High-Accuracy Timing

by Brian Frost

There is often a requirement for PC timing information or delays in the range of 10 microseconds to 100 milliseconds, and although one would think that this should be simple, obtaining such timing information is actually extremely difficult.

While short delays can be obtained using processor instructions or count loops and long delays can be created from the 18Hz clock tick or the real-time clock, there is no readily available method for obtaining accurate 'intermediate' delays that work for all targets. Such delays are useful for generating hardware pulses by combining a delay with toggling an I/O line, or for creating tests for a response time-out by using a maximum number of repeated calls to a test function, with a rate controlled by a fundamental short delay. Some programmers will have used the `Delay(Milliseconds)` procedure in the Borland Pascal 7 (BP7) `CRT` unit in this way and are often surprised to find that no such call exists in Delphi.

On other platforms it can be quite easy to fabricate your own delay. A common method that goes back to the earliest days of the microprocessor is to use nested count loops that give a known delay based on the counts and the clock speed of the processor. Such methods are still regularly used in many single-chip microprocessor applications where a fixed crystal clock exists. Unfortunately on the PC platform this is not a good solution due to the wide variety of processor speeds, other interleaved system activity and the requirement that CPU time should not be 'wasted' in a multi-tasking environment. In spite of this I maintain that there is still a need to be able to implement short, accurate delays, as long as they are used with care.

This article sets out to explain the various methods of obtaining timing and delays on the PC under DOS and Windows and their various limitations and benefits. In particular it concentrates on obtaining these 'intermediate' times in the range 10 microseconds to 100 milliseconds. It provides a solution to those who miss a `Delay` function in Delphi and derives a general purpose unit that works all the way from BP7 right through to 32-bit Delphi and offers accurate time measurement and delays with a resolution of better than 1 microsecond. The article then develops this short time measurement to create a useful generic low-level execution profiling tool for testing program speed and performance.

PC Timing Options

The PC is a difficult platform upon which to obtain high resolution timing since its basic hardware timer capability was really only designed to provide a series of interrupts that would keep the dynamic RAM refreshed and to generate a fixed tick at about 18Hz to maintain a software time-of-day clock. A third timer channel is actually available, but only in situations where the operating system permits it. There are a number of ways in which a programmer can obtain timing information, but each has various limitations and benefits as described in the following sections.

DOS Real And Protected Mode

Under MSDOS some standard absolute time functions are available in the BIOS services, but these are limited in resolution to one clock tick of approximately 55ms (18Hz). Since these functions are linked to the requirements of system and file times and dates, they are ideal for longer delays but of little use for short delays. It is possible to use the spare timer channel on the PC timer from your program to obtain flexible and high-resolution absolute or relative timing, but you must program it directly and restore it on exit, since other programs (like games) will often be using this same technique. A good example of this is in OPTIMER.PAS (see the note at the end of the article).

To avoid tying up timer hardware, many programs implement timing delays by using processor count loops that convert the required number of microseconds or milliseconds into a count via a precalibrated delay factor `CountsPerMillisecond`. An example of this is the Borland `Delay(Millisec)` procedure in the BP7 `CRT` unit. This solution works well unless the final DOS program is run as a task under Windows, where the multitasking processes can affect the initial calibration at program start or the actual delays themselves. Nowadays, of course, this is usually the case.

Windows

There is similarity between the DOS time-of-day functions and those available in the Windows API, but again these are limited in resolution depending on the version of Windows being used. Windows 3.X is limited to one clock tick of 55ms but Windows 95 improves this to 1ms. A popular method for obtaining timing under Windows is to use `GetTickCount` which returns the number of milliseconds that has elapsed since Windows was started, but the limited resolution prevents short delays from being created.

This same Windows tick resolution applies to any Windows timer that can be assigned to your program (eg the Delphi `TTimer`), but trying to implement delays with these timers is even less accurate since they are message based and so only really useful for events occurring at several 100s of milliseconds. An added limitation is that unrestrained use of these timers is discouraged as they are a limited resource under 16-bit Windows.

I work on DOS real and protected mode programs that now have to run under Windows and for some time I used the BP7 `Delay` procedure for my short delays. However, I noticed that although the program worked fine out of Windows, when run under Windows sometimes the actual delay was not consistent. The `Delay` procedure uses a software calibration loop to determine the scale factor (counts per millisecond) that is subsequently used to convert your required delay into a processor wait loop. This technique works fine to within about 2% under DOS only. When the same code is run as a DOS task under Windows, it seems that the calibration loop can be interrupted by Windows and gets the scale factor wrong about 1 time in 20 or 30. Since the calibration loop is only executed once at the start of the program, such an error can be quite serious and attempts to check the calibration against absolute timing, such as the clock tick, do help but are slow and still not foolproof. Windows 95 actually seems to make matters worse since it is much more intensive in stealing time away from DOS tasks, so creating longer holes in the CPU time that is made available to such calibration loops.

Under Windows the spare timer channel is a valuable operating system resource that must not (*cannot* in fact) be directly accessed, since Windows uses it to control its own internal timing.

General Purpose High-Res Timing

I write and support code for various Pascal based targets, ranging from self-contained MSDOS applications written using BP7 with their own menu system, or based on Turbo Vision, through to more recent 16-bit or 32-bit Windows applications written in Delphi. My task is often made more complex because some of my Pascal code has to be shared between all targets, including requirements for (say) a 1ms delay. Until recently I had got around these requirements by a mixture of `GetTickCount` type timing for delays of a few 100 ms or more, processor count loops for short non-accurate delays and direct hardware access to my own system-specific hardware timers for the really accurate short delays. My real Holy Grail, though, has always been a general purpose high resolution timing unit that could be used under all of these compilers to provide a set of simple delay procedures.

I decided to see if it was possible get at the PC hardware timer under all operating system modes and my starting point was TurboPower's OPTIMER.PAS (see note at end). This unit allows direct access to the spare timer channel under MSDOS and provides a high-resolution timer value that increments approximately every 840ns, so is ideal for obtaining short millisecond or microsecond delay values. However, I quickly discovered that running the same code as a DOS task under Windows gave very erratic results.

To find out why Windows upsets access to the PC timer I turned to the Microsoft Developer Network CDROM, where there is a good discussion of Windows timer services. This makes clear the reasons why such direct hardware access is not possible. Windows itself is using this timer at the highest privilege level (ring 0) and has 'virtualised' it causing accesses to its physical addresses to be trapped and to return unpredictable values, however the discussion goes on to show a legitimate method for accessing timer functions under Windows 3.1 or later. The timer is handled by a Windows device driver called the VTD (Virtual Timer Device) and it turns out that access to the VTD is possible by using interrupt 2F, the multiplex interrupt, which is designed to provide a connection between DOS applications or TSRs and any Windows host.

To recap, this most tricky timing situation occurs only when you need to get accurate timing information inside a BP7-compiled DOS real or protected mode program when it is being run under 16-bit or 32-bit Windows. If Windows is not running, you can get at and 'own' the timer directly. So to get timer information under all circumstances we have to do the following.

From within the DOS program, determine whether Windows is running. A function is provided using interrupt 2F for this. If Windows is not running we can use the timer directly, otherwise we must only access the timer through the Windows VTD (again using interrupt 2F).

If we are compiling a 16-bit Delphi target this technique still works, but of course Windows will always show up as present and all timer access is via the VTD. 32-bit Delphi targets are actually a lot simpler in their access to the timer. Before we look at the detail of the code for each target, we need to discuss where we are headed in terms of presenting timer functionality, since it is 32-bit Delphi that gives us some directions with this.

Looking at the Win32 API I was very pleased to find that access to the high-resolution timer is provided directly by two routines in the `Windows` unit:

```
function
  QueryPerformanceCounter(
    var lpPerformanceCount:
    TLargeInteger): BOOL;
    stdcall;
function
  QueryPerformanceFrequency(
    var lpFrequency:
    TLargeInteger): BOOL;
    stdcall;
```

These are not at all well documented but give us all of the access that we need to implement accurate time measurement or delays. They are really intended for the implementation of program profiling tools, where nanosecond resolution is required, but are standard within the API, allowing us to make good use of them (even for program profiling too, as we will see later!).

Basically, you first call `QueryPerformanceFrequency` once to get a timer resolution value of how many times per second the timer counts. You can later use this value with repeated calls to `QueryPerformanceCounter` which returns an actual

timer value that increments at the QueryPerformanceFrequency rate. TLargeInteger is a 64-bit value that ensures that it is a long time before the count rolls over (some 1.5E13 seconds, or about 490,000 years in fact, enough for several versions of Delphi and Windows!). Interestingly, although Microsoft have given us a way of obtaining the counts per second value of the timer, all PC systems seem to return 1,193,180 or 838.09ns, which is the count rate of the spare timer channel. Presumably this may be a different value on other platforms or perhaps even under future versions of Windows.

It was the existence of these routines that provided the prototype for the functions that I decided to implement in my general purpose high-resolution timer unit, so I set about implementing these same named functions to work under 16-bit targets for Windows 3.1 and MSDOS. The unit HIRESTMR.PAS implements various fragments of code depending on the target with which it is compiled (there are various conditional directives to control this) and within the unit accesses to the timer are as follows:

- ***32-Bit Compiler:*** This is easy: any requirement for the named functions QueryPerformanceFrequency and QueryPerformanceCounter causes a direct link with these functions in the Win32 API (defined in Delphi's Windows unit).
- ***16-Bit Compiler (Delphi 1 or BP7):*** For 16-bits, HIRESTMR.PAS has some initialisation code that attempts to call Windows to determine if Windows is running and which version it is, and then either calls the timer directly (if DOS is the only operating system) or calls the timer via the Windows VTD driver. In both cases, the timer value is returned in a 64-bit form compatible with the QueryPerformanceCounter Win32 API function.

The result of this is a general purpose unit that gives us the same functionality with 16-bit code as we get with the Win32 API timer functions. This basic timer capability can then be used to create short delays and to do other work as we will see.

Interface To HIRESTMR.PAS

The interface section of this unit is common to all targets and provides some useful types and constants as well as the actual timer and delay procedures. See Listing 1.

At the start are some constants of type TLargeInteger. This type is defined in 32-bit Delphi but for 16-bit targets we have defined it. These are 64-bit integers that contain copies of useful values that describe the timer: its maximum value (before it starts again at zero), the number of counts per second, and the count of the overhead in calling it. This last value may be important because on slower systems running Windows the time taken to get the timer value from the Windows VTD driver may be of significance. I have measured times between 2us and 5us on a Pentium 133 up to around 50us on a 386SX-40.

Following on after these constants are some Delay... routines offering microseconds, milliseconds and seconds. Note that the parameter type is very flexible with these procedures: DelayUS is passed a LongInt microseconds value allowing a wide range, also DelayS is passed a floating point type allowing fractional seconds. It must be stressed that, technically, these Delay... procedures violate the concept of a multi-tasking operating system such as Windows and so should be used with care, however they are *very* useful where short, accurate delays in the microseconds and low millisecond range are required, or where it is guaranteed that your application is the only application running. Remember to check the effect on other system I/O performance such as networking, printing etc!

The final *Windows support routines* are exported in case they can be of use elsewhere, but otherwise they are only used internally.

Delay Procedures Detail

Each delay procedure is based on a call to:

```
procedure WaitForElapsed(
  const AValue : TLargeInteger);
```

which repeatedly calls QueryPerformanceCounter until a value of AValue or greater is obtained. AValue is calculated at the start of the delay from the current time plus the required delay. For example, the DelayUS procedure is constructed as shown in Listing 2 (over the page).

Note the use of the constant r_CounterCallOverhead. This contains the number of timer counts that will be lost during each call to get the timer value and has been determined once at the start of the program. The minimum achievable delay will be twice this value and so this count is removed from the required delay.

➤ *Listing 1*

```
{ ** Timer definitions }
{ Definitions relating to the performance counter in use }
const
  { The max value before rollover. Default is -1 but
    MSDOS timer counter rolls over each midnight and will
    reprogram this value }
  r_CounterMaxValue : TLargeInteger = (QuadPart : -1);

  { The resolution which defaults in DOS and Win3 to about 0.8us,
    Delphi 2 32-bit will load this value too, in case
    it changes in the future }
  r_CountsPerSec    : TLargeInteger = (QuadPart : 1193180);
  { The number of counter counts taken to call the performance
    counter }
  r_CounterCallOverhead : TLargeInteger = (QuadPart : 0);
procedure DelayUS(AValue : longint);
  { Delays this value using the performance counter }
procedure DelayMS(AValue : longint);
  { Delays this value using the performance counter }
procedure DelayS(const AValue : extended);
  { Delays this value using the performance counter }
{ ** Windows support routines }
const
  b_WindowsInstalled : boolean = False;
  w_WindowsVersion   : word    = 0;
```

Note that `WaitForElapsed` does *not* call `Yield` or any other Windows or Delphi message processing because the whole point of these delay procedures is to implement an *accurate* delay (under DOS too). If you do need a 'well behaved' Delphi delay of (say) 5s, you could use code like that in Listing 3. This is not as neat as using a `TTimer` but can be useful for in-line delays where accuracy and neatness are not a problem. Vary the 50ms delay to vary the effect of the 'holes' this will insert into Windows activity.

Timing In 16-Bit

As explained earlier, 16-bit mode requires that we either call the timer directly or via Windows depending on whether we find Windows running or not. If we have only MSDOS running, the timer channel is completely ours and we must execute our own code to read it in a form that gives us a 64-bit timer value. The actual timer only returns a 16-bit count value, so we must do some additional work to obtain a 64-bit value as shown in Listing 4.

In MSDOS mode where there is no Windows running, this code does three main tasks. Firstly, at the start of the program it puts the timer into the required count mode in case other applications have left it in some arbitrary state. During the program the procedure `QueryPerformanceCounter` does the work of returning a 64-bit timer value' and finally at the end of the program, the timer default state is restored.

The code for reading the timer is based on the OPTIMER.PAS unit but extends it to combine the 16-bit timer value (incrementing every 840ns) with another 32-bit value held in the BIOS data area that is incremented on each overflow of the timer. This produces a 48-bit timer value which is returned as a 64-bit value with the high-order 16-bits set to zero. The effect of this is

➤ *Listing 3*

```
procedure GoodDelphiDelay;
var I : integer;
begin
  for I := 1 to 100 do begin
    DelayMS(50);
    Application.ProcessMessages;
  end;
end;
```

➤ *Listing 2*

```
procedure DelayUS(AValue : longint);
{ Delays this value using the performance counter }
var r_Elapsed  : TLargeInteger;
begin
  r_Elapsed.QuadPart := ((AValue * r_CountsPerSec.QuadPart) * 1e-6) -
                        (r_CounterCallOverhead.LowPart * 2);
  If r_Elapsed.QuadPart < 0 then r_Elapsed.QuadPart := 0;
  WaitForElapsed(r_Elapsed);
end;
```

➤ *Listing 4*

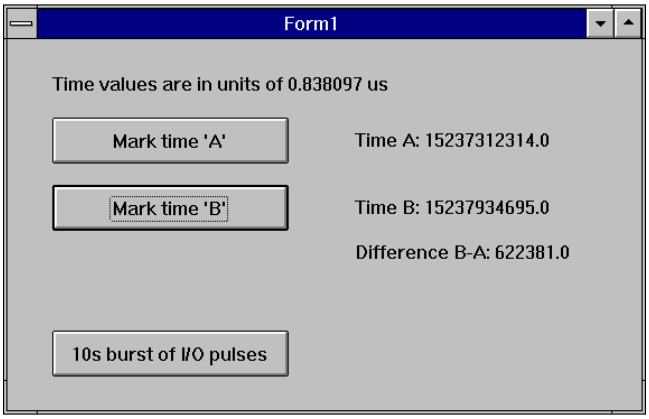
```
procedure InitializeTimerCounter;
{ Setup the timer chip to required mode
  Thanks to TurboPower inc for information in OPTIMER.PAS }
begin
  { mode 2, read/write channel 0}
  Port[$43] := $34;        {00110100b}
  asm
    jmp @1  {delay}
   @1:
  end;
  Port[$40] := $00;        {LSB = 0}
  asm
    jmp @2  {delay}
   @2:
  end;
  Port[$40] := $00;        {MSB = 0}
end;
procedure RestoreTimerCounter;
{ Restore the timer chip to its normal state
  Thanks to TurboPower inc for information in OPTIMER.PAS }
begin                      {RestoreTimer}
  {select timer mode 3, read/write channel 0}
  Port[$43] := $36;        {00110110b}
  asm
    jmp @1 {delay}
   @1:
  end;
  Port[$40] := $00;        {LSB = 0}
  asm
    jmp @2 {delay}
   @2:
  end;
  Port[$40] := $00;        {MSB = 0}
end;
procedure QueryMSDOSTimerCounter(Var AValue : TLargeInteger);
{ Returns the value of the hi perf counter.
  Thanks to TurboPower inc for information in OPTIMER.PAS }
begin
  asm
    cli             {Disable interrupts}
    mov  dx,$20     {Address PIC ocw3}
    mov  al,$0A     {Ask to read irr}
    out  dx,al
    mov  al,$00     {Latch timer 0}
    out  $43,al
    in   al,dx      {Read irr}
    mov  di,ax      {Save it in DI}
    in   al,$40     {Counter -> bx}
    mov  bl,al      {LSB in BL}
    in   al,$40
    mov  bh,al      {MSB in BH}
    not  bx         {Need ascending counter}
    in   al,$21     {Read PIC imr}
    mov  si,ax      {Save it in SI}
    mov  al,$0FF    {Mask all interrupts}
    out  $21,al
    mov  ax,$40     { delay }
    mov  ax,$40     {read low word of time}
    mov  es,ax      {from BIOS data area}
    mov  dx,es:[$6C]
    mov  cx,es:[$6E]{result now as CX:DX:BX}
    mov  ax,si      {Restore imr from SI}
    out  $21,al
    sti             {Enable interrupts}
    mov  ax,di      {Retrieve old irr}
    test al,$01     {Counter hit 0?}
    jz   @done      {Jump if not}
    cmp  bx,$FF     {Counter > $FF?}
    ja   @done      {Done if so}
    add  dx,1       {Else count int req.}
    adc  cx,0       {ripple carry}
   @done:
    les di, AValue
    mov es:[di],  bx  {lsw}
    mov es:[di+2],dx
    mov es:[di+4],cx
    mov ax,0
    mov es:[di+6],ax  {msw}
  end;
end;
var
  SaveExitProc : pointer;
{$F+}
procedure OurExitProc;
{ Restore timer to its original state}
begin
  ExitProc := SaveExitProc;
  RestoreTimerCounter;
end;
{$F-}
procedure InitialiseMSDOSTimerCounter;
{ Called at start to setup timer }
begin
  {set up our exit handler}
  SaveExitProc := ExitProc;
  ExitProc := @OurExitProc;
  {reprogram the timer chip}
  InitializeTimerCounter;
  { Set the counter max value }
  With r_CounterMaxValue do begin
    LowPart  := $00B0FFFF;
    HighPart := $18;
  end;
end;
```

that our MSDOS timer value rolls over more quickly outside Windows than when Windows is running, although this is academic since HIRESTMR.PAS includes code to take care of the roll-over anyway.

Incidentally, note that there are some 32-bit 386 instructions in this code so unfortunately the code will not run on 286 or earlier processors without some modification: I did not consider that this was a severe limitation.

➤ *Table 1: Demo programs on this month's disk*

Compiler	EXE file for demo
BP7 16-bit DOS real mode	DEMO_RL.EXE
BP7 16-bit DOS protected mode	DEMO_PR.EXE
Delphi 1 16-bit Windows 3 or 95	DEMO_W16.EXE
Delphi 2 32-bit Windows 95	DEMO_W32.EXE



➤ *Figure 1*

➤ *Listing 5*

```
function GetDeviceEntryPointAddress(ADeviceID : word) : pointer;
{ Returns the entry point for this specified device driver }
begin
  If not b_WindowsInstalled then
    GetDeviceEntryPointAddress := nil
  else
    asm
      mov     bx, ADeviceID  { Device identifier }
      mov     ax, 1684h      { Get Device Entry Point Address }
      int     2Fh            { multiplex interrupt }
      mov word ptr @Result,    di
      mov word ptr @Result+2,  es
    end;
end;
function GetVTDDeviceEntryPointAddress : pointer;
{ Returns the entry point for the virtual timer device driver }
begin
  GetVTDDeviceEntryPointAddress := GetDeviceEntryPointAddress(5 {VTD identifier});
end;
const VTDAddress : pointer = nil;
procedure QueryWindowsVTDCounter(var AValue : TLargeInteger);
{ Returns the value of the hi perf counter }
begin
  If VTDAddress = nil then
    VTDAddress := GetVTDDeviceEntryPointAddress;
  If VTDAddress = nil then
    RunError; {No VTD installed - needs windows}
  asm
    mov ax,$100
    call VTDAddress
    db $66, $50    {push eax}
    db $58         {pop  ax}
    db $5B         {pop  bx}
    db $66, $52    {push edx}
    db $59         {pop  cx}
    db $5A         {pop  dx}
    les di, AValue
    mov es:[di+0],   ax {w0 lsw}
    mov es:[di+2],   bx {w1 }
    mov es:[di+4],   cx {w2 }
    mov es:[di+6],   dx {w3 msw}
  end;
end;
```

If we detect that we are running with Windows installed, the procedures shown in Listing 5 are used. `QueryWindowsVTDCounter` is the equivalent of `QueryPerformanceCounter` implemented for 16-bit Windows. This derives a `VTDAddress` once, which is the entry point of the Windows VTD timer device driver obtained by using the 2F interrupt. On all subsequent calls to the timer, this address is used together with `AX=100h` to get a 64-bit count value passed out as `AValue`.

Miscellaneous Procedures

There are some other less important routines I've exported in HIRESTMR.PAS. `BeginCriticalSection` and `EndCriticalSection` are useful calls made available by the 2F interrupt that allow 16-bit Windows to be temporarily suspended for a short period while some time-critical activity is taking place. It would appear that these are the forerunners of the thread functions in Win32. There should be an equal number of calls to `Begin` and `End`and during the intermediate section Windows messaging is suspended, although interrupt activity continues. I have used these procedures to get a more controlled measurement of the time overhead in calling `QueryPerformanceCounter`.

Demonstration

You are likely to find HIRESTMR.PAS useful for all of the Borland Pascal and Delphi products, so a simple demonstration program has been created using each compiler and included on the disk, from a single source file DEMO.PAS. You may find a Delphi project called DEMO.PAS confusing at first, but the program does not use any VCL components and can be compiled happily as a *.PAS file instead of a *.DPR file. Table 1 shows the file names. A VCL-based Delphi demo is also provided as DELDEMO.DPR. This has two push buttons to read timer values and to show the difference (Figure 1).

Limitations

There are some limitations with the HIRESTMR unit of which you should be aware. The delays are

typically useful down to around 50us. For example, on a Pentium P133, a 100us delay is within about 5%. As the speed of the machine slows or the delay drops below 100us the time overhead to execute the code and to get the timer value becomes significant and up to 50us should be allowed on older slower 386 machines. Note that the demo EXE files report this overhead time, so if you need to assess it, run it on the slowest machine of interest.

In the same way, the delay calculations are made using floating point values so a co-processor is desirable to keep the overhead time down. If a co-processor is not available, this will not prevent the routines from working, it only increases the minimum delay possible but again, 100us delays should still be within 30% to 50%.

Finally, remember that other system activity such as interrupts and dynamic RAM refresh will always be running and that this activity will cause inaccuracies in the delays. Normally this is not a problem, because the use of these delays is to obtain a guaranteed minimum time (for example for a hardware measurement voltage to settle). If you need absolute timing accuracy, call `QueryPerformanceCounter` directly: we will see an example of this in the next section where this timer is used for program timing.

Low Level Program Profiling

Since I am often concerned with the interface between software and hardware in real-time control situations, it is of great interest to understand the timing characteristics of an application within a certain part of the program. Borland's Turbo Profiler is a tool that makes these assessments by monitoring the program's execution path and recording source code line numbers against times, allowing timing to be viewed for each source code line. Unfortunately, I have had difficulty in using this tool with large applications and so it has been less useful than I would have liked. To get around this I have developed a low-level profiling tool that others may find useful and which is linked closely to the high-resolution timing capability in HIRESTMR.PAS. It allows programs to be assessed by observing their actual timing, or by using hardware I/O signals. All code is in the unit PROFIT.PAS.

Profiling Using I/O Signals

The simplest mode of the profiler unit PROFIT.PAS is to show execution of the program at a chosen point by using the parallel printer port as a collection of I/O signals. The main procedures for this are declared at the start of the unit as:

```
procedure PPPRInitialise(
  APortAddress : word);
  { Opens the printer port
    setting defaults }
procedure PPPRSetAllBits;
  { Sets the port to all
    data bits = 1 }
procedure PPPRClearAllBits;
  { Sets the port to all
    data bits = 0 }
```

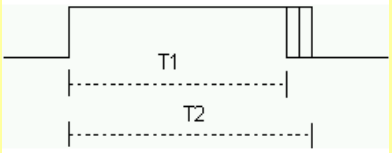
A typical use of these procedures would be as follows. At the start of the program call

```
PPPRInitialise(
  <Printer port address>)
```

to set the printer port data lines to a defined state. Then, around the code to be monitored place a call to `PPPRSetAllBits` and `PPPRClearAllBits`. When the program is run, all printer port data lines will change at these points and this action can be inspected using a storage oscilloscope or timer-counter to get a high degree of accuracy. The necessary pin connections can be seen listed in the unit.

The advantage of this I/O method of profiling is that the code required to change the printer port lines is very short and so does not affect the timing of the actual program. The conditional define `PrinterPortProfiler` is required to activate this code allowing these calls to be left in the program but inactive without the definition.

Take care to check your printer port I/O address. It will almost certainly be one of `$378`, `$278` or `$3BC`, but it must be passed to `PPPRInitialise` or you will get no output.



➤ *Figure 2*

It may seem that observing I/O signals with an oscilloscope is a rather difficult way to debug a program's execution, but in fact you can often extract more information in this way than can be gained from a simple timing dump. For example when a section of code is executed several times, the display will often appear as shown in Figure 2, showing a minimum time T1 and a maximum time T2. The difference between T2 and T1 is a time jitter caused either by a different execution path (eg changed program data) or some other system activity such as interrupts that occurred during T2 but not T1. Being able to view this jitter is a powerful aid in assessing the program activity and provides information that is much less clear when presented as numeric timing values. Another use of I/O profiling is to mark both the occurrence of an external event (for example a button push) and the response of the program to that event. Often interrupts are in use for such tasks and a traditional profiler cannot be used to observe the relationship between an external hardware event and the program response. Using these I/O signals, an oscilloscope can be triggered from the external hardware event and the trace can show the program response. A display as above will be seen which clearly shows both the absolute response time of the program together with how it may vary between repeated events. I've used this technique to diagnose software problems that were causing external events to be 'missed' due to a program's 'inattention' for short periods.

Finally, there are some additional I/O routines for the printer port in PROFIT.PAS that support

```
procedure TMPROpen(const AFileName : string; AMaxItems : integer);
  { Opens the timing profiler }
procedure TMPRStart;
  { Resets the timing profiler }
procedure TMPRMark(ACode : integer);
  { Marks this profile point }
procedure TMPRClose;
  { Closes the timing profiler }
```

➤ *Listing 6*

```
Timing profile dump
Ref:    0, Abs (ms):   0.1198, Diff (ms):   0.0000
Ref:    1, Abs (ms):1000.1768, Diff (ms):1000.0570
Ref:    2, Abs (ms):1000.5330, Diff (ms):   0.3562
Ref:    3, Abs (ms):1999.2809, Diff (ms): 998.7479
Ref:    4, Abs (ms):1999.5499, Diff (ms):   0.2690
Ref:    5, Abs (ms):2999.6405, Diff (ms):1000.0905
Ref:    6, Abs (ms):3000.1106, Diff (ms):   0.4702
Ref:    7, Abs (ms):4639.2573, Diff (ms):1639.1467
```

➤ *Listing 7*

using the printer port as an input. This can be useful for suspending your program until an external event takes place.

The VCL-based Delphi-only demonstration DELDEMO.DPR has a button marked `10s burst of I/O pulses` (Figure 1) which illustrates the working of these I/O signals via your printer port.

Profiling Using Timing Information

Having created the timing unit HIRESTMR.PAS it was an obvious move to extend the profiler to offer timing information at the chosen program points as well as I/O signal activity. PROFIT.PAS has a second very simple section that implements a simple timing profiler, shown in Listing 6.

These procedures are all that are necessary to write timing information from your program at specific points. An example of using this is seen in the HIRESTMR demo program DEMO.PAS which writes actual time values of the example delays to a disk file.

At the start of your program call

```
TMPROpen(<filename>, MaxItems)
```

passing the name of a file that will be created with the timing information and the maximum number of timing items that will be recorded in a heap-based memory array. At a suitable point in the program, simply call `TMPRMark(RefNum)`, passing a code that will define this point. The code simply allows you to see it in the file easily. Use as many of these points as you like until you are likely to approach `MaxItems` when the program is run. Finally, before your program exits, call `TMPRClose` to flush the memory array to the disk file. A typical file will look like Listing 7.

The `Ref` column is the code that you assigned to `TMPRMark`, `Abs` is the absolute time from the start and `Diff` is the incremental time between successive marks. Resolution is to better than 1us. The dump in Listing 7 (from DEMO.PAS) shows the effect of creating several 1 second delays out of repeated calls to shorter delays. The values at references 1, 3 and 5 show a close agreement to 1000ms, but the final timing at reference 7 shows 1.64 seconds where a 1s delay was attempted from 100,000 separate 10us delays, with an obvious code overhead that increased the theoretical 10us.

The use of `QueryPerformanceCounter` is quite obvious within the PROFIT.PAS unit and this profiling technique works on all of the Pascal compilers from BP7 through to Delphi 2 (I've not been able to check it with Delphi 3 yet). With this implementation you are limited to about 6000 recorded items, but otherwise you should find it quite useful. As with the I/O profiler, a conditional definition is required to 'enable' the code, so all calls to these profiling procedures can be left in the program and are inactive without the definition.

Conclusion

I hope that you find these units useful. If you have any comments or suggestions please get in touch.

Brian Frost is the principal of Dorset Design and Developments, working in electronics as well as software development, and can be contacted by email as
bfrost@cix.compulink.co.uk

OPTIMER

Thanks to TurboPower Inc for the information about accessing the PC timer from their freeware OpTimer unit which is included on this month's disk.