ExacTicks 1.1

Precision Timing Tools For Windows

Ryle Design
Purveyors of Big Science Since 1987

License Agreement and Official Fine Print

BY INSTALLING **EXACTICKS** ON YOUR SYSTEM YOU INDICATE YOUR AGREEMENT TO THE FOLLOWING TERMS AND CONDITIONS. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS DO NOT INSTALL OR USE THIS SOFTWARE PRODUCT.

EXACTICKS AND ASSOCIATED DOCUMENTATION ARE DISTRIBUTED AS IS. **RYLE DESIGN** MAKES NO WARRANTY OF ANY KIND, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THIS SOFTWARE AND DOCUMENTATION. IN NO EVENT SHALL **RYLE DESIGN** BE LIABLE FOR ANY DAMAGES, INCLUDING LOST PROFITS, LOST SAVINGS, OR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR THE INABILITY TO USE THIS PROGRAM, EVEN IF **RYLE DESIGN** HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

This software license is for a single developer on a single user system. You may install ExacTicks on more than one computer so long as there is no possibility of it being used at more than one location at any one time, or by more than one developer at any time. If you wish to install ExacTicks on a server in a networked development environment, one copy of ExacTicks is required for each software developer using the toolkit. If more than one developer will be using ExacTicks on single user systems, each developer must have a separate licensed copy of ExacTicks.

You may distribute ExacTicks object code, drivers, and dynamic link libraries with bona fide commercial applications subject to the following conditions:

1. The ExacTicks source code, object libraries, header files, and documentation may not be distributed.

2. The application must be an “end user” type of application, and not a “programmer’s toolkit” that would allow end users to construct their own executable applications that use ExacTicks.

Ryle Design, PO Box 22, Mt. Pleasant Michigan 48804 USA

Voice: 517.773.0587 Fax: 517.775.5530

Web Site: http://www.ryledesign.com

Info & Sales Email: info@ryledesign.com

Technical Support Email: support@ryledesign.com

Table of Contents

ExacTicks Introduction

Some important things to know before you start using ExacTicks

Welcome to **ExacTicks**, a comprehensive set of timing and scheduling tools for Windows 3.1, Windows 95, and Windows NT. ExacTicks is the definitive collection of software tools to measure and manage time, and provides the following types of timing components:

- **Timers** to measure time (just like a stopwatch), accurate to a microsecond.
- **Delays** to pause program execution, accurate to a microsecond
- **Alarms** to signal the passing of a specified time interval, accurate to a microsecond
- **Events** to schedule the execution of a user-written function or the posting of a user-specified message, accurate to a millisecond

All the functionality of ExacTicks is contained in a single DLL (dynamic link library) for 16 bit Windows, and another DLL for 32 bit Windows, making ExacTicks both easy to use and simple to distribute with the applications you develop. ExacTicks is compatible with the leading software development tools for Windows, and complete source code is included.

The most important thing to do first is to **read this manual carefully from beginning to end**. Many questions we get from developers using our tools are already answered in our documentation, and becoming thoroughly familiar with the ExacTicks documentation now will generally save valuable time later.

The next thing to do is to return the product registration card enclosed if you have not purchased ExacTicks directly from Ryle Design. Technical support and maintenance releases are only available to customers with a valid serial number, and unfortunately these days we must match the serial number to the registered user. If you purchased ExacTicks directly from Ryle Design then in most cases we have pre-registered you in our customer database, and no registration card will be enclosed. If you have purchased ExacTicks from an "online" vendor or CDROM (such as ComponentSource) and you do not have a floppy disk with a serial number, please let us know by email and we will assign you a serial number as soon as we confirm your purchase with the vendor.

Finally, after you've installed the ExacTicks components, carefully read the README file (available in both ASCII and HTML format) which is in the root directory of your ExacTicks installation. This contains the most recent technical information on ExacTicks, including corrections to the manual, new features that do not appear in the printed documentation, and newly reported problems or limitations and their workarounds.

Operating System Support

ExacTicks supports both 16 and 32 bit Windows application development with an identical API subject to the following limitations:

- 32 bit ExacTicks applications will not run under Windows 3.1 and Win32s.

- 16 bit ExacTicks applications will not run under NT.

ExacTicks will detect either of these two conditions and will not allow you to allocate ExacTicks resources, so the failure of ExacTicks under these conditions is considered orderly and "graceful".

Technical Support

Superior technical support is one of the reasons developers continue to purchase tightly focused software tools from small companies like Ryle Design. In most cases your technical questions will be answered directly by the main developer of the software - a level of support very difficult to get from companies like Borland and Microsoft. This level of attention comes with a price - your technical questions must be focused and to the point, preferably with a short example to demonstrate your problem. We are simply not able to advise developers on the proper installation of their compilers, the general design of their software, or the navigation of their compiler's menus and configuration options. Make sure you have run our test programs prior to contacting us, and be sure and know your ExacTicks serial number, the compiler and version you are using, the version of Windows you are developing for, and be in front of your computer if you are contacting us by telephone.

Email support is the most efficient support option available for both customer and vendor. We have a special email address for technical support questions only: **support@ryledesign.com**. We answer most email within one business day. If you would like to be notified of new revisions and maintenance releases, send email to **news@ryledesign.com** with the subject "Subscribe" and place a valid email address in the message body that you would like Ryle Design news sent to.

Fax support is available by faxing to (USA) 517.775.5530. Our fax is available 24 hours a day. Please include a return email address (if you have one) in your fax.

Telephone support is available by calling (USA) 517.773.0587. Our office hours are generally Monday-Friday from 10am to 4pm USA Eastern Time. We are generally closed Fridays during the summer months (June - August). Please use telephone support for installation and emergency technical questions only.

Our World Wide Web page is available at **www.ryledesign.com**. From our web page you can download maintenance releases and get the latest product news. Due to the complexity of the Windows software development environment periodic revisions issued as maintenance releases are a fact of life, so be sure and visit our web page from time to time to check for new revisions.

Finally, don't forget that the source code to the entire ExacTicks set of timing components is included. If you have a particular question on how a function operates don't hesitate to load up the source code in your text editor and dive in. Our source is written to be readable and maintainable, and even if C is not your primary language, the source to the ExacTicks DLL should be straightforward to follow.

ExacTicks Installation

ExacTicks is distributed on a single high-density floppy disk with a Windows hosted install program called **setup**. Executing the **setup** program (via the Program Manager or the Windows Explorer) will install ExacTicks in the path you specify. Assuming you specified \ticks as the target install path, your ExacTicks directory structure should look something like this:

```
\ticks          Root ExacTicks directory, home of the README file
\ticks\c            C/C++ header file, import libs, and test progams
\ticks\delphi   Delphi interface and and test programs
```

\ticks\dll ExacTicks dynamic link libraries and source
\ticks\test ExacTicks installation test programs
\ticks\vb Visual Basic interface and test programs

The install program will place the WINHRTxx.DLL dynamic link libraries in your system's main Windows directory.

The **very first thing to do now** is to read the README file in the ExacTicks root directory for updates and corrections to this manual. Note that this file is available in both text and HTML formats.

Next, run the test programs in the ExacTicks \test directory to make sure you have the DLLs moved to the correct place. TEST16.EXE tests WINHRT16.DLL and TEST32.EXE tests WINHRT32.DLL. You don't need to run TEST32 unless you are running Win95 or NT, and TEST16 will not run under NT. Successfully running TEST16 or TEST32 should cause the Windows Notepad to pop-up and display an ExacTicks timer and DLL status report, looking something like this:

```
      ExacTicks 1.1 © 1996,1997 Ryle Design  www.ryledesign.com

                              ExacTicks Win16 Test

     Timer Name       Activations   Elapsed (secs)    Average (secs)
-------------------  -----------  ----------------  ----------------
Microsecond Timer          10000          0.008313          0.000001

                           WINHRT16 DLL Resource Summary

DLL version 1.10-16 loaded xxx xxx xx 14:29:55 1997
Timer frequency : 1193180.00 Hz
Timer overhead  : 11 microseconds
Delay atom      : 5029 nanoseconds
Delay minimum   : 7 microseconds
Delay switch    : 250 microseconds
Alarm overhead  : 28 microseconds
Event resolution: 1 milliseconds
Timestamps used : 4  available: 507
Timers used     : 1  available: 126
Alarms used     : 1  available: 126
Delays used     : 1  available: 126
Events used     : 1  available: 30

ExacTicks report complete xxx xxx xx 14:29:58 1997
```

If the test programs do not run (remember TEST32 is for Win95 and NT only, and TEST16 is not compatible with NT) the install program has failed to place the ExacTicks DLLs in your Windows root directory. Do this yourself manually (using File Manager or Windows Explorer), test again, and please report this install failure to us via email to *support@ryledesign.com*..

Finally, the last step in the installation and testing of ExacTicks is to build a test application with your compiler of choice. The ExacTicks README file contains instructions on building applications with the compilers supported. Review the appropriate section in the file for your development tools and build the simple test program as instructed. Successful completion of this final step means ExacTicks is fully installed and ready to go to work for you.

Distributing ExacTicks

You may distribute the ExacTicks dynamic link libraries and related object code with any application you develop that uses ExacTicks. You many not distribute our source code, header files, or import

libraries. If you are doing consulting or development work for a customer that requires delivery of *all* source code, please contact us.

To include ExacTicks in an automated Windows install program, simply instruct the installer to load the appropriate ExacTicks DLL (either WINHRT16.DLL for 16 bit apps, or WINHRT32.DLL for 32 bit apps) into either your application's root directory, or the Windows root directory.

Removing ExacTicks

Removing ExacTicks from a development system is simple:

1. Delete the ExacTicks host directory.

2. Delete the ExacTicks dynamic link libraries (WINHRT16.DLL and WINHRT32.DLL) from the Windows root directory

ExacTicks makes no entries in any Windows .INI files or the Win32 Registry.

ExacTicks Technical Overview

Precision timing and event scheduling under Windows

ExacTicks relies on the timing and scheduling mechanism built into Windows 3.1, Windows 95, and Windows NT. In order to understand the capabilities of ExacTicks a technical review of the timing and event scheduling capabilities of Windows is in order

Technical Terms

In order to discuss the various timing and scheduling functions of Windows, we must first establish some technical definitions.

When using the term "**Windows**" we are referring the entire "family" of Windows platforms. This includes Windows 3.1, Windows For Workgroups, Windows 95, and Windows NT. If we use the term "**16 bit Windows"** or "**Win16**" we mean Windows 3.1 and Windows For Workgroups executing native 16 bit Windows applications. If we use the term "**32 bit Windows"** or "**Win32**" we mean Windows 95 and Windows NT executing native 32 bit Windows applications. Win32s, a run-time mechanism by which Windows 32 bit applications can execute on 16 bit Windows platforms, is not supported by ExacTicks, and none of our technical discussions apply to Win32s.

A "**timestamp**" is the fundamental time measurement component in ExacTicks. A timestamp retrieves a 64 bit count from the host systems timer controller. A timestamp has no relationship to the system's actual time of day clock, and there is no simple way to correlate a timestamp value to the actual time of day it was retrieved. Under NT a timestamp may also retrieve a value that is relative to the CPU time consumed by the host process or thread.

When using the term "**timer**" we are referring to the timer component in ExacTicks, which is roughly analogous to a stopwatch. We are not referring to a Windows native timer, which is quite a different thing, and should we need to reference a Windows native timer we will refer to it as such. A timer returns elapsed "wallclock" time. Under NT a timer may also return a value that is relative to the CPU time consumed by the host process or thread.

A "**delay**" is a software component that simply pauses execution of the current application or thread for a specified interval. Control of the executing code goes into the delay and only returns after a specified period of time has elapsed.

An "**alarm**" is a software component that is somewhat analogous to a alarm timer on a cooking stove. An ExacTicks alarm is set for certain time interval and checked periodically, and when the timer interval has passed it will inform the user checking the alarm that the time has expired. It is a synchronous device, meaning it must be checked, and not an asynchronous device, meaning it will not go off on its own and do something when the alarm has expired (this is what an *event* will do).

We use the term "**event**" to describe the asynchronous execution of a program module or delivery of a window message. An ExacTicks event is roughly analogous to an Interrupt Service Routine (ISR) under DOS, and is the same thing as a Windows Callback function.

An ExacTicks "**report**" contains a listing of all named timers that are active, with summary information on activations, elapsed time, and average activiation time. Optional information in the report provides a summary of the ExacTicks DLL resources available and calibration details.

Time Measurement

Precision time measurement under MSDOS is done by directly accessing the 8253/8254 timer controller found in all PC compatible systems. Ryle Design has published **PC Timer Tools** and **PC Timer Objects** for quite some time to accomplish this and provide a wide variety of other timing and scheduling functions. The same techniques used in MSDOS development cannot be used in the same fashion under Windows however, event though the same 8253/8254 timer controller is obviously still on the motherboard busily counting away. The 8253/8254 count can be retrieved, but different methods are required depending on the whether you are using 16 bit or 32 bit Windows

16 Bit Windows

In a 16 bit Windows environment the 8253/8254 timer controller is accessed via the INT 2F function interface. Passing AX=1684h and BX=5h returns the VTD (Virtual Timer Device) function dispatch address. Calling this address with AX=100h returns a 64 bit count from the VTD.

32 Bit Windows

In a 32 bit Windows environment the Microsoft API engineers provided two functions to read the high resolution timer in a platform independent manner (remember that Windows NT runs on systems other than the classic Intel PC platform). **QueryPerformanceFrequency()** returns the frequency that the system's precision timer ticks at, and **QueryPerformanceCounter()** returns a 64 bit count from the precision timer. Although Microsoft's Win32 API documentation indicates these functions are not supported under Windows 95, in fact they are.

Getting an accurate count from a system timer is only the first step in measuring time accurately - you must be able to do something useful with these 64 bit timestamps, and be able to calibrate the timing system to take into account how long it actually takes to retrieve a timestamp so that this additional time is not factored into the time calculations. Finally you need to mold these precision timestamps into timing components that are easy to add to your application under development. ExacTicks takes care of these requirements and presents you with timing components that share a common API under both Win16 and Win32, fully self-calibrating at run-time.

Event Scheduling

Although MSDOS is not a multi-tasking system, it is possible to cause asynchronous program execution to occur at precise intervals by hooking either interrupt 08 or 1C and installing a user written Interrupt Service Routine. Ryle Design's **PC Timer Tools** and **PC Timer Objects** have been providing these types of services under MSDOS for years. Windows on the other hand has functions in the API that are designed to closely approximate the same type of functionality as a DOS timer ISR but without the trauma of writing and testing a DOS ISR. There are five functions in both the Win16 and Win32 API for scheduling and managing events: **timeGetDevCaps**, **timeBeginPeriod**, **timeEndPeriod**, **timeSetEvent**, and **timeKillEvent**. A description of these functions are not required to use ExacTicks, but interested readers are encouraged to examine the Windows API documentation for further details.

Although the event scheduling functions are the same under all the Windows platforms, the way they are implemented and what a scheduled event may do when invoked are quite different in Win16 and Win32.

16 Bit Windows

Under Win16 an event is invoked via a hardware interrupt, and will occur with excellent accuracy regardless of most system activity. The event that is invoked must reside in fixed data and code segments, meaning it must reside in a dynamic link library. The event has an extremely limited set of Windows API functions it can call: **PostMessage**, **timeGetSystemTime**, **timeGetTime**, **timeSetEvent**, **timeKillEvent**, **midiOutShortMsg**, **midiOutLongMsg**, and **OutputDebugString**.

32 Bit Windows

Under Win32 an event is invoked via a thread dedicated to scheduling events, and it can occur with accuracy ranging from very good to very poor, largely depending on the level of other system activity. The event invoked does not have to be in fixed data and code segments, so it no longer must live in a DLL as it does under Win16. The Win 16 restrictions on Windows API calls that the event can reference have also been lifted, although Microsoft cautions that the scheduled event "*should not call functions that take a long time to complete. Calls to such functions may interfere with other uses of the timeXXX functions because TimeProcs are all called from the same thread within a specified process*".

To summarize the event scheduling capabilities of Windows we would say the following:

1. Win16 offers superior accuracy in scheduling events but suffers from severe limitations on the implementation of the scheduled event and what it can do when invoked.

2. Win32 offers less precise event scheduling but much greater flexibility in implementation of the scheduled event code and what it can do when invoked.

3. Whether developing for Win16 or Win32 the scheduled event should be of the shortest duration possible, accomplishing only the minimum set of tasks that must be performed in pseudo real-time.

Events are a very restricted resource under all the Windows platforms. Here is Microsoft's table that details the number of events that can be scheduled under the various Windows platforms:

```
Operating System      16-bit app   32-bit app
-------------------------------------------------------------
Windows 3.11               8           0 (Not supported on Win32s)
Windows 95                32          32
Windows NT                16          16 per process
```

The following **ExacTicks Cookbook** chapter deals with event scheduling in much greater detail.

Bibliography

Complex subjects like time measurement and event scheduling as implemented by the various Windows platforms require extensive technical references to gain a good understanding of techniques and limitations. The best and most complete reference we have found is the **Microsoft Developers Network Level 1 CDROM**, which is available by subscription from Microsoft for $200/year for four quarterly updates, or for $99 for a single copy. This publication has extensive technical papers and background white papers on many diverse subjects including timing and event scheduling. Some of the more interesting articles included at the time of this writing are:

“Timers and Timing Under Windows” explains the Win16 VTD interface.

“Overcoming Timer-Latency Problems In Midi Sequencers” gives an excellent overview of event scheduling limitations under Win32.

“Availability of Multimedia Timers” documents the availability of callback timers used by ExacTicks to schedule events.

“High Precision Timing Under Windows, Windows NT, and Windows 95” explains the event scheduling mechanism and limitations used by the various Windows platforms.

There are many more useful articles on the MSDN CDROM, and if you are doing anything more than casual Windows development, this CDROM is well worth the investment.

The other main source of technical information these days is the comp. Usenet newsgroups, and the new support newsgroups run by Microsoft from their own server (nttp://news.microsoft.com). Our experience with information passed from developer to developer via the newsgroups is that the information is of highly variable quality and accuracy, and the accurate source most often quoted comes from an article on the MSDN CDROM, so buying the MSDN CDROM first will save you a lot of time wading through the noise of the Usenet.

ExacTicks Cookbook

How to use the ExacTicks timing components in application development

Providing a toolkit of software functions for an application developer to use is more than just documenting the functions and saying "Here you go - you're on your own". A tool like ExacTicks is only useful if an application developer can use it to quickly solve a problem or requirement in an efficient manner. Many third party software tools use examples as their main tutorial method, leaving it mostly to the tool user to wade through all the example code to find something that looks like what they are trying to do, and then simply use that code (whether they understand it or not). A big problem with Windows hosted example programs is that the application framework used to develop the example code (be it OWL, MFC, VB, or VCL) clogs up the example with large amounts of extraneous code just to start and maintain the GUI.. Our solution to this problem is via this "cookbook" section, which features "snippets" or short code examples to demonstrate proper usage of a particular timing component. These code snippets are not meant to be compiled but rather are for tutorial purposes only.

A word about our ExacTicks DLLs: the ExacTicks DLL export nearly all of their functions, making them available for you to use. All of these exported functions are documented in the ExacTicks Reference section, but for most timing tasks only a small subset of these functions will be needed. Don't be overwhelmed by the 70 or so functions in the ExacTicks DLL - only a few from each component group are needed to solve various timing problems.

Some Useful Things to Know

There are three subjects worth reviewing before diving into the operation of ExacTicks. First, ExacTicks exists primarily as a DLL (**D**ynamic **L**ink **L**ibrary) and the data allocated and used by the DLL is treated differently under Win16 and Win32. Second, ExacTicks makes extensive use of 32 bit unsigned integers, which are not well supported by Delphi and Visual Basic. Finally, there is a fundamental time measurement restriction under 16 bit Windows.

DLL Data Sharing

When you pass a timer, timestamp, delay, alarm, or event "handle" into the ExacTicks DLL you are really passing an index into a data structure that ExacTicks allocates and uses to keep track of the timing components in use. Under Win16 these data structures reside in a data segment that is shared by all applications that have loaded the ExacTicks DLL. This means that the timing resources provided by ExacTicks are shared by all 16 bit Windows applications that use ExacTicks, and all of the named timers in use by the various applications will show up on the timer report. This also means that any changes one Win16 application makes to the various ExacTicks calibration parameters will affect all Win16 apps that are using ExacTicks. Under Win32 the DLL's data segment is not shared, but rather each application that loads the ExacTicks DLL will have it's own private set of ExacTicks timing resources and calibration settings. This means that ExacTicks timing resources are not shared, and changing an ExacTicks calibration setting will not affect the settings in use by other applications that are using ExacTicks.

One important note: the ExacTicks event components use a Windows system resource that is subject to some very stringent availability restrictions regardless of whether you are using a 16 or 32 bit DLL. These restrictions are less under Win32 than under Win16, but they still exist and may impact your

ability to have numerous events scheduled. The previous section ("*ExacTicks Technical Overview*") details these restrictions.

32 Bit Unsigned Integers

ExacTicks uses 64 bits of internal precision to calculate time differences but returns time measurement values using a 32 bit unsigned integer, as support for 64 bit integers is currently only available using Win32 and certain compilers. Delphi and Visual Basic do not support even a 32 bit unsigned integer, so Delphi and Visual Basic users must be aware of the range limitations they will encounter when using ExacTicks functions that return either LongInt (Delphi) or Long (Visual Basic) values. The maximum meaningful value returned will be 2,147,483,647 microseconds, milliseconds, or seconds. There are two useful ways to circumvent this limitation: (1) use the ExacTicks timer report, which is generated by the ExacTicks DLL and thus handles 32 bit unsigned integers, so the times reflected in the timer report will have full resolution, and (2) use the timer functions that return a double precision floating point number rather than an integer value.

Win16 Timer Interval Restriction

Due to the way ExacTicks calculates time intervals and the way Win16 returns a high resolution timestamp value, a single timed interval using either the ExacTicks timestamp or timer components may not exceed 4,294,977,287 microseconds, 4,294,977 milliseconds, or 4,294 seconds. This is approximately 71 minutes. Under Win32 the limits are the same for all resolutions: 4,294,977,287 microseconds, milliseconds, or seconds.

Timestamps

Timestamps are the basic element of time measurement in ExacTicks. They are used by most of the other ExacTicks components to acquire a very high resolution time snapshot from the host system, and other timestamp functions exist to calculate the time difference between two timestamps. *Most developers will not need to use the timestamp components directly - measuring time intervals is better accomplished using the **timer** component.*

Here is a list of the timestamp functions and what each one does:

hrt_timestamp_alloc	Allocates a timestamp and returns a timestamp handle
hrt_timestamp_avail	Returns the number of timestamps available
hrt_timestamp_free	Frees a timestamp and invalidates the handle
hrt_timestamp_get	Gets a timestamp from the system timing system
hrt_timestamp_inuse	Returns the number of timestamps in use
hrt_timestamp_msecdiff	Calculates the difference between two timestamps in milliseconds
hrt_timestamp_secdiff	Calculates the difference between two timestamps in seconds
hrt_timestamp_usecdiff	Caluclates the difference between two timestamps in microseconds

Suppose you wish to measure a time interval using a timestamp, and return the interval value in units of microseconds. Here's an example in **C**:

```
// allocate some storage for our our timestamp handles and time interval

short   stamp1, stamp2;
DWORD       timelen;

// allocate our timestamps - we will assume some are available so no error
checking

stamp1 = hrt_timestamp_alloc(HRT_WALLCLOCK);
stamp2 = hrt_timestamp_alloc(HRT_WALLCLOCK);

// now start the interval to be measured
```

```
hrt_timestamp_get(stamp1);
//
// do something here ...
//
// now get the second timestamp to mark the end of the timed interval
hrt_timestamp_get(stamp2);

// now calculate the length of the interval in microseconds
timelen = hrt_timestamp_usecdiff(stamp1, stamp2);
// all done - free the timestamps
hrt_timestamp_free(stamp1);
hrt_timestamp_free(stamp2);
```

The same code (minus most of the comments for brevity) in **Delphi**:

```
var
        stamp1, stamp2      : word;
        timelen             : LongInt;
begin
        stamp1 := hrt_timestamp_alloc(HRT_WALLCLOCK);
        stamp2 := hrt_timestamp_alloc(HRT_WALLCLOCK);

        hrt_timestamp_get(stamp1);
        { something happens }
        hrt_timestamp_get(stamp2);
        timelen := hrt_timestamp_usecdiff(stamp1,stamp2);

        hrt_timestamp_free(stamp1);
        hrt_timestamp_free(stamp2);
end;
```

Finally, the example (again minus most of the comments) in **Visual Basic**:

```
Dim stamp1, stamp2, timelen

stamp1 = hrt_timestamp_alloc(HRT_WALLCLOCK)
stamp2 = hrt_timestamp_alloc(HRT_WALLCLOCK)

hrt_timestamp_get stamp1
‘ something happens
hrt_timestamp_get stamp2
timelen = hrt_timestamp_usecdiff(stamp1,stamp2)

hrt_timestamp_free stamp1
hrt_timestamp_free stamp2
```

In these examples we are passing **HRT_WALLCLOCK** as the parameter to **hrt_timestamp_alloc**, meaning the timestamp will measure elapsed “wallclock” time. Under NT it is possible to measure process or thread CPU time - the flags **HRT_PROCESS** and **HRT_THREAD** are provided for this. These flags are ignored under Win16 and Win95.

That's about all there is to using a timestamp. Unlike a timer, timestamp calculations do not factor the amount of time it takes to retrieve the timestamps into the interval calculation, so the overhead of the timestamp retrieval appears in the interval result.

Timers

Timers are much more flexible and powerful than timestamps, and an ExacTicks timer is very similar to a chronograph (a very powerful type of stopwatch) in operation and capabilities. A timer keeps track of the number of times it has been started and stopped, it accumulates total elapsed time, time for the last start/stop sequence, and elapsed time since started if running. A timer, unlike a timestamp, is assigned a resolution when allocated, and the developer has a choice of microsecond, millisecond, or single second resolution. Finally, a timer may be given a name and if named the timer will appear in a timer report that can be generated to summarize the activity of all named timers active in the system. This timer report is very useful if multiple timers are being used to measure events or profile code.

Here is an alphabetical summary of all the timer functions:

hrt_timer_alloc	Allocates a timer, assigns it a resolution, optionally gives it a name, and returns a handle to the timer
hrt_timer_avail	Returns the number of timers available for allocation
hrt_timer_count	Returns the number of times a timer has been activated
hrt_timer_current	Returns the current elapsed time on the timer since it was last started in the timer's unit of resolution
hrt_timer_currentsecs	Returns the current elapsed time on the timer since it was last started as a float value in seconds
hrt_timer_elapsed	Returns the elapsed time accumulated on the timer in the timer's unit of resolution
hrt_timer_elapsedsecs	Returns the elapsed time accumulated on the timer as a float value in seconds
hrt_timer_free	Frees a timer and invalidates the timer's handle
hrt_timer_getname	Returns the name of the timer
hrt_timer_getohead	Returns the overhead of a timer start/stop sequence as calibrated by the timer engine
hrt_timer_getresolution	Returns the timer's resolution as assigned to it when it was allocated
hrt_timer_getstate	Returns the timer's state, which may be stopped, running, or suspended
hrt_timer_getstatus	Returns the timer's status, which will flag any timer logic errors such as trying to start a timer already running
hrt_timer_inuse	Returns the number of timers allocated and in use
hrt_timer_lastelapsed	Returns the elapsed time of the last start/stop sequence in the timer's unit of resolution
hrt_timer_lastelapsedsecs	Returns the elapsed time of the last start/stop sequence as a float value in seconds
hrt_timer_reset	"Zeros" a timer, resetting activation and elapsed time counters
hrt_timer_resume	Resumes a suspended timer
hrt_timer_setname	Gives a name to a timer already allocated
hrt_timer_setohead	Overrides the timer system overhead calibration value
hrt_timer_start	Starts a timer running
hrt_timer_stop	Stops a running timer
hrt_timer_suspend	Suspends a running timer

Although there are 23 timer functions listed, only a few are required to measure time and accumulate timing statistics. Suppose we want to measure a system's floating point multiplication performance. Here's how we do it in **C**:

```
// allocate some storage
short   timer1, indx;
```

```
double ave_mult, alpha;
// allocate a microsecond timer, assume there is one available so no error checking
timer1 = hrt_timer_alloc(HRT_MICROSECOND,””); // timer is not named
// now loop 10,000 times and measure
for (indx=0; indx<10000; indx++)
{
        hrt_timer_start(timer1);
        alpha = 123456.789 * 987.654321;
        hrt_timer_stop(timer1);
}
// calculate the average time for each multiply
ave_mult = hrt_timer_elapsedsecs(timer1) / hrt_timer_count(timer1);
// all done - free the timer
hrt_timer_free(timer1);
```

The same example (minus the comments for brevity) in **Delphi**:

```
var
        timer1, indx    : word;
        ave_mult, alpha        : double;
begin
        timer1 := hrt_timer_alloc(HRT_MICROSECOND,’’);
        for indx := 1 to 10000 do
        begin
                hrt_timer_start(timer1);
                alpha := 123456.789 * 987.654321;
                hrt_timer_stop(timer1);
        end;
        ave_mult := hrt_timer_elapsedsecs(timer1) / hrt_timer_count(timer1);
        hrt_timer_free(timer1);
end;
```

Finally, the same example (again minus the comments) in **Visual Basic**:

```
dim     timer1, indx, ave_mult, alpha

timer1 = hrt_timer_alloc(HRT_MICROSECOND,””);
for indx = 1 to 10000
        hrt_timer_start timer1
        alpha = 123456.789 * 987.654321
        hrt_timer_stop timer1
next indx
ave_mult = hrt_timer_elapsedsecs(timer1) / hrt_timer_count(timer1)
hrt_timer_free timer1
```

The floating point multiply in this example is going to be of very short duration on a system with an FPU, so it is important that the overhead of the timer start and stop calls is not part of the timing calculation. ExacTicks calibrates the timers so that the time required to start and stop is removed when the timer’s elapsed time is updated.

In this example we retrieved the elapsed time accumulated on the timer using the function **hrt_timer_elapsedsecs**, which returns a floating point value in units of seconds. We also could have used the **hrt_timer_elapsed** function to return an integer value in units (in this case) of microseconds.

There are a couple of things to keep in mind about how the timers update their elapsed time. The functions **hrt_timer_elapsed**, **hrt_timer_elapsedsecs**, **hrt_timer_lastelapsed**, and **hrt_timer_lastelapsedsecs** return accumulated elapsed time *as of the last time the timer was stopped*. This means that if a timer was started, stopped, and started again, the elapsed time returned will not include the new time accumulated on the timer since it was last started - the timer would have to be stopped again for this last time interval to be added to the timer's accumulated time. In order to retrieve the amount of time accumulated on a running timer since it was last started use the functions **hrt_timer_current** and **hrt_timer_currentsecs**.

The **hrt_timer_suspend** and **hrt_timer_resume** functions stop and restart a timer without incrementing the timer's activation count - useful if the timer's activation count is an important statistic. The function **hrt_timer_reset** works just like the reset button on a stopwatch - all counts are zeroed and the timer state is reset to stopped.

Under NT it is possible to measure process or thread CPU time in instead of "wallclock" time. The flags **HRT_PROCESS** and **HRT_THREAD** are provided for this and may be logically ORed with the resolution parameter flag when allocating a timer with **hrt_timer_alloc**. These flags are ignored under Win16 and Win95.

Finally, if you are using the timer report (explained in a later section) to display timer statistics, do not deallocate the timer with **hrt_timer_free** until after the report has been generated, and don't forget to assign the timer a name when it is allocated. Only allocated timers with names show up in the timer report.

Delays

Delays are very useful timing components to "pause" program execution for a specified period. When executing a delay the application's execution goes into the delay and doesn't return until the delay completes - it is not possible for the application to be doing anything else while the delay is executing (use an alarm if you need that capability).

There are nine functions available to generate and manage delays, as follows:

hrt_delay_alloc	Allocates a delay, sets the resolution and duration, and returns a handle to the delay
hrt_delay_avail	Returns the number of delays available for allocation
hrt_delay_do	Executes a delay
hrt_delay_free	Frees a delay previously allocated and invalidates the delay handle
hrt_delay_getatom	Returns the fundamental unit of the delay engine
hrt_delay_getmin	Returns the minimum delay available in microseconds
hrt_delay_getswitch	Returns the delay to alarm threshold in microseconds
hrt_delay_inuse	Returns the number of delays allocated
hrt_delay_setswitch	Sets the delay to alarm threshold

There are only three functions required to use and generate delays: **hrt_delay_alloc**, **hrt_delay_do**, and **hrt_delay_free**. The following example generates a 100 microsecond delay using **C**:

```
// allocate storage for the delay handle

short  delay1;
```

```
// allocate a microsecond delay, set to 100 microseconds
// do this ahead of time, before you actually need to use the delay
delay1 = hrt_delay_alloc(100L, HRT_MICROSECOND);
//
//  code to do various things
//
// time to use the delay
hrt_delay_do(delay1);
// when done, free the delay handle
hrt_delay_free(delay1);
```

The same example for **Delphi**:

```
var
        delay1 : word;
begin
        { allocate our delay before we need it }
        delay1 := hrt_delay_alloc(100,HRT_MICROSECOND);

        { various code to do things }

        { now use the delay }

        hrt_delay_do(delay1);

        { free the delay when no longer needed }

        hrt_delay_free(delay 1)
end;
```

Finally, the same example for **Visual Basic**:

```
dim     delay1

‘ allocate the delay before we need it
delay1 = hrt_delay_alloc(100,HRT_MICROSECOND)

‘ other code here to do things
‘
‘ now time to use the delay
hrt_delay_do delay1

‘ free delay when no longer needed
hrt_delay_free delay1
```

As these brief examples show, delays are simple to manage and use.

Alarms

Alarms provide a mechanism to measure a certain passage of time while also executing other code. An alarm is set for an interval, then checked periodically (typically in a loop) for expiration while other

processing continues. Alarms are very useful for timeout functions, where a piece of hardware is polled for a response, and the polling loop exits either by response from the device or after a specified period has elapsed.

There are 10 functions to manage and execute alarms:

hrt_alarm_alloc	Allocates an alarm, sets resolution, and returns a handle
hrt_alarm_avail	Returns the number of alarms available for allocation
hrt_alarm_cancel	Cancels a pending alarm and sets the alarm valid flag to false. Invalid alarms always return immediately when checked.
hrt_alarm_check	Checks an alarm for expiration.
hrt_alarm_free	Frees an allocated alarm and invalidates the alarm handle
hrt_alarm_getohead	Returns the alarm overhead in microseconds
hrt_alarm_inuse	Returns the number of alarms in use
hrt_alarm_reset	Resets an alarm, which restarts the countdown process
hrt_alarm_set	Sets an alarm for a certain interval and begins the countdown process
hrt_alarm_setohead	Overrides the alarm overhead established by the alarm calibration routine

Alarms have many uses, limited primarily by your imagination. The “classic” use for an alarm is to provide an exit mechanism for a loop. This is often used when polling a device for a specific length of time or at a specific rate. An example of a time-limited processing loop follows, where the loop will execute for 250 milliseconds. First in **C**:

```
// allocate storage for our alarm
short   alarm1

// allocate a millisecond resolution alarm
alarm1 = hrt_alarm_alloc(HRT_MILLISECOND)

// now set the alarm and enter the loop
hrt_alarm_set(alarm1, 250L, HRT_ONCE);
do
{
        // some processing of some sort here
}
while (hrt_alarm_check(alarm1) == 0L);

// all done later - free the alarm
hrt_alarm_free(alarm1)
```

The same example in **Delphi**:

```
var
        alarm1: word;
begin
        alarm1 := hrt_alarm_alloc(HRT_MILLISECOND);
        hrt_alarm_set(alarm1,250,HRT_ONCE);
        repeat
                { some processing here }
        until (hrt_alarm_check(alarm1) <> 0);
        hrt_alarm_free(alarm1)
end;
```

Finally, the same example in **Visual Basic**:

```
dim     alarm1
```

```
alarm1 = hrt_alarm_alloc(HRT_MILLISECOND)
hrt_alarm_set alarm1, 250, HRT_ONCE
do
        ' some processing here
loop until (hrt_alarm_check <> 0)
hrt_alarm_free alarm1
```

Some additional alarm functions and capabilities are not demonstrated by this simple example but are certainly worth noting. First, when an alarm is checked using **hrt_alarm_check** it returns zero if the alarm has not expired, or the amount of total time elapsed since the alarm was set if the alarm has expired. This provides a handy mechanism to determine if you are checking your alarm often enough to get the accuracy you require. For example: in the above sample code the **hrt_alarm_check** function may return 275 when it finally expires - this means 275 milliseconds have elapsed since the alarm was set, which may mean that the processing you are doing inside the alarm loop may be excessive if you want the loop to run exactly for 250 milliseconds. The alarm in this example was a "one shot" type, meaning it counts once and when checked and found to be expired it is then no longer a valid alarm (invalid alarms always return immediately with the value 1) and must be reset before being used again. Another type of alarm is the periodic alarm, which automatically resets itself each time it is checked and found to be expired. This feature is very powerful and can be used to create self-sustaining loops for data acquisition and process control. Finally, an alarm can be reset using **hrt_alarm_reset**, which is a quick way of reseting the alarm to its original value and starting the counting process again, while **hrt_alarm_cancel** invalidates ("cancels") an active alarm.

In summary, alarms can be used in a wide variety of configurations, and their auto-reset capability make them ideal for data acquisition loops and pseudo event scheduling applications.

Events

The ExacTicks event scheduler is a very powerful mechanism to schedule asynchronous functions or messages ("events") to occur at a specified wallclock time, once after a certain interval, or at a periodic interval. An ExacTick event is roughly analogous to a timer interrupt under MSDOS, and under Win16 has a very similar set of restrictions in terms of what it can do when invoked. The development environments supported by ExacTicks support ExacTicks events with varying degrees of success.

Events can be scheduled in one of several ways:

- An event can be scheduled to occur after a predetermined period of time with millisecond resolution. The event may occur once or may continue to occur at the same periodic rate until canceled.

- An event can be scheduled to occur after a predetermined period of time with single second resolution. The event may occur once or may continue to occur at the same periodic rate until canceled

- An event can be scheduled to occur at a specified wallclock time, accurate to one second. The event may occur once or may occur again at the same time on following days until canceled.

When an event occurs, one of two things may happen:

- A function may be invoked

- A message may be posted to a specified window.

The restrictions on events are as follows:

- Under Win16 a function called by an event must reside in fixed code and data segments inside a DLL. There is a very short list of Windows API calls that can be safely made (see the Event Scheduler section in the ExacTicks Technical Overview for more details), so the event function will have limited usefulness in interacting with the Windows system environment.

- Under Win32 a function called by an event may reside anywhere in an application (the DLL restriction has been removed) and although the restrictions on Windows API calls have also been removed, it is *highly recommended* that the event function be short and do the minimal processing required to accomplish the task at hand.

- If the ExacTicks event is posting a message rather than calling a function, there are no limits on the messages posted under Win16 or Win32.

Clearly the idea of sending a message to an application window under Win16 was designed so the event can communicate with an idle application, signaling it in a very precise manner. The foreground idle application can then do whatever processing is required without any restrictions on Windows API calls or having to reside in a user written DLL.

There are 13 functions to create and manage events. They are as follows:

hrt_event_alloc	Allocates an event and returns an event handle
hrt_event_avail	Returns the number of event handles available for allocation
hrt_event_cancel	Cancels a pending event
hrt_event_clockcallback	Schedules a function to be invoked at a specified time of day
hrt_event_clockmsg	Schedules a message to be sent to a window at a specified time of day
hrt_event_free	Frees an allocated event and invalidates an event handle
hrt_event_inuse	Returns the number of event handles in use
hrt_event_maxperiod	Returns the maximum period for millisecond resolution events
hrt_event_minperiod	Returns the minimum period for millisecond resolution events
hrt_event_pending	Reports whether an event is scheduled and waiting for delivery
hrt_event_reset	Reschedules an event
hrt_event_setcallback	Schedules a function to be invoked after a specified time interval
hrt_event_setmessage	Schedules a message to be sent to a window after a specified time interval

The first decision to make when using an ExacTicks event is whether you want the event to occur once or on a periodic basis. The **hrt_event_alloc** function allocates either an event of type HRT_ONESHOT (occurs once only) or of type HRT_PERIODIC (occurs periodically until canceled). The next decision is what resolution your event should have, again set by the **hrt_event_alloc** function: HRT_MILLISECOND allocates an event scheduled on a time interval basis with millisecond resolution, HRT_SECOND allocates an event scheduled on a time interval basis, and HRT_WALLCLOCK schedules an event to occur at a specified system time.

Once the event is allocated, you may schedule it to either call a function or send a message. If the event was allocated with millisecond or second resolution, **hrt_event_setcallback** will schedule a function to be executed, while **hrt_event_setmessage** will schedule a message to be sent. If the event was allocated with HRT_WALLCLOCK resolution in order to do something at a given system time, **hrt_event_clockcallback** will schedule a function to be executed, and **hrt_event_clockmsg** will schedule a message to be sent. Note that these functions may fail, even though an event handle was allocated successfully, if a Windows multimedia timer is not available, so be sure and test the return values of these functions.

Once the event is scheduled, **hrt_event_pending** will tell you whether the event is pending delivery or not, and **hrt_event_cancel** will cancel a pending event. **hrt_event_reset** will reset either a pending or canceled event to the previous interval it was set to.

When scheduling millisecond resolution events there are minimum and maximum time intervals the host Windows system supports, which are returned by **hrt_event_minperiod** and **hrt_event_maxperiod** respectively. On most systems the values returned are 1 and 65535 milliseconds respectively.

Finally, when you no longer require the event you allocated, free the ExacTicks event data structure with **hrt_event_free**.

Example: Scheduling A Function

Firing off a user-written function via the ExacTicks event scheduler is a very powerful tool especially useful for data acquisition and process control applications. The following example shows how to schedule a function to go off every 250 milliseconds.

C Example

The function we wish to call is named **dll_event**, and looks like this under Win16:

```
void CALLBACK _export dll_event(short id, DWORD userdata)
//
//  This is the event function that gets scheduled and called
//  by the WINHRT16.DLL
//
//  First parameter id is the event handle
//  Second parameter userdata is user defined data passed when the event was
scheduled
//
{
        // do something useful here!

} // dll_event
```

Under Win16 this function must reside in a DLL with fixed code and data segments. Under Win32 the function is declared the same but without the _export qualifier, and the function may reside in any module of the application - it does not need to be in a DLL.

The following code schedules this function to go off every 250 milliseconds until canceled.

```
        short   istat, event1;

        // allocate an event handle
        event1 = hrt_event_alloc(HRT_PERIODIC,HRT_MILLISECOND);

        // schedule the event
        istat = hrt_event_setcallback(event1,250L,(FARPROC) dll_event,0L);
        if (istat == HRT_FALSE)
        {
                // error handling code here - no callback timers were available
        }

        // the dll_event function is now going off every 250 milliseconds ...

        // code to do things you need to do ....

        // now time to cancel the event
```

```
hrt_event_cancel(event1);

// all done ... free the event resource and invalidate the handle

hrt_event_free(event1);
```

As you can see, scheduling an event function is very straightforward, and the only tricky bit is having to place the event function inside a DLL under Win16. All of the C compilers supported by ExacTicks have extensive documentation on how to build a DLL - just don't forget to mark the code and data segments as FIXED.

Delphi Example

This code works basically the same way in Delphi. There is a predefined procedure type known as **hrt_event** that is defined as:

```
type hrt_event = procedure(event_id : word; userdata : LongInt)
```

The scheduled function should be declared with the same parameter types, and (in Win16) placed in a DLL and exported. The code that schedules the event declares a variable of type **hrt_event** and equates the address of the DLL function to this variable. This variable, which is actually a pointer to the DLL function, is passed along with the other required parameters to the appropriate ExacTicks event scheduler function. This all looks something like this:

```
procedure dll_event(p1 : word; p2 : LongInt); external;  { in DLL under
Win16 }
var
        istat, event1   : word;
        event_proc      : hrt_event;
begin
        event_proc := dll_event;
        event1 : = hrt_event_alloc(HRT_PERIODIC,HRT_MILLISECOND);
        istat := hrt_event_setcallback(event1,250,event_proc,0);
        if (istat = HRT_FALSE) then
        begin
                { no timers available - handle the error here }
        end;

        { dll_event now being called every 250 milliseconds }

        { later ... all done with event }
        hrt_event_cancel(event1);
        hrt_event_free(event1);
end;
```

Visual Basic Example

Unfortunately Visual Basic does not support passing a function address, so there is no way to schedule a function using ExacTicks and VB. There are three indirect ways around this problem

1. Use an alarm driven loop to call a function you wish to schedule periodically while doing other processing.

2. Write a DLL in C or Delphi to host the scheduled event function and have the DLL call the ExacTicks event scheduler to schedule the function.

3. Use the Visual Basic **Timer** control, which gives some of the capabilities of the ExacTicks event scheduler but with much less accuracy.

The first workaround is obviously much simpler than the second, while the third will not provide the same level of precise accuracy as ExacTicks.

Example: Scheduling A Message

Scheduling a message to be sent to an idle Windows application allows the application to be notified on a fairly precise basis that it is time to do something. Under Win16 this scheme eliminates the restrictions placed on scheduled functions, so the application that receives the message is free to use any Windows API function needed.

Messages sent to Windows applications contain the message value and two additional parameters known as the "wParam" and the "lParam". When an ExacTicks event sends the message, the wParam contains the number of times the event tried and failed to post the message, and the lParam contains user data you passed to the ExacTicks message scheduling function. If the wParam parameter is sent with a non-zero value, this means that the application window message queue was full one or more times when it was time to send the message and the message could not be sent. This value increments for every successive failure to send the message, and then is reset to zero after the message is sent successfully again.

Our code example will be similar to the previous example, except that we will send a message every 250 milliseconds rather than schedule a function.

C Example

```
#define   WM_OURMESSAGE         1024   // define our message value
short   istat, event1;
HWND ourwindow;

// allocate an event handle
event1 = hrt_event_alloc(HRT_PERIODIC,HRT_MILLISECOND);

// get the window handle for the app window you want to transmit to
ourwindow = SomeWindowHandle;

// schedule the event
istat =
hrt_event_setmessage(event1,250L,ourwindow,WM_OURMESSAGE,0L);
if (istat == HRT_FALSE)
{
        // error handling code here - no callback timers were available
}

// the message is being sent to the window every 250 milliseconds ...

// code to do things you need to do ....

// now time to cancel the event

hrt_event_cancel(event1);
```

```
// all done ... free the event resource and invalidate the handle

hrt_event_free(event1);
```

The only nebulous bit here is where the application's window handle comes from. This will be completely dependent on what, if any, application framework you are using, so this is somewhat indeterminate in this example. The processing of the message will also be completely dependent on what application framework you are using.

Delphi Example

The same code in Delphi, minus most of the comments for brevity:

```
const
        WM_OURMESSAGE = 1024;
var
        istat, event1   : word;
begin
        event1 : = hrt_event_alloc(HRT_PERIODIC,HRT_MILLISECOND);
        istat :=
hrt_event_setmessage(event1,250,SomeForm.Handle,WM_OURMESSAGE,0);
        if (istat = HRT_FALSE) then
        begin
                { no timers available - handle the error here }
        end;
        { message now being posted every 250 milliseconds }

        { later ... all done with event }
        hrt_event_cancel(event1);
        hrt_event_free(event1);
end;
```

Processing the message involves adding a message handler for the WM_OURMESSAGE message to the form that receives the message. You place a declaration for the message handler in the form's private declaration section:

```
procedure OurMessage (var message: TMessage) ; message
WM_OURMESSAGE;
```

Then add the OurMessage procedure to the form and do what you want to do. That's all there is to it!

Visual Basic Example

Once again, Visual Basic has severe limitations when dealing with asynchronous events. In this case Visual Basic has no mechanism to explicitly process messages - the entire Visual Basic message processing engine is hidden from the application developer, so it is not possible using Visual Basic alone to make an application respond to user-defined messages in a user-defined manner. A employee of Microsoft developed a very popular VBX known as the "Message Blaster" which was placed into the public domain and is included in the ExacTicks distribution disk (look in the \vb directory). While this takes care of 16 bit VB, there is no corresponding 32 bit OCX Message Blaster in the public domain, so at this time there is no way to use ExacTicks to send a message to a 32 bit VB app. We will be looking into this problem more closely to find a way to do this (or cook up our own OCX for you to use) so check the README.TXT file for any new fixes or workarounds to this problem and please contact us if you need this capability and are willing to test a solution if one is not yet provided in the README.TXT file.

A Win16 implementation of scheduling a message looks like this:

```
global const WM_OURMESSAGE = 1024
dim istat, event1

event1 = hrt_event_alloc(HRT_PERIODIC,HRT_MILLISECOND)
istat  = hrt_event_setmessage(event1, 250, SomeForm.hWnd,
WM_OURMESSAGE,0)
if istat = HRT_FALSE then
  ' no timers available - handle the error here
end if

' message now being posted every 250 milliseconds

' later ... all done with event
hrt_event_cancel event1
hrt_event_free event1
```

The Message Blaster VBX documentation gives complete details on how to use the VBX to respond to a specific message sent to a form.

Miscellaneous Functions

There are a number of useful functions in the ExacTicks DLL that provide some miscellaneous services. They are as follows:

hrt_calibrate_alarm	Calibrates the ExacTicks alarm logic
hrt_calibrate_delay	Calibrates the ExacTIcks delay logic
hrt_calibrate_event	Calibrates the ExacTIcks event scheduler logic
hrt_calibrate_timer	Calibrates the ExacTicks timer logic
hrt_cpu_process	Returns host process CPU time under NT
hrt_cpu_thread	Returns host thread CPU time under NT
hrt_isNT	Detects NT host for Win32 apps
hrt_report	Generates an ExacTicks timer and DLL status report
hrt_supported	Returns whether the host Windows platform supports ExacTicks or not
hrt_timestring	Converts a floating point value into a standard time format

The **hrt_calibrate_xxxxx** functions calibrate various parts of the ExacTicks timing logic. Normally this is done when the first alarm, delay, event, or timer is allocated by an application that loads ExacTicks, so there is usually no need for an application to explicitly call any of the calibration functions. It is possible, however, that there could be some circumstance where the initial calibration settings where no longer valid due to some system event, so the calibration functions are exported for your convenience should it be useful sometime to run them explicitly.

The **hrt_cpu_process** and **hrt_cpu_thread** functions return the accumlated CPU time for the host process or thread when running under NT. Return units may be microsecond, milliseconds, or seconds. These functions return 0 when called from Win16 or Win95.

The **hrt_isNT** function is used to detect whether a 32 bit ExacTicks app is running on NT (as opposed to Win95). This allows the developer to know at run-time whether the process and thread specific timestamp, timer, and timing functions (including the **hrt_cpu_xxx** functions described above) are available.

The **hrt_report** function executes the timer and DLL status report. Each timer that is assigned a name when allocated will appear in the report, which looks something like this:

```
      ExacTicks 1.1 © 1996,1997 Ryle Design  www.ryledesign.com

                              ExacTicks Win16 Test

     Timer Name       Activations   Elapsed (secs)    Average (secs)
-------------------  -----------  ----------------  ----------------
Square Root Timer          10000          0.008313          0.000001
Disk IO Timer               1000          1.123000          0.001230

ExacTicks report complete xxx xxx xx 14:29:58 1997
```

The timer report may also have the complete ExacTicks DLL resource and calibration status appended to it, which would make the above report look like this:

```
      ExacTicks 1.1 © 1996,1997 Ryle Design  www.ryledesign.com

                              ExacTicks Win16 Test

     Timer Name       Activations   Elapsed (secs)    Average (secs)
-------------------  -----------  ----------------  ----------------
Square Root Timer          10000          0.008313          0.000001
Disk IO Timer               1000          1.123000          0.001230

                          WINHRT16 DLL Resource Summary

DLL version 1.10-16 loaded xxx xxx xx 14:29:55 1997
Timer frequency : 1193180.00 Hz
Timer overhead  : 11 microseconds
Delay atom      : 5029 nanoseconds
Delay minimum   : 7 microseconds
Delay switch    : 250 microseconds
Alarm overhead  : 28 microseconds
Event resolution: 1 milliseconds
Timestamps used : 4  available: 507
Timers used     : 1  available: 126
Alarms used     : 1  available: 126
Delays used     : 1  available: 126
Events used     : 1  available: 30

ExacTicks report complete xxx xxx xx 14:29:58 1997
```

The additional details are useful for diagnostic purposes, and if you are having problems with ExacTicks we will ask you to fax or email us a timer report with the DLL status information. The ExacTicks timer report may be sent to a printer, the “console”, or a disk file, and a title may be assigned that will appear on the report. The “console” destination causes the ExacTicks DLL to send the report out to a temporary file and then fire off the Windows notepad to open it for user viewing - at that time the user may choose to edit, save, and print the report file.

The **hrt_supported** function returns HRT_TRUE if ExacTicks is supported by the host Windows system, or HRT_FALSE if not. This function will return HRT_FALSE if a 32 bit app using ExacTicks is executed under Win32s on a Windows 3.1 or WFWG system, or if a 16 bit app using ExacTicks is executed on NT.

The **hrt_timestring** function accepts a floating point time value in units of seconds and returns a string in the format h:mm:ss.xxxxxx. This is useful when you wish to display timer results in a standard format regardless of the timer resolution. For example: passing 1234.567000 seconds returns a string of **0:20:34.567000**.

ExacTicks Reference

An alphabetical listing of all ExacTicks functions and procedures

hrt_alarm_alloc

C/C++ short hrt_alarm_alloc(short p1)

Delphi function hrt_alarm_alloc(p1 : word) : word

VB Function hrt_alarm_alloc (ByVal p1 As Integer) As Integer

Arguments p1 - resolution constant for alarm resoution desired: HRT_MICROSECOND, HRT_MILLISECOND, or HRT_SECOND

Returns Alarm handle (value > 0) if alarm is available, 0 if no alarm available

Comments Must be called before alarm may be used. Free allocated alarm when done with **hrt_alarm_free**.

hrt_alarm_avail

C/C++ short hrt_alarm_avail(void)

Delphi function hrt_alarm_avail : word

VB Function hrt_alarm_avail () As Integer

Arguments No arguments

Returns Number of alarms available for use. Under Win16 this returns the total number of alarms available for all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of alarms available for the calling application

Comments None

hrt_alarm_cancel

C/C++ void hrt_alarm_cancel(short p1)

Delphi procedure hrt_alarm_cancel(p1 : word)

VB Sub hrt_alarm_cancel (ByVal p1 As Integer)

Arguments p1 - alarm handle to cancel

Returns No return value

Comments Sets the alarm's valid bit to false. Invalid alarms always return 1 when checked.

hrt_alarm_check

C/C++ DWORD hrt_alarm_check(short p1)

Delphi function hrt_alarm_check(p1 : word) : LongInt

VB Function hrt_alarm_check (ByVal p1 As Integer) As Long

Arguments p1 - alarm handle to check

Returns 0 if alarm has not expired, otherwise the total amount of time elapsed since the alarm was set.

Comments If the alarm was set with the **HRT_RESET** flag, the alarm will reset when it is checked and expired, otherwise the alarm's valid bit is set to false.

hrt_alarm_free

C/C++ void hrt_alarm_free(short p1)

Delphi procedure hrt_alarm_free(p1 : word)

VB Sub hrt_alarm_free (ByVal p1 As Integer)

Arguments p1 - handle to alarm to free

Returns No return value

Comments An alarm allocated with **hrt_alarm_alloc** should be freed with this function when it is no longer being used.

hrt_alarm_getohead

C/C++ DWORD hrt_alarm_getohead(void)

Delphi function hrt_alarm_getohead : LongInt

VB Function hrt_alarm_getohead () As Long

Arguments No arguments

Returns Alarm overhead used to set microsecond resolution alarms. This value is subtracted from the microsecond alarm when set, so that the overhead of the alarm mechanism does not make the alarm take longer than specified.

Comments Of academic interest only. This value is set when the alarm engine is calibrated. This value may be overridden (not recommended) by using **hrt_alarm_setohead**.

hrt_alarm_inuse

C/C++ short hrt_alarm_inuse(void)

Delphi function hrt_alarm_inuse : word

VB Function hrt_alarm_inuse () As Integer

Arguments No Arguments

Returns Number of alarms in use. Under Win16 this returns the total number of alarms in use by all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of alarms in use by the calling application.

Comments None

hrt_alarm_reset

C/C++ void hrt_alarm_reset(short p1)

Delphi procedure hrt_alarm_reset(p1 : word)

VB Sub hrt_alarm_reset (ByVal p1 As Integer)

Arguments p1 - alarm handle to reset

Returns No return value

Comments Resets the alarm and begins counting down.

hrt_alarm_set

C/C++ void hrt_alarm_set(short p1, DWORD p2, short p3)

Delphi procedure hrt_alarm_set(p1 : word; p2 : LongInt; p3 : word)

VB Sub hrt_alarm_set (ByVal p1 As Integer, ByVal p2 As Long, ByVal p3 As Integer)

Arguments p1 - handle to alarm to set
p2 - length of alarm in alarm resolution units (resolution specified when alarm was allocated)
p3 - alarm reset action: HRT_ONCE alarm does not reset when checked and expired, HRT_RESET alarm resets when checked and expired

Returns No return value

Comments None

hrt_alarm_setohead

C/C++ void hrt_alarm_setohead(DWORD p1)

Delphi procedure hrt_alarm_setohead(p1 : LongInt)

VB Sub hrt_alarm_setohead (ByVal p1 As Long)

Arguments p1 - value to use for microsecond alarm overhead

Returns No return value

Comments This value is set automatically by the alarm calibration routine and there should be no need to override it. You can retrieve the current value by using **hrt_alarm_getohead**.

hrt_calibrate_alarm

C/C++ short hrt_calibrate_alarm(void)

Delphi function hrt_calibrate_alarm : word

VB Function hrt_calibrate_alarm () As Integer

Arguments None

Returns HRT_TRUE if calibration successful, HRT_FALSE if not.

Comments There is no need to explicitly call this function - this function is called automatically the first time an alarm is allocated.

hrt_calibrate_delay

C/C++ short hrt_calibrate_delay(void)

Delphi function hrt_calibrate_delay : word

VB Function hrt_calibrate_delay () As Integer

Arguments None

Returns HRT_TRUE if calibration successful, HRT_FALSE if not.

Comments There is no need to explicitly call this function - this function is called automatically the first time a delay is allocated.

hrt_calibrate_event

C/C++ short hrt_calibrate_event(void)

Delphi function hrt_calibrate_event : word

VB Function hrt_calibrate_event () As Integer

Arguments None

Returns HRT_ TRUE if calibration successful, HRT_FALSE if not.

Comme There is no need to explicitly call this function - this function is called

nts automatically the first time an event is allocated.

hrt_calibrate_timer

C/C++ short hrt_calibrate_timer(void)

Delphi function hrt_calibrate_timer : word

VB Function hrt_calibrate_timer () As Integer

Arguments None

Returns HRT_TRUE is calibration successful HRT_FALSE if not

Comments There is no need to explicitly call this function - this function is called automatically the first time a timer is allocated.

hrt_cpu_process

C/C++ DWORD hrt_cpu_process(short p1)

Delphi function hrt_cpu_process(p1 : word) : LongInt

VB Function hrt_cpu_process (ByVal p1 As Integer) As Long

Arguments p1 - resolution constant for return value desired: HRT_MICROSECOND, HRT_MILLISECOND, or HRT_SECOND

Returns Accumulated CPU time for the host process if a 32 bit app on NT, 0 otherwise.

Comments Check for NT at run-time using **hrt_isNT**

hrt_cpu_thread

C/C++ DWORD hrt_cpu_thread(short p1)

Delphi function hrt_cpu_thread(p1 : word) : LongInt

VB Function hrt_cpu_thread (ByVal p1 As Integer) As Long

Arguments p1 - resolution constant for return value desired: HRT_MICROSECOND, HRT_MILLISECOND, or HRT_SECOND

Returns Accumulated CPU time for the host thread if a 32 bit app on NT, 0 otherwise.

Comments Check for NT at run-time using **hrt_isNT**

hrt_delay_alloc

C/C++ short hrt_delay_alloc(DWORD p1, short p2)

Delphi function hrt_delay_alloc(p1 : LongInt; p2 : word) : word

VB Function hrt_delay_alloc (ByVal p1 As Long, ByVal p2 As Integer) As Integer

Arguments p1 - length of delay in delay resolution units
p2 - delay resolution: HRT_MICROSECOND, HRT_MILLISECOND, or HRT_SECOND

Returns Delay handle (value > 0) or 0 if no delays available.

Comments Remember to free the delay when no longer needed using **hrt_delay_free**.

hrt_delay_avail

C/C++ short hrt_delay_avail(void)

Delphi function hrt_delay_avail : word

VB Function hrt_delay_avail () As Integer

Arguments No arguments

Returns Number of delays available for use. Under Win16 this returns the total number of delays available for all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of delays available for the calling application

Comments None

hrt_delay_do

C/C++ void hrt_delay_do(short p1)

Delphi procedure hrt_delay_do(p1 : word)

VB Sub hrt_delay_do (ByVal p1 As Integer)

Arguments p1 - handle to delay to execute

Returns No return value

Comments Executes the delay allocated and set by **hrt_delay_alloc**.

hrt_delay_free

C/C++ void hrt_delay_free(short p1)

Delphi procedure hrt_delay_free(p1 : word)

VB Sub hrt_delay_free (ByVal p1 As Integer)

Arguments p1 - handle to delay to free

Returns No return value

Comments Always free a delay no longer needed.

hrt_delay_getatom

C/C++ DWORD hrt_delay_getatom(void)

Delphi function hrt_delay_getatom : LongInt

VB Function hrt_delay_getatom () As Long

Arguments No arguments

Returns Fundamental microsecond resolution delay unit. The unit of measure is nanoseconds. This value is set by the delay engine calibration function.

Comments Of academic or diagnostic interest only.

hrt_delay_getmin

C/C++ DWORD hrt_delay_getmin(void)

Delphi function hrt_delay_getmin : LongInt

VB Function hrt_delay_getmin () As Long

Arguments No arguments

Returns Minimum microsecond delay possible. This value is set by the delay engine calibration function.

Comments This will be the shortest duration microsecond delay possible in the current run-time environment.

hrt_delay_getswitch

C/C++ DWORD hrt_delay_getswitch(void)

Delphi function hrt_delay_getswitch : LongInt

VB Function hrt_delay_getswitch () As Long

Arguments No arguments

Returns Delay length in microseconds when the delay engine changes from running calibrated loops to an internal alarm to generate the requested delay. This value is initialized to 250 microseconds.

Comments Short duration microsecond delays must be generated by using calibrated loops. This technique tends to induce greater errors as the delay gets longer, so the delay engine switches to using an internal alarm to generate the delay (which improves accuracy) at a certain delay length threshold. This threshold value may be retrieved with this function and set with the **hrt_delay_setswitch** function.

hrt_delay_inuse

C/C++ short hrt_delay_inuse(void)

Delphi function hrt_delay_inuse : word

VB Functin hrt_delay_inuse () As Integer

Arguments No arguments

Returns Number of delays in use. Under Win16 this returns the total number of delays in use by all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of delays in use by the calling application

Comments None

hrt_delay_setswitch

C/C++ void hrt_delay_setswitch(DWORD p1)

Delphi procedure hrt_delay_setswitch(p1 : LongInt)

VB Sub hrt_delay_setswitch (ByVal p1 As Long)

Arguments p1 - new threshold for microsecond delays to run as internal alarms rather than calibrated loops.

Returns No return value.

Comments See **hrt_delay_getswitch**.

hrt_event_alloc

C/C++ short hrt_event_alloc(short p1, short p2)

Delphi function hrt_event_alloc(p1, p2 : word) : word

VB Function hrt_event_alloc (ByVal p1 As Integer, ByVal p2 As Integer) As Integer

Arguments p1 - event type - HRT_ONESHOT for an event to occur once only, HRT_PERIODIC for an event to occur repeatedly until explicitly canceled.
p2 - event resolution - HRT_MILLISECOND for an event scheduled in units of one millisecond, HRT_SECOND for an event scheduled in units on one second, or HRT_WALLCLOCK for an event scheduled to occur at a specified time of day.

Returns Handle to allocated event.

Comments Events are very scarce resources under all Windows platforms. This function allocates a data structure in the ExacTicks DLL but does not actually allocate a Windows multimedia callback timer (which is what generates the event). A successful allocation here does not guarantee that a callback timer will be available when the event is actually scheduled.

hrt_event_avail

C/C++ short hrt_event_avail(void)

Delphi function hrt_event_avail : word

VB Function hrt_event_avail () As Integer

Arguments No arguments

Returns Number of events available for use. Under Win16 this returns the total number of events available for all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of events available for the calling application

Comments This value is the number of event data structures available in the ExacTicks DLL and not the number of Windows multimedia callback timers available for use.

hrt_event_cancel

C/C++ void hrt_event_cancel(short p1)

Delphi procedure hrt_event_cancel(p1 : word)

VB Sub hrt_event_cancel (ByVal p1 As Integer)

Arguments p1 - handle to event to cancel.

Returns No return value.

Comments If the event was pending it will be canceled and the Windows multimedia callback timer associated with the pending event will be freed.

hrt_event_clockcallback

C/C++ short hrt_event_clockcallback(short p1, short p2, short p3, short p4, FARPROC p5, DWORD p6)

Delphi function hrt_event_clockcallback(p1, p2, p3, p4 : word; p5 : hrt_event; p6 : LongInt) : word

VB Function hrt_event_clockcallback (ByVal p1 As Integer, ByVal p2 As Integer, ByVal p3 As Integer, ByVal p4 As Integer, ByVal p5 As Long, ByVal p6 As Long) As Integer

Arguments
p1 - handle to event to schedule
p2 - hour component (0-23) of time for event to occur
p3 - minute component (0-59) of time for event to occur
p4 - second component (0-59) of time for event to occur
p5 - address of function to invoke when event occurs
p6 - user defined data to be passed to the event function when invoked.

Returns HRT_TRUE if event was scheduled, HRT_FALSE if event could not be scheduled. Failure of this function is caused by the unavailability of a Windows multimedia callback timer used to schedule the event.

Comments See the ExacTicks Technical Overview section for details on the availability of multimedia callback timers. Under Win16 there are significant restrictions on what the event function can do - see the ExacTicks Technical Overview and the ExacTicks Cookbook sections for more details.

VB: At this time Visual Basic does not support passing a function address, so **this function is not supported by current versions of Visual Basic.**

hrt_event_clockmsg

C/C++ short hrt_event_clockmsg(short p1, short p2, short p3, short p4, HWND p5, UINT p6, DWORD p7)

Delphi function hrt_event_clockmsg(p1, p2, p3, p4 : word; p5 : THandle; p6 : Integer; p7 : LongInt) : word

VB **(16 bit)** Function hrt_event_clockmsg (ByVal p1 As Integer, ByVal p2 As Integer, ByVal p3 As Integer, ByVal p4 As Integer, ByVal p5 As Integer, ByVal p6 As Integer, ByVal p7 As Long)

(32 bit) Function hrt_event_clockmsg (ByVal p1 As Integer, ByVal p2 As Integer, ByVal p3 As Integer, ByVal p4 As Integer, ByVal p5 As Long, ByVal p6 As Long, ByVal p7 As Long)

Arguments
p1 - handle to event to schedule
p2 - hour component (0-23) of time for event to occur
p3 - minute component (0-59) of time for event to occur
p4 - second component (0-59) of time for event to occur
p5 - window handle to post message to
p6 - message to post to window
p7 - user defined data to be passed along with message to window

Returns HRT_TRUE if event was scheduled, HRT_FALSE if event could not be scheduled. Failure of this function is caused by the unavailability of a Windows multimedia callback timer used to schedule the event.

Comments See the ExacTicks Technical Overview section for details on the availability of multimedia callback timers.

VB: At this time Visual Basic does not support explicit message processing by the application developer. A public domain VBX is provided to implement this capability, but there is no 32 bit OCX version available in the public domain, so this function is not supported by 32 bit VB. Check the README.TXT file for any updates on the status of this problem.

hrt_event_free

C/C++ void hrt_event_free(short p1)

Delphi procedure hrt_event_free(p1 : word)

VB Sub hrt_event_free (ByVal p1 As Integer)

Arguments p1 - handle to event to free

Returns No return value

Comments If the event is pending, it is canceled. Always free an allocated event when it is no longer needed.

hrt_event_inuse

C/C++ short hrt_event_inuse(void)

Delphi function hrt_event_inuse : word

VB Function hrt_event_inuse () As Integer

Arguments No arguments

Returns Number of events in use. Under Win16 this returns the total number of events in use by all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of events in use by the calling application

Comments This value is the number of event data structures in use by the ExacTicks DLL and not the number of Windows multimedia callback timers in use by the host Windows system..

hrt_event_maxperiod

C/C++ short hrt_event_maxperiod(void)

Delphi function hrt_event_maxperiod : word

VB Function hrt_event_maxperiod () As Integer

Arguments No arguments

Returns Maximum length of millisecond resolution event period. On most systems this value will be 65535.

Comments None.

hrt_event_minperiod

C/C++ short hrt_event_minperiod(void)

Delphi function hrt_event_minperiod : word

VB Function hrt_event_minperiod () As Integer

Arguments No arguments

Returns Minimum length of millisecond resolution event period. On most systems this value will be 1.

Comments None

hrt_event_pending

C/C++ short hrt_event_pending(short p1)

Delphi function hrt_event_pending(word : p1)

VB Function hrt_event_pending (ByVal p1 As Integer) As Integer

Arguments p1 - handle to event to check pending status

Returns HRT_TRUE if the event is pending (i.e. scheduled and waiting to be delivered), HRT_FALSE if the event is not pending.

Comments None

hrt_event_reset

C/C++ short hrt_event_reset(short p1)

Delphi function hrt_event_reset(p1 : word) : word

VB Function hrt_event_reset (ByVal p1 As Integer) As Integer

Arguments p1 - handle to event to reset

Returns HRT_TRUE if successful, HRT_FALSE if failure. Failure of this function is caused by the unavailability of a Windows multimedia callback timer used to schedule the event.

Comments The event reset is canceled if pending and rescheduled using the values originally used to set it..

hrt_event_setcallback

C/C++ short hrt_event_setcallback(short p1, DWORD p2, FARPROC p3, DWORD p4)

Delphi function hrt_event_setcallback(p1 : word; p2 : LongInt; p3 : hrt_event; p4 : LongInt) : word

VB Function hrt_event_setcallback (ByVal p1 As Integer, ByVal p2 As Long, ByVal p3 As Long, ByVal p4 As Long) As Integer

Arguments p1 - handle to event to schedule
p2 - length of time for event to occur.
p3 - address of function to invoke when event occurs
p4 - user defined data passed to the event function

Returns HRT_TRUE if event was scheduled, HRT_FALSE if event could not be scheduled. Failure of this function is caused by the unavailability of a Windows multimedia callback timer used to schedule the event.

Comments See the ExacTicks Technical Overview section for details on the availability of multimedia callback timers. Under Win16 there are significant restrictions on what the event function can do - see the ExacTicks Technical Overview and the ExacTicks Cookbook sections for more details. If the event has HRT_MILLISECOND resolution, p2 should not exceed the value returned by **hrt_event_maxperiod**, which is usually 65535 milliseconds on most systems.

VB: At this time Visual Basic does not support passing a function address, so **this function is not supported by current versions of Visual Basic.**

hrt_event_setmessage

C/C++ short hrt_event_setmessage(short p1, DWORD p2, HWND p3, UINT p4, DWORD p5)

Delphi function hrt_event_setmessage(p1 : word; p2 : LongInt; p3 : THandle; p4 : Integer; p5 : LongInt) : word

VB **(16 bit)** Function hrt_event_setmessage (ByVal p1 As Integer, ByVal p2 As Long, ByVal p3 As Integer, ByVal P4 As Integer, ByVal p5 As Long) As Integer

(32 bit) Function hrt_event_setmessage (ByVal p1 As Integer, ByVal p2 As Long, ByVal p3 As Long, ByVal p4 As Long, ByVal p5 As Long) As Integer

Arguments p1 - handle to event to schedule
p2 - length of time for event to occur
p3 - handle to window to post message
p4 - message to post to window
p5 - user defined data to post with message

Returns HRT_TRUE if event was scheduled, HRT_FALSE if event could not be scheduled. Failure of this function is caused by the unavailability of a Windows multimedia callback timer used to schedule the event.

Comments See the ExacTicks Technical Overview section for details on the availability of multimedia callback timers. If the event has HRT_MILLISECOND resolution, p2 should not exceed the value returned by **hrt_event_maxperiod**, which is usually 65535 milliseconds on most systems

VB: At this time Visual Basic does not support explicit message processing by the application developer. A public domain VBX is provided to implement this capability, but there is no 32 bit OCX version available in the public domain, so this function is not supported by 32 bit VB. Check the README.TXT file for any updates on the status of this problem.

hrt_isNT

C/C++ short hrt_isNT(void)

Delphi function hrt_isNT: word

VB Function hrt_alarm_alloc As Integer

Arguments None

Returns HRT_TRUE if the host application is running as a 32 bit app under NT
HRT_FALSE if the host application is running as a 32 bit app under Win95 or as 16 bit app on anything.

Comments Use to detect at run-time if the process and thread specific CPU timing functions are available.

hrt_report

C/C++ short hrt_report(LPSTR p1, short p2, LPSTR p3)

Delphi function hrt_report(title : PChar; p2 : word; p3 : PChar) : word

VB Function hrt_report (ByVal p1 As String, ByVal p2 As Integer, ByVal p3 As String) As Integer

Arguments p1 - title of report. If an empty string is passed, the report title will be “ExacTicks Timer Summary”
p2 - destination of report. HRT_CONSOLE opens the report in Windows Notepad

for viewing, printing, or saving. HRT_PRINTER prints the report on the default print device. HRT_DISKFILE outputs the report to the diskfile specified in p3. You may logically OR the constant HRT_DLLSTATUS to any of these constants to append the ExacTicks DLL status summary to the bottom of the report.
p3 - complete path and name for output file if p2 uses HRT_DISKFILE constant.

Returns HRT_TRUE if success, HRT_FALSE if report could not be generated.

Comments Logically ORing the constant HRT_DLLSTATUS to parameter p2 gives complete diagnostic information on the ExacTicks DLL resource and calibration status. A timer must have a name associated with it to appear in the timer report. Under Win16 all named timers owned by all applications that reference the ExacTicks DLL will be listed. Under Win32 only the named timers owned by the calling application will be listed.

hrt_supported

C/C++ short hrt_supported(void)

Delphi function hrt_supported : word

VB Function hrt_supported () As Integer

Arguments No arguments

Returns Returns HRT_TRUE if ExacTicks is supported by the host Windows platform. Returns HRT_FALSE if ExacTicks is not supported by the host Windows platform.

Comments ExacTicks is not supported under Win32s.

hrt_timer_alloc

C/C++ short hrt_timer_alloc(short p1, LPSTR p2)

Delphi function hrt_timer_alloc(p1 : word; p2 : PChar) : word

VB Function hrt_timer_alloc (ByVal p1 As Integer, ByVal p2 As String) As Integer

Arguments p1 - resolution of timer: HRT_MICROSECOND, HRT_MILLISECOND, HRT_SECOND. On NT you may logically OR one of these values with HRT_PROCESS or HRT_THREAD, to measure the CPU time of the host process or thread respectively. Using the HRT_PROCESS or HRT_THREAD flag under Win16 or Win95 is ignored.
p2 - optional name for timer. Named timers are listed in the ExacTicks timer report. If you do not wish the timer to appear in the timer report, pass an empty string for p2.

Returns Handle to the timer (value > 0) if successful, 0 if no timers available

Comments Use **hrt_timer_free** to free allocated timers when no longer needed.

hrt_timer_avail

C/C++ short hrt_timer_avail(void)

Delphi function hrt_timer_avail : word

VB Functin hrt_timer_avail () As Integer

Arguments No arguments

Returns Number of timers available for use. Under Win16 this returns the total number of timers available for all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of timers available for the calling application

Comments None

hrt_timer_count

C/C++ DWORD hrt_timer_count(short p1)

Delphi function hrt_timer_count(p1 : word) : LongInt

VB Function hrt_timer_count (ByVal p1 As Integer) As Long

Arguments p1 - handle of timer to query

Returns Number of activation counts for the specified timer. The timer's activation count is incremented when the **hrt_timer_start** function is called for the given timer.

Comments None

hrt_timer_current

C/C++ DWORD hrt_timer_current(short p1)

Delphi function hrt_timer_current(p1 : word) : LongInt

VB Function hrt_timer_current (ByVal p1 As Integer) As Long

Arguments p1 - handle of timer to query

Returns Amount of time in timer's resolution units since timer was lasted started with **hrt_timer_start**.

Comments If timer is not running when queried, returns 0.

hrt_timer_currentsecs

C/C++ double hrt_timer_currentsecs(short pt1)

Delphi function hrt_timer_currentsecs(p1 : word) : double

VB Functin hrt_timer_currentsecs (ByVal p1 As Integer) As Double

Arguments p1 - handle of timer to query

Returns Amount of time in seconds since timer was last started with **hrt_timer_start**

Comments If timer is not running when queried, returns 0.

hrt_timer_elapsed

C/C++ DWORD hrt_timer_elapsed(short p1)

Delphi function hrt_timer_elapsed(p1 : word) : LongInt

VB Function hrt_timer_elapsed (ByVal p1 As Integer) As Long

Arguments p1 - handle of timer to query

Returns Elapsed time accumulated by timer in units of timer resolution.

Comments Elapsed time is added to the timer's accumulated total when **hrt_timer_stop** is invoked. If the timer is running when this function is called, only the elapsed time up to the last **hrt_timer_stop** call will be returned.

hrt_timer_elapsedsecs

C/C++ double hrt_timer_elapsedsecs(short p1)

Delphi function hrt_timer_elapsedsecs(p1 : word) : double

VB Function hrt_timer_elapsedsecs (ByVal p1 As Integer) As Double

Arguments p1 - handle of timer to query

Returns Elapsed time accumulated by timer in units of seconds.

Comments Elapsed time is added to the timer's accumulated total when **hrt_timer_stop** is invoked. If the timer is running when this function is called, only the elapsed time up to the last **hrt_timer_stop** call will be returned.

hrt_timer_free

C/C++ void hrt_timer_free(short p1)

Delphi procedure hrt_timer_free(p1 : word)

VB Sub hrt_timer_free (ByVal p1 As Integer)

Arguments p1 - handle of timer to free

Returns No return value

Comments Always free allocated timers.

hrt_timer_getname

C/C++ LPSTR hrt_timer_getname(short p1)

Delphi function hrt_timer_getname(p1 : word) : PChar

VB Function hrt_timer_getname (ByVal p1 As Integer) As String

Arguments p1 - handle of timer to query

Returns Name of timer (max 20 characters) if timer was named when allocated.

Comments None

hrt_timer_getohead

C/C++ DWORD hrt_timer_getohead(void)

Delphi function hrt_timer_getohead : LongInt

VB Function hrt_timer_getohead () As Long

Arguments No arguments

Returns Overhead of **hrt_timer_start** and **hrt_timer_stop** sequence, which is subtracted from time accumulated by timer.

Comments This value is calculated automatically by the timer calibration routine when the first timer is allocated.

hrt_timer_getresolution

C/C++ short hrt_timer_getresolution(short p1)

Delphi function hrt_timer_getresolution(p1 : word) : word

VB Function hrt_timer_getresolution (ByVal p1 As Integer) As Integer

Arguments p1 - handle of timer to query

Returns Resolution of timer (set when allocated by **hrt_timer_alloc**). The value will be HRT_MICROSECOND, HRT_MILLISECOND, or HRT_SECOND.

Comments None

hrt_timer_getstate

C/C++ short hrt_timer_getstate(short p1)

Delphi function hrt_timer_getstate(p1 : word) : word

VB Function hrt_timer_getstate (ByVal p1 As Integer) As Integer

Arguments p1 - handle of timer to query

Returns Timer state: either HRT_RUN, HRT_STOP, or HRT_SUSPEND.

Comments None

hrt_timer_getstatus

C/C++ short hrt_timer_getstatus(short p1)

Delphi function hrt_timer_getstatus(p1 : word) : word

VB Function hrt_timer_getstatus (ByVal p1 As Integer) As Integer

Arguments p1 - handle of timer to query

Returns Timer error status
HRT_OK - no errors
HRT_UNDERRUN - **hrt_timer_start** or **hrt_timer_resume** was invoked on a timer not stopped or suspended
HRT_OVERRUN - **hrt_timer_stop** or **hrt_timer_suspend** was invoked on a timer not running
HRT_OVERFLOW - the timer's accumlulated elapsed time has overflowed

Comments When queried with this function the timer's status is reset to HRT_OK..

hrt_timer_inuse

C/C++ short hrt_timer_inuse(void)

Delphi function hrt_timer_inuse : word

VB Function hrt_timer_inuse () As Integer

Arguments No arguments

Returns Number of timers in use. Under Win16 this returns the total number of timers in use by all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of timers in use by the calling application

Comments None

hrt_timer_lastelapsed

C/C++ DWORD hrt_timer_lastelapsed(short p1)

Delphi function hrt_timer_lastelapsed(p1 : word) : LongInt

VB Function hrt_timer_lastelapsed (ByVal p1 As Integer) As Long

Arguments p1 - handle of timer to query

Returns Elapsed time of last **hrt_timer_start** / **hrt_timer_stop** sequence in units of the timer's resolution.

Comments None

hrt_timer_lastelapsedsecs

C/C++ double hrt_timer_lastelapsedsecs(short p1)

Delphi function hrt_timer_lastelapsedsecs(p1 : word) : double

VB Function hrt_timer_lastelapsedsecs (ByVal p1 As Integer) As Double

Arguments p1 - handle of timer to query

Returns Elapsed time of last **hrt_timer_start** / **hrt_timer_stop** sequence in seconds

Comments None

hrt_timer_reset

C/C++ void hrt_timer_reset(short p1)

Delphi procedure hrt_timer_reset(p1 : word)

VB Sub hrt_timer_reset (ByVal p1 As Integer)

Arguments p1 - handle of timer to reset

Returns No return value

Comments A timer reset zeros the timer's accumulated elapsed time and activation count.

hrt_timer_resume

C/C++ void hrt_timer_resume(short p1)

Delphi procedure hrt_timer_resume(p1 : word)

VB Sub hrt_timer_resume (ByVal p1 As Integer)

Arguments p1 - handle of timer to resume

Returns No return value

Comments The **hrt_timer_suspend** / **hrt_timer_resume** pair of functions are useful if you need to stop and start a timer without incrementing the timer's activation count.

hrt_timer_setname

C/C++ void hrt_timer_setname(short p1, LPSTR p2

Delphi procedure hrt_timer_setname(p1 : word; p2 : PChar)

VB Sub hrt_timer_setname (ByVal p1 As Integer, ByVal p2 As String)

Arguments p1 - handle of timer to reference
p2 - character string (max 20 characters) to assign to timer as the timer's name, which shows up in the timer report.

Returns No return value.

Comments A timer may be assigned a name using this function or when the timer is allocated usign **hrt_timer_alloc.**

hrt_timer_setohead

C/C++ void hrt_timer_setohead(DWORD p1)

Delphi procedure hrt_timer_setohead(p1 : LongInt)

VB Sub hrt_timer_setohead (ByVal p1 As Long)

Arguments p1 - new timer overhead value in units of microseconds

Returns No return value

Comments The timer overhead value is calculated by the timer calibration function when the first timer is allocated. This function allows that automatic calculation to be overridden.

hrt_timer_start

C/C++ void hrt_timer_start(short p1)

Delphi procedure hrt_timer_start(p1 : word)

VB Sub hrt_timer_start (ByVal p1 As Integer)

Arguments p1 - handle of timer to start

Returns No return value

Comments Starts the timer running and increments the timer's activation count.

hrt_timer_stop

C/C++ void hrt_timer_stop(short p1)

Delphi procedure hrt_timer_stop(p1 : word)

VB Sub hrt_timer_stop (ByVal p1 As Integer)

Arguments p1 - handle of timer to stop

Returns No return value

Comments Stops a running timer. The timer's accumulated elapsed time is updated.

hrt_timer_suspend

C/C++ void hrt_timer_suspend(short p1)

Delphi procedure hrt_timer_suspend(p1 : word)

VB Sub hrt_timer_suspend (ByVal p1 As Integer)

Arguments p1 - handle of timer to suspend

Returns No return value.

Comments The **hrt_timer_suspend** / **hrt_timer_resume** pair of functions are useful if you need to stop and start a timer without incrementing the timer's activation count.

hrt_timestamp_alloc

C/C++ short hrt_timestamp_alloc(short p1)

Delphi function hrt_timestamp_alloc(p1 : word) : word

VB Function hrt_timestamp_alloc (ByVal p1 As Integer) As Integer

Arguments p1 - HRT_WALLCLOCK to obtain a timestamp to measure "wallclock" elapsed time. HRT_PROCESS or HRT_THREAD may be passed under NT to measure process or thread CPU time respectively. If HRT_PROCESS or HRT_THREAD are used when not running 32 bits on NT, these flags are ignored and HRT_WALLCLOCK is assumed.

Returns A handle to a timestamp (value > 0) if successful, or 0 if no timestamps are available

Comments Always free allocated timestamps with **hrt_timestamp_alloc**

hrt_timestamp_avail

C/C++ short hrt_timestamp_avail(void)

Delphi function hrt_timestamp_avail : word

VB Function hrt_timestamp_avail () As Integer

Arguments No arguments

Returns Number of timestamps available for use. Under Win16 this returns the total number of timestamps available for all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of timestamps

available for the calling application

Comments None

hrt_timestamp_free

C/C++ void hrt_timestamp_free(short p1)

Delphi procedure hrt_timestamp_free(p1 : word)

VB Sub hrt_timestamp_free (ByVal p1 As Integer)

Arguments p1 - handle of timestamp to free

Returns No return value

Comments Always free allocated timestamps when no longer in use.

hrt_timestamp_get

C/C++ void hrt_timestamp_get(short p1)

Delphi procedure hrt_timestamp_get(p1 : word)

VB Sub hrt_timestamp_get (ByVal p1 As Integer)

Arguments p1 - handle of timestamp to get

Returns No return value

Comments None

hrt_timestamp_inuse

C/C++ short hrt_timestamp_inuse(void)

Delphi function hrt_timestamp_inuse : word

VB Function hrt_timestamp_inuse () As Integer

Arguments No arguments

Returns Number of timestamps in use. Under Win16 this returns the total number of timestamps in use by all applications that reference WINHRT16.DLL. Under Win32 this returns the total number of timestamps in use by the calling application

Comments None

hrt_timestamp_msecdiff

C/C++ DWORD hrt_timestamp_msecdiff(short p1, short p2)

Delphi function hrt_timestamp_msecdiff(p1, p2 : word) : LongInt

VB Function hrt_timestamp_msecdiff (ByVal p1 As Integer, ByVal p2 As Integer) As Long

Arguments p1 - handle of first timestamp, the least recent or “start” value
p2 - handle of second timestamp, the most recent or “stop” value

Returns Difference between the two timestamps in milliseconds. Two error conditions are also possible and returned as constant values: HRT_UNDERFLOW if the values are passed in the wrong order, and HRT_OVERFLOW if the elapsed time is greater than can be held by the four byte unsigned integer.

Comments None

hrt_timestamp_secdiff

C/C++ DWORD hrt_timestamp_secdiff(short p1, short p2)

Delphi function hrt_timestamp_secdiff(p1, p2 : word) : LongInt

VB Function hrt_timestamp_secdiff (ByVal p1 As Integer, ByVal p2 As Integer) As Long

Arguments p1 - handle of first timestamp, the least recent or “start” value
p2 - handle of second timestamp, the most recent or “stop” value

Returns Difference between the two timestamps in seconds. Two error conditions are also possible and returned as constant values: HRT_UNDERFLOW if the values are passed in the wrong order, and HRT_OVERFLOW if the elapsed time is greater than can be held by the four byte unsigned integer

Comments None

hrt_timestamp_usecdiff

C/C++ DWORD hrt_timestamp_usecdiff(short p1, short p2)

Delphi function hrt_timestamp_usecdiff(p1, p2 : word) : LongInt

VB Function hrt_timestamp_usecdiff (ByVal p1 As Integer, ByVal p2 As Integer) As Long

Arguments p1 - handle of first timestamp, the least recent or “start” value
p2 - handle of second timestamp, the most recent or “stop” value

Returns Difference between the two timestamps in microseconds. Two error conditions are also possible and returned as constant values: HRT_UNDERFLOW if the values are passed in the wrong order, and HRT_OVERFLOW if the elapsed time is greater than can be held by the four byte unsigned integer

Comme None

nts

hrt_timestring

C/C++	LPSTR hrt_timestring(double p1, LPSTR p2)
Delphi	function hrt_timestring(p1 : double; p2 : PChar) : PChar
VB	Function hrt_timestring (ByVal p1 As Double, ByVal p2 As String) As String
Arguments	p1 - time in seconds p2 - pointer to string to receive formatted time.
Returns	string formatted as follows H:MM:SS.xxxxxxx
Comments	Allowing 20 characters for the string should handle all possible strings formatted.

Developer Notes

Tools used and comments from the developer

ExacTicks 1.1 was developed by Thomas Leathley using the following tools:

- Microsoft Visual C++ 4.2 and 1.5
- Borland C++ 4.5 and C++ Builder 1.0
- Watcom C++ 10.6
- Delphi 1.0 and 3.0
- Visual Basic 4.0
- Visual SlickEdit 2.0
- Microsoft Word For Windows 95
- Microsoft Windows 3.1 and WFWG
- Microsoft Windows 95 OSR2
- Microsoft Windows NT 4 with Service Pack 3
- The brain of W. Eric Wentz

About Ryle Design …

Ryle Design, located in Mt. Pleasant Michigan, was formed in 1987 to develop graphics and timing tools for the PC/MSDOS and Windows platforms. In the past we have specialized in precision timing tools for MSDOS, and drivers and tools for the Borland BGI DOS graphics library. ExacTicks is our second developer's toolkit for Windows. Custom development and consulting is available. Our web site is located at **www.ryledesign.com** and we can be contacted by email at **info@ryledesign.com**.