);                          {Wait for response}
         until Response in ['f','g'];
         rt := t;                                    {Record reaction time}
         clrscr;
         Cursor(true);                                         {Cursor on}
         writeln('Response = ',Response,'  Reaction Time = ',RT:4,' ms');
         write('Another Trial ? ');
         readln(Another);
      until Another <> 'y';
      Timer_Off;                                            {Reset timing}
   END.
```

REFERENCES

Borland International, Incorporated. (1984). *Turbo Pascal Version 2.0 Reference Manual.* Scotts Valley, CA: Author.

Borland International, Incorporated. (1985). *Turbo Pascal Version 3.0 Reference Manual.* Scotts Valley, CA: Author.

Borland International, Incorporated. (1987). *Turbo Pascal Version 4.0 Reference Manual.* Scotts Valley: Author.

Bührer, M., Sparrer, B., & Weitkunat, R. (1987). Interval timing routines for the IBM PC/XT/AT microcomputer family. *Behavior Research Methods, Instruments, & Computers*, **19**, 327-334.

Reed, A. V. (1979). Microcomputer display timing: Problems and solutions. *Behavior Research Methods & Instrumentation*, **11**, 572-576.

(Manuscript received November 3, 1987;
revision accepted for publication February 10, 1988.)