

Simplify and enhance data collection performance and analysis.”CodeTune Timing Analyzer” is a library aimed to help software developers improve the performance of applications, drivers and system software.

Designed to be as simple and fast as possible, CodeTune uses a series of statistical analysis on the inputted data in order to you know what is the actual speed of the code you are creating.

Whether you are tuning for the first time or doing advanced performance optimization, CodeTune provides a simple way of performance insight about the speed of your code.

Analysis is faster and easier because CodeTune understands the differences of the speed of the code you are testing, and perform a statistical analysis of the best speed that represents the one that is being generated by your code.

The generated data is easier to interpret. You can use its powerful analysis to sort, filter and visualize results on your source.

What Is Profiling?

Profiling is a way to measure where a program spends time. After you identify which functions are consuming the most time, you can evaluate them for possible performance improvements. Also, you can profile your code as a debugging tool. For example, determining which lines of code does not run can help you develop test cases that exercise that code. If you get an error in the file when profiling, you can see what ran and what did not to help you isolate the problem.

CodeTune library uses 8 (eight) different methods to calculate the correct timing of your code. On each one of them, it uses a statistical analysis of the events observed and if it finds the correct timing, it returns the values on a summary that is basically the Standard Deviation of the observed events.

The Standard deviation technique is used to compute the best time from where your code is actually running on your processor.

The results are evaluated, primarily, in Nanoseconds, from where you can convert later to clock cycles, megahertz, gigahertz, or other unit converters available upon this library.

Optimizing code



“Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered.” Donald Knuth.

Optimizing code to make it run faster is an iterative process:

1. Find the biggest bottleneck (the slowest part of your code).
2. Try to eliminate it (you may not succeed but that's ok).
3. Repeat until your code is “fast enough.”

This sounds easy, but it's not.

Even experienced programmers have a hard time identifying bottlenecks in their code. Instead of relying on your intuition, you should profile your code: use realistic inputs and measure the run-time of each individual operation. Only once you've identified the most important bottlenecks can you attempt to eliminate them.

Statistical profilers

CodeTune library is a statistical profiler. Therefore, it operates by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. The library works as accurate as possible trying to transpose the general knowledge that sampling profiles were typically less numerically accurate and specific.

The resulting data are not exact, but a very good statistical approximation with a margin of error of less of 1% in most cases.

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since CodeTune is a statistical profiler, it don't affect the execution speed as much, it can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of CPU cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

CodeTune General Overview

CodeTune library uses 8 (eight) different Algorithms to calculate the exact time your code is running. Each algorithm is described in this help file as a "Method".

The functionality of the library works in different stages, each one of them is interpreted internally to try to guarantee a better result.

The **1st stage** is described as below:



Once the method is chosen by the user and his code selected as the target to be analyzed, the algorithm performs a series of computations to calculate the difference between the user function started and when it ended.

This computation is done as many times the user configures the algorithm. Each one of these computations is called as "iteration" that is basically a loop of XXX times from where on each loop the difference is being added to the previous one. The total amount of time of the iterations (loops) is defined by the user.

CodeTune Reference Guide

Once the iterations ends, the collected data is then passed through a function that performs a Standard deviation computation of the differences found on each loop.

Accordingly to the results of the Standard Deviation, the resultant value can be considered a good value or not, as long as it matches certain criteria's designed to keep the resultant value the most accurate as possible. If, those criteria's don't match, the function returns and performs the iterations all over again. If the values are Ok, they are considered as a "good sample" to be analyzed later.

In general, whenever the values don't match the criteria it means that the algorithm found errors during the collecting, such as overheads, throughputs, pipelines problems and so on.

The **2nd stage** then, performs another analysis basically identical to the previous one. The difference is on the total amount of "good samples" to be collected.

On input, the user must choose how many samples he wants to be collected and how much iteration are necessary to be performed.

The more samples to be collected and the more iteration, more accurate the result is. The side effect of choosing too many samples or iterations is that it will take more time to the algo finishes.

A acceptable value of samples to be collected are 300 and the amount of iterations are acceptable to be 3000. Those values are enough to collect the necessary data to be analyzed.

So, the 2nd stage involves doing the 1st stage all over again XXX times as defined by the user how many samples he wants to collect to be analyzed.

Once the algorithm finishes his job, the resultant amount of "good samples" is then ready to be interpreted internally on the 3rd stage.

But, wait, there's more !

The 1st and 2nd stage described here shows the general rule of the functionality of the main Api.

But, what we are analyzing exactly ?

Well, we are analyzing (Benchmarking) the time the user's code completes his work. But...(Yes, there's a "but" in here), aren't we also analyzing the prologue and epilogue as long with the user code ?

Yes, in a matter of fact, we are! The user's code on input is located from inside another function. Internally, the function creates a placeholder to find the user's code.



CodeTune Reference Guide

So, we need to calculate 1st the values of good samples of the place holder without the user's code inside. Then, only after these computations, we can do the same for the user's code.

The advantage is that we can not only exclude from the result any errors generated by the accumulation of loops (iterations), but also generate more accurate values that represents the true timing your function is taking to work.

Why the errors are generated? Well, because there is a fundamental problem with deterministic profilers involving accuracy. The most obvious restriction is that the underlying "clock" is only ticking at a rate (typically) of about 100 nanoseconds. Hence no measurements will be more accurate than the underlying clock.

If enough measurements are taken, then the "error" will tend to average out. Unfortunately, removing this first error induces a second source of error.

The second problem is that it "takes a while" from when an event is dispatched until the profiler's call to get the time actually gets the state of the clock. Similarly, there is a certain lag when exiting the profiler event handler from the time that the clock's value was obtained (and then squirreled away), until the user's code is once again executing.

As a result, functions that are called many times, or call many functions, will typically accumulate this error. The error that accumulates in this fashion is typically less than the accuracy of the clock (less than one clock tick), but it can accumulate and become very significant.

The problem is more important with profile than with the lower-overhead. For this reason, CodeTune provides another approach while calibrating itself so that this error can be probabilistically (on the average) removed.

After the profiler is calibrated, it will be more accurate (in a least square sense).

Nevertheless, during the 1st and 2nd stages, CodeTune will eventually find negative numbers (when call counts are exceptionally low, and the gods of probability work against you :-). Those results are one of the issues that are solved by the criteria of consider a event as a "good sample" or not.

Don't worry, because all negative results are solved during all the 1st and 2nd stages during calibration or benchmarking the code itself.

The **3rd and final stage**

After all of those computations, the collected samples are now considered "good ones".

But, are they really good samples?

Well, now comes the final stage involving the internal interpretation of the collected data.

CodeTune Reference Guide

The analyses is done comparing the values found during the calibration and benchmarking to see if they fits the minimum requirement to effectively be considered a good sample.

It checks if the differences among each one of them are smaller then 1% (*on their own processes, i.e., the differences from the samples collected during calibration and the ones in the benchmarking*), then subtract the benchmark results with the calibration results and see if there are no negative values and a couple of others minor analysis.



If the resultant samples are good enough it selects the best candidates and perform a final Standard deviation computation on the remaining samples to retrieve the most accurate result as possible.

As a result, we have found and chosen some samples from where we can have a better estimation of the time your code actually is consuming.

The resultant data is displayed in nanoseconds. In general, a good interpretation of the Standard Deviation results indicates that, despites the value found on the Mean you can safely interpret the final estimation as the Minimum found on the Standard Deviation summary.

For example, if the summary of the resultant data using Algo Method n° 8 (*which uses rdtsc instruction as will be explained in the Reference topic of this document*) is:

Mean: 5.60425181229179 ns

Population Standard Deviation (Minimum): 4.95078946689592 ns

Population Standard Deviation (Maximum): 6.25771415768766 ns

Population Variance: 0.42701303685027 ns

Population Standard Deviation: 0.65346234539587 ns

Sample Standard Deviation (Minimum): **4.94968284062409 ns**

Sample Standard Deviation (Maximum): 6.25882078395948 ns

Sample Variance: 0.42846053867010 ns

Sample Standard Deviation: 0.65456897166769 ns

You can assume that your function is taking only 4.94968284062409 ns to work while it was being analyzed under Method 8.

What the Variance and Standard Deviation means for the scope of this Library?

The higher variance and Standard Deviation found means how much that the chosen Method influences the result. And, therefore, indicates if you function can be influenced by the code that surrounds it. I.e., if you function causes or is influenced by Stallings, stacks problems, overheads and so on. It is a indicative of the stability of your function.

CodeTune Reference Guide

It is important to also measure the values of the Variation and Standard Derivation to see if you will need to align your function or insert code to serialize it before and after it.

In general lines, you function will be extremely stable when:

- The variance and Standard deviations have a value near to zero or, at least less then 0.001
- The difference between the “Population” and “Sample Standard Deviation” for variance and Standard Deviation is also closer to Zero. For example, if variance from “Population Variance” is 0.01 and Sample Variance is 0.0099

But, these interpretations are mainly subjective. Some methods are worst then others to acquire the correct values of the timings. For example you may found that using Method 8 (rdtsc instruction) frequently generates high differences in the variance and standard deviation. This results alone does not means that you function is not stable. You need to evaluate the stability using the other methods individually.

If you test your function with the 8 methods and all of them have low differences and the only one that has a high Variance/STD is Mrthod 8 (or 1, 2, etc), then you shouldn't be worry about it because some instructions when working isolated does not produces accurate results anyway. CodeTune was specifically designed to overcome this particular problem, in order to it tries to keep the most accurate result as possible so you can be able to compare the result with other methods to better analyze how fast your function is.

The key for the interpretation here is analyzing your function under all available methods and see the differences of the results (Of variance, STD, Maximum and Minimum – from Popular and Sample) on each Summary. The less difference they have, more stable your function is.

According to the results you will see not only how fast your function is, but, how much it can be influenced by code surrounding it.



HAVE FUN !

CodeTune API Reference

The following is a list of the reference content for the CodeTune Application Programming interface (API).

Using the CodeTune API, you can develop benchmark applications that run successfully on all versions of Windows while taking advantage of the features and capabilities unique to each version.

Below is the list of available APIs on the current version. All functions used on this library are in stdcall format.

I - Functions

1 - Main functionality:

- [CreateTimeProfile](#)
- [CreateTimeProfileEx](#)

2 - Complementary functions:

- [RunTimeDataProc](#)
- [SetupTimeProfiler](#)
- [UserTargetProc](#)

3 - Extras:

- [CpuSettings](#)
- [GetCpuFrequencyEx](#)

II - Structures

- [CPUData](#)
- [CT_STANDARD_DEVIATION](#)
- [CT_STDEX](#)
- [CT_Nfo](#)

III - Equates

- [CPU_CPUID_AVAILABLE](#)
- [CPU_RDTSCP_AVAILABLE](#)
- [CPU_RDTSC_AVAILABLE](#)
- [CT_ALGO1](#)
- [CT_ALGO2](#)
- [CT_ALGO3](#)
- [CT_ALGO4](#)
- [CT_ALGO5](#)
- [CT_ALGO6](#)
- [CT_ALGO7](#)
- [CT_ALGO8](#)

CodeTune Reference Guide

- [CT ALGO_METHOD_ERROR](#)
- [CT ANALYSIS_ERROR1](#)
- [CT ANALYSIS_ERROR2](#)
- [CT ANALYSIS_ERROR3](#)
- [CT ANALYSIS_ERROR4](#)
- [CT ANALYSIS_START](#)
- [CT ANALYSIS_SUCESS](#)
- [CT BENCHMARK_FINISHED](#)
- [CT BENCHMARK_RUNNING](#)
- [CT BENCHMARK_START](#)
- [CT CALIBRATION_FINISHED](#)
- [CT CALIBRATION_RUNNING](#)
- [CT CALIBRATION_START](#)
- [CT_ERROR_BENCHMARK_OVERHEAD](#)
- [CT_ERROR_CALIBRATION_OVERHEAD](#)
- [CT_ERROR_INPUT_VALUE](#)
- [CT_INCONCLUSIVE](#)
- [CT_INSUFFICIENT_FEATURES](#)
- [CT_STATUSCODE_ERROR](#)
- [MAX_ITERATIONS](#)
- [MAX_SAMPLES](#)
- [OVERHEAD_LIMIT](#)

IV – Type Definitions

- [LP_RUNTIME_DATA_CALLBACK_ROUTINE](#)
- [LP_USER_TARGET_CALLBACK_ROUTINE](#)

V – Additional Information

- [APENDIX I - Understanding Mean, Variance and Standard Deviation](#)
 - [1\) How to Find the Mean](#)
 - [2\) Standard Deviation and Variance](#)
 - [3\) Standard Deviation Formulas](#)
- [APENDIX II - Standard Deviation and Variance](#)

I - FUNCTIONS

CreateTimeProfile function

Create the timing profile. This is the main function of CodeTune. It creates the time profile from where you can analyze how fast your function runs. CreateTimeProfile calibrates itself internally and interprets the results in order to try to achieve a high level of accuracy when the function finishes.

The function also checks if your CPU supports the chosen method. I.e.: If your processor has, for example, support for rdtscp, cupid, rdtsc etc. If you processor don't support the chosen method used by these instructions, a proper error message is retrieved on dwStatusCode parameter.

CreateTimeProfile works in three main stages on the following order:

- a) **Calibration:** from where it tries to minimize potential errors,
- b) **Benchmarking:** to collect the timings and choose the ones considered as best candidates for being measured. The timing values are then used as samples to be passed through a Standard Deviation computation. On this stage, it also computes the amount of overheads and errors found while the samples were being collected. Once the analysis of each sample is finished and it passed through some criteria's to minimize potential errors on the result, it is considered as a "good sample". The benchmark stage collects as many "good samples" as defined by the user on input.
- c) **Interpretation:** to perform a fine tune on the samples chosen on the Benchmarking stage, keeping or excluding samples to be used on a Standard deviation Computation to find the best Values considered the minimum as possible to the near timing your function is working.

For a more complete functionality, see [CreatetimeProfileEx](#).

Syntax

```
PCT STANDARD DEVIATION CreateTimeProfile (
    _In_      UINT      Samples,
    _In_      UINT      Iterations,
    _In_      UINT      AlgoMethod,
    _Out_opt_ LP\_RUNTIME\_DATA\_CALLBACK\_ROUTINE lpCallBack,
    _In_      LP\_USERTARGET\_CALLBACK\_ROUTINE lpStartAddress
    _Out_     PINT      dwStatusCode,
);
```

Parameters

Samples [in]

The total amount of samples you want to be analyzed on each iteration. This value will be used to compute the Standard Deviation of the total amount of time your function will take to run.

CodeTune Reference Guide

The minimum value is 1 (one), because since we are analyzing the Standard Deviation of the collected samples, it cannot perform the computation of only 1 sample, because it will then be computing the single value itself.

The maximum amount of samples you can use on input is 100000 defined as the equate `MAX_SAMPLES`.

For a regular day usage in real life applications, 300 samples to be analyzed are enough for the function creates a correct summary of the timing profile which will be stored on the [CT_STANDARD_DEVIATION](#) structure when this function exits successfully.

However, you can increase the amount of samples and get better results, but keep in mind that the more samples you are analyzing, more time the function will take to finish.

By experiment, the overall error in the timing computation is around 1 nanosecond.

These samples are the ones to be considered as “good ones”. So, are the amount of samples that you want [CreateTimeProfile](#) effectively uses after excluding potential errors during the benchmark or calibration internal stages.

Iterations [in]

The total amount of iterations (loops) you want the function to perform per each sample.

The minimum value is 1 (one) and the maximum is 100000 defined as the equate `MAX_ITERATIONS`.

For a regular day usage in real life applications, 3000 iterations to be performed are enough for the function creates a correct summary of the timing profile which will be stored on the [CT_STANDARD_DEVIATION](#) structure when this function exits successfully.

However, you can increase the amount of samples and get better results, but keep in mind that the more iterations you are doing, more time the function will take to finish.

It is not uncommon to have variations when analyzing different amount of iterations or samples or Algo Methods. This is due to small variations on the CPU frequency and the accumulation of errors generated by the underlying “clock” of the processor who is only ticking at a rate of about 100 nanoseconds which may interferes on the chosen Algo method used for the analysis.

So, to get a better analysis of the results, it is a good strategy you perform the computation twice using different values for samples and one single value for the iterations under different methods.

CodeTune Reference Guide

This can ensure a more accurate time analysis. For example, you can have these timings:

Method Chosen: Algo Method 1 (cpuid+rdtsc)
Samples: 300
Iterations: 3000
Resultant mean: 5.89280439262591

Samples: 3000
Iterations: 3000
Resultant mean: 5.34815722924948

The one with the smaller value is the one that is closer to the real timing of your tested function. Also, consider analyzing the value found on the `SampleStd.Min` or `PopulationStd.Min` member of the structure [CT_STANDARD_DEVIATION](#) using different Algo methods. This is because, [CreateTimeProfile](#) tries to keep the results on the lower value as possible (*after excluding errors generated during the internal analysis*) to determine what is the closer to the real timing your function is running. So, the values found on the `Mean` member of the [CT_STANDARD_DEVIATION](#) are an approximation due to the minimum fluctuations of the results, but, the value found on the `Minimum` data analyzed by the Standard deviation Summary is the one that better represents the correct time. Although the interpretation of the summary of the Standard Deviation is subjective, [CreateTimeProfile](#) tries to keep them the more accurate as possible.

Do not increase the value of iterations too much. It has 2 side-effects. One is that it will take more time to finish. The other is that it may retrieve incorrect results, since you may be measuring samples that are over clocked or with a high level of errors accumulated.

That's why there is a limitation on the total amount of samples and iterations to be used. Keep in mind that the total amount of operations the function will perform is, at least, `Samples X Iterations`. So, if you try 3000 samples X 3000 iterations you are performing 9.000.000 operations, not considering the ones that were discarded internally (which can easily double or triple that amount of operations involved)

AlgoMethod [in]

Use one of the following flags to specify which algorithm you want to use to perform the timing computation.

Name	Value	Description
CT_ALGO1	1	This method uses <code>cpuid + rdtsc</code> opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) <code>cpuid</code> <code>rdtsc</code> (...)

CodeTune Reference Guide

CT_ALGO2	2	<p>This method uses lfence + rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) lfence rdtsc (...)</pre>
CT_ALGO3	3	<p>This method uses lfence + rdtsc + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) lfence rdtsc lfence (...)</pre>
CT_ALGO4	4	<p>This method uses rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) rdtscp (...)</pre>
CT_ALGO5	5	<p>This method uses lfence + rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) lfence rdtscp (...)</pre>
CT_ALGO6	6	<p>This method uses cpuid + rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) cpuid rdtscp (...)</pre>
CT_ALGO7	7	<p>This method uses rdtscp + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) rdtscp lfence (...)</pre>
CT_ALGO8	8	<p>This method uses rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) rdtsc (...)</pre>

CodeTune Reference Guide

For a better interpretation, you may want to use different methods and analyze the resultant data on each one of them.

Since the different methods may interfere on the results (*depending on the function you are testing*), it is wise to use several methods and choose the one that result in the smaller value.

LpCallback [out/optional]

A pointer to a application-defined callback function from where the collected data is stored while the function is still running. The callback function contains only 2 parameters.

One that is a pointer to a structure ([CT_Nfo](#)) containing the values internally stored from [CreateTimeProfile](#) such as: overheads, the amount of good samples that are being collected while the main function is operating in the calibration mode and benchmark mode.

And the second parameter used as a placeholder for the Status of the function while it is running internally.

This parameter is optional, but you may use it to get the runtime measures of the internal stages of the function. For example, if you may want to display the results of the good samples that are being collected or the overheads while the function is running, to display those values in runtime you can use put the [CreateTimeProfile](#) function inside a Thread with the CreateThread Windows Api.

If you don't want to use a callback function, simply set this parameter as 0. For more information, see [RunTimeDataProc](#).

lpStartAddress [in]

A pointer to a function (void = without any parameters) from where the user can insert his code to be analyzed. This function has no parameters.

This is the main part of the [CreateTimeProfile](#) Api. On this parameter you must point to your code/function that you want to be analyzed.

For more information, see [UserTargetProc](#).

dwStatusCode [out]

A reference to a status code. Status codes indicate the success or failure of the function. It is a pointer to a UINT (32 bits: Dword) variable from where the status code is stored.

If the function fails it returns 0 and also outputs the following status code.

Name	Value	Description
CT_ERROR_INPUT_VALUE	0	The user inputted an incorrect value for Samples or Iterations. The amount of each one of them are limited to 1 to 100000 as defined in MAX_SAMPLES and

CodeTune Reference Guide

		MAX_ITERATIONS
CT_ERROR_CALIBRATION_OVERHEAD	0-1	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during calibration.
CT_ERROR_BENCHMARK_OVERHEAD	0-2	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during Benchmarking.
CT_ANALYSIS_ERROR1	0-3	This happens when all samples collected in benchmark are smaller or equal to the calibration. So, the user code is too short to perform a full analysis, because the app can distinguish between both. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions
CT_INCONCLUSIVE	0-4	Benchmark time fits the required range to be considered a good sample, but only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this.
CT_ANALYSIS_ERROR2	0-5	Sample too short and only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions.
CT_ANALYSIS_ERROR3	0-6	Although the sample fits the required range, not all of them fit the required characteristics to be a good sample. The resultant Standard deviation has negative values.
CT_ANALYSIS_ERROR4	0-7	Happens on too short code or non existent. Not all of them fit the required characteristics to be a good sample. The resultant Standard deviation has negative values. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions
CT_ALGO_METHOD_ERROR	0-8	Invalid algo method. The chosen algorithm doesn't exists on the user CPU
CT_STATUSCODE_ERROR	0-9	The user did not inputted a pointer to the Status code
CT_INSUFFICIENT_FEATURES	0-10	Your processor does not supports the minimum instructions to perform a timing measure.

If the function succeeds and also outputs the following status code.

CT_CALIBRATION_START	1	Calibration Started
CT_CALIBRATION_RUNNING	2	Calibration is Running
CT_CALIBRATION_FINISHED	3	Calibration Finished
CT_BENCHMARK_START	4	Benchmark Started
CT_BENCHMARK_RUNNING	5	Benchmark is Running
CT_BENCHMARK_FINISHED	6	Benchmark Finished
CT_ANALYSIS_START	7	Analysis Started
CT_ANALYSIS_SUCESS	8	Analysis Completed Successfully

Return value

If the function fails, it returns 0.

If the function succeeds it returns a pointer to a [CT_STANDARD_DEVIATION](#) structure containing the resultant values timings of the tested user's code.

Remarks

For a more accurate result, consider aligning your functions with a 16 byte boundary. To understand why you would want to align the code (or data) in a section on a given boundary, get more information about how cache lines work.

By aligning the start of a function on a cache line, you may be able to slightly increase the execution speed of that function as it may generate fewer cache misses during execution. For this reason, many programmers like to align all their functions at the start of a cache line.

Although the size of a cache line varies from CPU to CPU, a typical cache line is 16 to 64 bytes long, so many compilers, assemblers, and linkers will attempt to align code and data to one of these boundaries. On the 80x86 processor, there are some other benefits to 16-byte alignment, so many 80x86-based tools default to a 16-byte section alignment for object files.

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
Library	CodeTune.lib
DLL	CodeTune.dll

See Also

[CreatetimeProfileEx](#)

[UserTargetProc](#)

[RunTimeDataProc](#)

[CT_STANDARD_DEVIATION](#)

[CT_Nfo](#)

[LP_RUNTIMEDATA_CALLBACK_ROUTINE](#)

[LP_USERTARGET_CALLBACK_ROUTINE](#)

Examples of Usage (*RosAsm Syntax*):

- 1) Using [CreateTimeProfile](#) to display the results without Showing what is happening inside of it.

```
(...)  
call 'CodeTune.CreateTimeProfile' 300, 3000, CT_ALGO3, 0, AlgoFcn, StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STANDARD_DEVIATION.MEANDis | fstp R$Mean  
(...)
```

CodeTune Reference Guide

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:  
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}  
EndP
```

- 2) Using [CreateTimeProfile](#) to display the results showing what is happening inside of it.

```
(...)  
call 'CodeTune.CreateTimeProfile' 300, 3000, CT_ALGO3, RunTimeDataProc, AlgoFcn,  
StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STANDARD_DEVIATION.MEANDis | fstp R$Mean  
(...)
```

Display the StatusCode messages when [CreateTimeProfile](#) is finished

```
Proc RunTimeDataProc:  
    Arguments @pOutStruct, @dwStatusCode  
(...)  
    .If D@dwStatusCode = CT_ANALYSIS_ERROR1  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, {B$ "Samples collected 0", 0}  
        C_call 'msvcrt.sprintf' SzPass, {B$ "Samples collected %.d", 0},  
D$edi+CT_Nfo.GoodSamplesDis  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, SzPass  
        call 'USER32.SetDlgItemTextA' D$ThisWindow, IDC_WARNING_ERROR_MESSAGE,  
Sz_Err1  
    .End_If  
(...)  
EndP
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:  
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}  
EndP
```

- 3) Using [CreateTimeProfile](#) from Inside a thread without displaying error messages

```
(...)  
call 'KERNEL32.CreateThread' &NULL, &NULL, MyThread, D@hwnd, &FALSE, ThreahID  
(...)  
  
Proc MyThread:  
    Argument @lParam  
(...)
```

CodeTune Reference Guide

```
call 'CodeTune.CreateTimeProfile' 300, 3000, CT_ALGO3, 0, AlgoFcn, StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STANDARD_DEVIATION.MEANDis | fstp R$Mean  
    (...)  
EndP
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:  
  
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}  
  
EndP
```

- 4) Using [CreateTimeProfile](#) from Inside a thread displaying error messages in runtime (while the thread is running)

```
(...)  
call 'KERNEL32.CreateThread' &NULL, &NULL, MyThread, D@hwnd, &FALSE, ThreadID  
(...)  
  
Proc MyThread:  
    Argument @lParam  
(...)  
call 'CodeTune.CreateTimeProfile' 300, 3000, CT_ALGO3, RunTimeDataProc, AlgoFcn,  
StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STANDARD_DEVIATION.MEANDis | fstp R$Mean  
    mov eax D$StatusCode  
(...)  
EndP
```

Display the StatusCode messages on runtime (Thread is not finished)

```
Proc RunTimeDataProc:  
    Arguments @pOutStruct, @dwStatusCode  
(...)  
    .If D@dwStatusCode = CT_ANALYSIS_ERROR1  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, {B$ "Samples collected 0", 0}  
        C_call 'msvcrt.sprintf' SzPass, {B$ "Samples collected %.d", 0},  
D$edi+CT_Nfo.GoodSamplesDis  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, SzPass  
        call 'USER32.SetDlgItemTextA' D$ThisWindow, IDC_WARNING_ERROR_MESSAGE,  
Sz_Err1  
    .End_If  
(...)  
EndP
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

CodeTune Reference Guide

Proc **AlgoFcn**:

```
C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}
```

EndP

CreateTimeProfileEx function

Create the timing profile using all available Methods in CodeTune Library. This is an extension to [CreateTimeProfile](#) function. It creates the time profile from where you can analyze how fast your function runs. CreateTimeProfileEx calibrates itself internally and interprets the results in order to try to achieve a high level of accuracy when the function finishes.

It uses all available Algorithm methods for your CPU as described in [CreateTimeProfile](#) with the advantage that you don't need to select which one to be used since the function will create the summary biased on all of them at once.

The function also checks if your CPU supports the chosen method. I.e.: If your processor has, for example, support for rdtscp, cupid, rdtsc etc. If your processor doesn't support the chosen method used by these instructions, a proper error message is retrieved on dwStatusCode parameter.

CreateTimeProfileEx analyses all available Algo Methods and on each one of them works in three main stages on the following order:

- a) **Calibration:** from where it tries to minimize potential errors,
- b) **Benchmarking:** to collect the timings and choose the ones considered as best candidates for being measured. The timing values are then used as samples to be passed through a Standard Deviation computation. On this stage, it also computes the amount of overheads and errors found while the samples were being collected. Once the analysis of each sample is finished and it passed through some criteria's to minimize potential errors on the result, it is considered as a "good sample". The benchmark stage collects as many "good samples" as defined by the user on input.
- c) **Interpretation:** to perform a fine tune on the samples chosen on the Benchmarking stage, keeping or excluding samples to be used on a Standard deviation Computation to find the best Values considered the minimum as possible to the near timing your function is working.

Syntax

```
PCT_STDEx CreateTimeProfileEx (
    _In_      UINT      Samples,
    _In_      UINT      Iterations,
    _Out_opt_ LP\_RUNTIME\_DATA\_CALLBACK\_ROUTINE lpCallBack,
    _In_      LP\_USERTARGET\_CALLBACK\_ROUTINE lpStartAddress
    _Out_     PINT      dwStatusCode,
);
```

Parameters

Samples [in]

The total amount of samples you want to be analyzed on each iteration. This value will be used to compute the Standard Deviation of the total amount of time your function will take to run.

The minimum value is 1 (one), because since we are analyzing the Standard Deviation of the collected samples, it cannot perform the computation of only 1 sample, because it will then be computing the single value itself.

The maximum amount of samples you can use on input is 100000 defined as the equate [MAX_SAMPLES](#).

For a regular day usage in real life applications, 300 samples to be analyzed are enough for the function creates a correct summary of the timing profile which will be stored on the [CT_STDEx](#) structure when this function exits successfully.

However, you can increase the amount of samples and get better results, but keep in mind that the more samples you are analyzing, more time the function will take to finish.

By experiment, the overall error in the timing computation is around 1 nanosecond.

These samples are the ones to be considered as “good ones”. So, are the amount of samples that you want [CreateTimeProfile](#) effectively uses after excluding potential errors during the benchmark or calibration internal stages.

Iterations [in]

The total amount of iterations (loops) you want the function to perform per each sample.

The minimum value is 1 (one) and the maximum is 100000 defined as the equate [MAX_ITERATIONS](#).

For a regular day usage in real life applications, 3000 iterations to be performed are enough for the function creates a correct summary of the timing profile which will be stored on the [CT_STDEx](#) structure when this function exits successfully.

However, you can increase the amount of samples and get better results, but keep in mind that the more iterations you are doing, more time the function will take to finish.

It is not uncommon to have variations when analyzing different amount of iterations or samples. This is due to small variations on the CPU frequency and the accumulation of errors generated by the underlying “clock” of the processor who is only ticking at a rate of about 100 nanoseconds which may interferes in one or more of the Algo methods used internally for the analysis.

So, to get a better analysis of the results, it is a good strategy you perform the computation twice using different values for samples and one single value for the iterations.

The function will always choose the smaller value found that is closer to the real timing of your tested function.

Also, consider analyzing the value found on the BestTiming member of the structure [CT_STDEx](#). This is because, [CreateTimeProfileEx](#) tries to keep the results on the lower value as possible (*after excluding errors generated during the internal analysis*) to determine what is the closer to the real timing your function is running. So, the value of the Mean found on the Fast.Mean member of the [CT_STDEx](#) is a approximation due to the minimum fluctuations of the results, but, the value found on the BestTiming member of this structure is the one that better represents the correct time.

Although the interpretation of the summary of the Standard Deviation is subjective, [CreateTimeProfileEx](#) tries to keep them the more accurate as possible.

Do not increase the value of iterations too much. It has 2 side-effects. One is that it will take more time to finish. The other is that it may retrieve incorrect results, since you may be measuring samples that are over clocked or with a high level of errors accumulated.

That's why there is a limitation on the total amount of samples and iterations to be used. Keep in mind that the total amount of operations the function will perform is, at least, Samples X Iterations. So, if you try 3000 samples X 3000 iterations you are performing 9.000.000 operations, not considering the ones that were discarded internally (which can easily double or triple that amount of operations involved)

LpCallback [out/optional]

A pointer to a application-defined callback function from where the collected data is stored while the function is still running. The callback function contains only 2 parameters.

One that is a pointer to a structure ([CT_Nfo](#)) containing the values internally stored from [CreateTimeProfileEx](#) such as: overheads, the Algo Methods that is being currently analyzed internally, the amount of good samples that are being collected while the main function is operating in the calibration mode and benchmark mode.

And the second parameter used as a placeholder for the Status of the function whiles it is running internally.

This parameter is optional, but you may use it to get the runtime measures of the internal stages of the function. For example, if you may want to display the results of the good samples that are being collected or the overheads while the function is running, to display those values in runtime you can use put the [CreateTimeProfileEx](#) function inside a Thread with the CreateThread Windows Api.

CodeTune Reference Guide

If you don't want to use a callback function, simply set this parameter as 0. For more information, see [RunTimeDataProc](#).

lpStartAddress [in]

A pointer to a function (*void = without any parameters*) from where the user can insert his code to be analyzed. This function has no parameters.

This is the main part of the [CreateTimeProfileEx](#) Api. On this parameter you must point to your code/function that you want to be analyzed.

For more information, see [UserTargetProc](#).

dwStatusCode [out]

A reference to a status code. Status codes indicate the success or failure of the function. It is a pointer to a UINT (32 bits: Dword) variable from where the status code is stored.

If the function fails it returns 0 and also outputs the following status code.

Name	Value	Description
CT_ERROR_INPUT_VALUE	0	The user inputted an incorrect value for Samples or Iterations. The amount of each one of them are limited to 1 to 100000 as defined in MAX_SAMPLES and MAX_ITERATIONS
CT_ERROR_CALIBRATION_OVERHEAD	0-1	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during calibration.
CT_ERROR_BENCHMARK_OVERHEAD	0-2	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during Benchmarking.
CT_ANALYSIS_ERROR1	0-3	This happens when all samples collected in benchmark are smaller or equal to the calibration. So, the user code is too short to perform a full analysis, because the app can distinguish between both. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions
CT_INCONCLUSIVE	0-4	Benchmark time fits the required range to be considered a good sample, but only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this.
CT_ANALYSIS_ERROR2	0-5	Sample too short and only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions.
CT_ANALYSIS_ERROR3	0-6	Although the sample fits the required range, not all of them fit the required characteristics

CodeTune Reference Guide

		to be a good sample. The resultant Standard deviation has negative values.
CT_ANALYSIS_ERROR4	0-7	Happens on too short code or non existent. Not all of them fit the required characteristics to be a good sample. The resultant Standard deviation has negative values. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions
CT_ALGO_METHOD_ERROR	0-8	Invalid algo method. The chosen algorithm doesn't exists on the user CPU
CT_STATUSCODE_ERROR	0-9	The user did not inputted a pointer to the Status code
CT_INSUFFICIENT_FEATURES	0-10	Your processor does not supports the minimum instructions to perform a timing measure.

If the function succeeds and also outputs the following status code.

CT_CALIBRATION_START	1	Calibration Started
CT_CALIBRATION_RUNNING	2	Calibration is Running
CT_CALIBRATION_FINISHED	3	Calibration Finished
CT_BENCHMARK_START	4	Benchmark Started
CT_BENCHMARK_RUNNING	5	Benchmark is Running
CT_BENCHMARK_FINISHED	6	Benchmark Finished
CT_ANALYSIS_START	7	Analysis Started
CT_ANALYSIS_SUCESS	8	Analysis Completed Successfully

Return value

If the function fails, it returns 0.

If the function succeeds it returns a pointer to a [CT_STDEx](#) structure containing the resultant values timings of the tested user's code.

Remarks

For a more accurate result, consider aligning your functions with a 16 byte boundary. To understand why you would want to align the code (or data) in a section on a given boundary, get more information about how cache lines work.

By aligning the start of a function on a cache line, you may be able to slightly increase the execution speed of that function as it may generate fewer cache misses during execution. For this reason, many programmers like to align all their functions at the start of a cache line.

Although the size of a cache line varies from CPU to CPU, a typical cache line is 16 to 64 bytes long, so many compilers, assemblers, and linkers will attempt to align code and data to one of these boundaries. On the 80x86 processor, there are some other benefits to 16-byte alignment, so many 80x86-based tools default to a 16-byte section alignment for object files.

CodeTune Reference Guide

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
Library	CodeTune.lib
DLL	CodeTune.dll

Examples of Usage (*RosAsm Syntax*):

- 1) Using [CreateTimeProfileEx](#) to display the results without showing what is happening inside of it.

```
(...)  
call 'CodeTune.CreateTimeProfileEx' 300, 3000, 0, AlgoFcn, StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STDEx.BestTimingDis | fstp R$fastestValue  
(...)
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:  
  
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}  
  
EndP
```

- 2) Using [CreateTimeProfileEx](#) to display the results showing what is happening inside of it.

```
(...)  
call 'CodeTune.CreateTimeProfileEx' 300, 3000, RunTimeDataProc, AlgoFcn, StatusCode  
  
    mov esi eax  
    fld R$esi+ CT_STDEx.BestTimingDis | fstp R$fastestValue  
(...)
```

Display the StatusCode messages when [CreateTimeProfileEx](#) is finished

```
Proc RunTimeDataProc:  
    Arguments @pOutStruct, @dwStatusCode  
(...)  
    .If D@dwStatusCode = CT_ANALYSIS_ERROR1  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, {B$ "Samples collected 0", 0}  
        C_call 'msvcrt.sprintf' SzPass, {B$ "Samples collected %.d", 0},  
D$edi+CT_Nfo.GoodSamplesDis  
        call 'user32.SetWindowTextA' D$hWarning_Goodpass, SzPass  
        call 'USER32.SetDlgItemTextA' D$ThisWindow, IDC_WARNING_ERROR_MESSAGE,  
Sz_Err1  
    .End_If  
(...)  
EndP
```

CodeTune Reference Guide

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:
```

```
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}
```

```
EndP
```

- 3) Using [CreateTimeProfileEx](#) from Inside a thread without displaying error messages

```
(...)  
call 'KERNEL32.CreateThread' &NULL, &NULL, MyThread, D@hwnd, &FALSE, ThreahID  
(...)
```

```
Proc MyThread:
```

```
    Argument @lParam
```

```
(...)
```

```
call 'CodeTune.CreateTimeProfileEx' 300, 3000, 0, AlgoFcn, StatusCode
```

```
    mov esi eax
```

```
    fld R$esi+ CT_STDEx.BestTimingDis | fstp R$fastestValue
```

```
(...)
```

```
EndP
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:
```

```
    C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}
```

```
EndP
```

- 4) Using [CreateTimeProfileEx](#) from Inside a thread displaying error messages in runtime (while the thread is running)

```
(...)  
call 'KERNEL32.CreateThread' &NULL, &NULL, MyThread, D@hwnd, &FALSE, ThreahID  
(...)
```

```
Proc MyThread:
```

```
    Argument @lParam
```

```
(...)
```

```
call 'CodeTune.CreateTimeProfileEx' 300, 3000, RunTimeDataProc, AlgoFcn, StatusCode
```

```
    mov esi eax
```

```
    fld R$esi+ CT_STDEx.BestTimingDis | fstp R$fastestValue
```

```
    mov eax D$StatusCode
```

```
(...)
```

```
EndP
```

CodeTune Reference Guide

Display the StatusCode messages on runtime (Thread is not finished)

```
Proc RunTimeDataProc:  
  Arguments @pOutStruct, @dwStatusCode  
  (...)  
  .If D@dwStatusCode = CT_ANALYSIS_ERROR1  
    call 'user32.SetWindowTextA' D$hWarning_Goodpass, {B$ "Samples collected 0", 0}  
    C_call 'msvcrt.sprintf' SzPass, {B$ "Samples collected %.d", 0},  
D$edi+CT_Nfo.GoodSamplesDis  
    call 'user32.SetWindowTextA' D$hWarning_Goodpass, SzPass  
    call 'USER32.SetDlgItemTextA' D$ThisWindow, IDC_WARNING_ERROR_MESSAGE,  
Sz_Err1  
  .End_If  
  (...)  
EndP
```

This is where the main user code to be analyzed must be used. On this example, we are analyzing the timing of the Api strlen inserting it inside a void function called “AlgoFcn”

```
Proc AlgoFcn:  
  
  C_call 'msvcrt.strlen' {B$ "Hello, this is a simple string", 0}  
  
EndP
```

RunTimeDataProc callback function

An application-defined callback function used as Storage of information retrieved by [CreateTimeProfile](#) or [CreateTimeProfileEx](#) functions. Specify this address when calling those functions.

Syntax

```
DWORD RunTimeDataProc (
    _Out_ PCT\_Nfo      pUserStruct,
    _Out_ INT          dwStatusCode,
);
```

Parameters

pUserStruct [out]

A pointer to a [CT_Nfo](#) structure that contains the values internally stored from [CreateTimeProfile](#) or [CreateTimeProfileEx](#) such as: overheads, the amount of good samples that are being collected while the main function is operating in the calibration mode and benchmark mode.

dwStatusCode [out]

A reference to a status code. Status codes indicate the success or failure of the function. It is a INT (32 bits: Dword) variable. For more information on the used values and equates, see: [CreateTimeProfile](#), [CreateTimeProfileEx](#)

Return value

The function does not return any value

Remarks

None

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h

Examples of Usage:

See [CreateTimeProfile](#) or [CreateTimeProfileEx](#) functions

SetupTimeProfiler function

Initialize and configure all internal data of the [CreateTimeProfile](#) and [CreateTimeProfileEx](#) functions. You may use this function on the initialization of your application. For example, on the WM_INITDIALOG or WM_CREATE messages, or when it starts at your Main function.

Although using SetupTimeProfiler is not mandatory for [CreateTimeProfile](#) or [CreateTimeProfileEx](#) to work (*Since that Api already setups itself internally*), you may want to previously setup CreateTime function in order to it runs a bit faster on the 1st time you call it.

Syntax

```
void SetupTimeProfiler (  
);
```

Parameters

None

Return value

The function does not return any value

Remarks

None

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
Library	CodeTune.lib
DLL	CodeTune.dll

Example of Usage (*RosAsm Syntax*):

```
Main:  
  call 'CodeTune.SetupTimeProfiler'  
  call 'kernel32.GetModuleHandleA' &NULL  
  call WinMain  
(...)
```

See also

[CreateTimeProfile](#)
[CreateTimeProfileEx](#)

UserTargetProc callback function

An application-defined callback function that notify [CreateTimeProfile](#) and [CreateTimeProfileEx](#) functions to start timing the user's code inside of it.

This is the most important callback function used as parameter in [CreateTimeProfile](#) and [CreateTimeProfileEx](#) function. From here the actual timing is being measure, since this is where the user must insert his code to be analyzed.

Syntax

```
DWORD UserTargetProc (  
);
```

Parameters

None

Return value

The function does not return any value

Remarks

None

Requirements

Minimum supported client Windows2000

Header CodeTune.h

Examples of Usage:

See [CreateTimeProfile](#) or [CreateTimeProfileEx](#) functions

See also

[CreateTimeProfile](#)

[CreateTimeProfileEx](#)

[LP_USERTARGET_CALLBACK_ROUTINE](#)

CpuSettings function

Provides information about your CPU such as clock frequency, vendor Id string, and if your processor supports Cpuid, rdtsc, rdtscp instructions.

Syntax

```
DWORD CpuSettings (
    _Out_ PCPUData pUserStruct,
);
```

Parameters

pUserStruct [out]

A pointer to a CPUData structure that contains information about your processor.

Return value

If the function fails, it returns FALSE

If the function succeeds, it returns TRUE

Remarks

The function will return FALSE if your processor doesn't support any of the necessary instructions for the function to work. I.e: Processors earlier then a 486 or that doesn't have support for cpuid instruction.

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
Library	CodeTune.lib
DLL	CodeTune.dll

Examples of Usage (*RosAsm Syntax*):

```
[CPUData:
CPUData.Flags: D$ 0
CPUData.ClockFrequency: R$ 0
CPUData.String: D$ 0]

(...)
    call 'CodeTune.CpuSettings' CPUData
(...)
```

See Also

[CpuData](#)

GetCpuFrequencyEx

This function retrieves the Clock frequency of your CPU.

Syntax

```
DWORD CpuFrequencyEx (  
    _Out_ PDouble    pOutFrequency,  
);
```

Parameters

pOutFrequency [out]

A pointer to a variable that stores the frequency of the CPU (In Gigahertz). The size of the value is FPU double: REAL, that is a 64 bit value.

Return value

If the function fails, it returns FALSE

If the function succeeds, it returns TRUE

Remarks

The function will return FALSE if your processor doesn't support the necessary instructions for the function to work. I.e: Processors earlier than a 486 or that don't have support for cpuid instruction.

To check if your processor have support for this instructions, consider using [CpuSettings](#) function, instead.

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
Library	CodeTune.lib
DLL	CodeTune.dll

Examples of Usage (*RosAsm Syntax*):

```
[ClockFrequency: R$ 0]  
(...  
    call 'CodeTune.CpuFrequencyEx' ClockFrequency  
(...)
```

See Also

[CpuSetings](#)

II- STRUCTURES

CPUData structure

Displays information about CPU settings, such as: frequency, instructions and CPUId vendor string.

Syntax

C++

```
typedef struct _CPUData {
    uint  Flags;
    double  ClockFrequency;
    *char  String;
} CPUData, *PCPUData
```

RosAsm

```
[CPUData:
CPUData.Flags: D$ 0
CPUData.ClockFrequency: R$ 0
CPUData.String: D$ 0]
```

Masm

```
CPUData struct
    Flags DWORD ?
    ClockFrequency REAL8 ?
    String DWORD ?
CPUData ends
```

Structure Displacement

(Equates related to the position of each member and size of the structure)

```
CPUData.FlagsDis 0
CPUData.ClockFrequencyDis 4
CPUData.StringDis 12

Size_Of_CPUData 16
```

Members

Flags

A flag containing information about the capability of the user's processor. It can be a combination of the following equates.

Name	Value	Description
CPU_CPUID_AVALIABLE	1	The CPU supports cpuid instruction
CPU_RDTSCP_AVALIABLE	2	The CPU supports rdtscp instruction
CPU_RDTSC_AVALIABLE	4	The CPU supports rdtsc instruction

CodeTune Reference Guide

ClockFrequency

The frequency of the CPU (in Gigahertz). The size of the value is FPU double: REAL, that is a 64 bit value.

String

A Pointer to the CPU's manufacturer ID string – a twelve-character ASCII string.

The following are known processor manufacturer ID strings:

String	Description
"AMDisbetter!"	early engineering samples of AMD K5 processor
"AuthenticAMD"	AMD
"CentaurHauls"	Centaur (Including some VIA CPU)
"CyrixInstead"	Cyrix
"GenuineIntel"	Intel
"TransmetaCPU"	Transmeta
"GenuineTMx86"	Transmeta
"Geode by NSC"	National Semiconductor
"NexGenDriven"	NexGen
"RiseRiseRise"	Rise
"SiS SiS SiS "	SiS
"UMC UMC UMC "	UMC
"VIA VIA VIA "	VIA
"Vortex86 SoC"	Vortex

The following are known ID strings from virtual machines:

String	Description
"KVMKVMKVM"	KVM
"Microsoft Hv"	Microsoft Hyper-V or Windows Virtual PC
" lrpepyh vr"	Parallels (it possibly should be "prl hyperv ", but it is encoded as " lrpepyh vr")
"VMwareVMware"	VMware
"XenVMMXenVMM"	Xen HVM

Remarks

None.

Requirements

Minimum supported client Windows2000
Header CodeTune.h

Examples of Usage:

See [CpuSettings](#) function

See also

[CpuSettings](#)

CT_STANDARD_DEVIATION structure

Displays Standard Deviation values measured from [CreateTimeProfile](#) function. The values are displayed in nanoseconds.

The Standard Deviation is a measure of how spread out numbers are. Its symbol is σ (the Greek letter sigma)

When only a sample of data from a population is available, the term standard deviation of the sample or sample standard deviation can refer to either the above-mentioned quantity as applied to those data or to a modified quantity that is a better estimate of the population standard deviation (the standard deviation of the entire population).

By using standard deviations, a minimum and maximum value can be calculated that the averaged weight will be within some very high percentage of the time (99.9% or more).

Syntax

C++

```
typedef struct _CT_STANDARD_DEVIATION {
    double Mean;
    double PopulationStd.Max;
    double PopulationStd.Min;
    double PopulationStd.Variance;
    double PopulationStd.StandardDeviation;
    double SampleStd.Max;
    double SampleStd.Min;
    double Sample.Variance;
    double Sample.StandardDeviation;
} CT_STANDARD_DEVIATION, *PCT_STANDARD_DEVIATION
```

RosAsm

```
[CT_STANDARD_DEVIATION:
CT_STANDARD_DEVIATION.Mean: R$ 0
CT_STANDARD_DEVIATION.PopulationStd.Max: R$ 0
CT_STANDARD_DEVIATION.PopulationStd.Min: R$ 0
CT_STANDARD_DEVIATION.PopulationStd.Variance: R$ 0
CT_STANDARD_DEVIATION.PopulationStd.StandardDeviation: R$ 0
CT_STANDARD_DEVIATION.SampleStd.Max: R$ 0
CT_STANDARD_DEVIATION.SampleStd.Min: R$ 0
CT_STANDARD_DEVIATION.Sample.Variance: R$ 0
CT_STANDARD_DEVIATION.Sample.StandardDeviation: R$ 0]
```

Masm

```
CT_STANDARD_DEVIATION struct
    Mean REAL8 ?
```

CodeTune Reference Guide

```
Population_StdMax REAL8 ?
Population_StdMin REAL8 ?
Population_StdVariance REAL8 ?
Population_StdStandardDeviation REAL8 ?
Sample_StdMax REAL8 ?
Sample_StdMin REAL8 ?
Sample_Variance REAL8 ?
Sample_StandardDeviation REAL8 ?
CT_STANDARD_DEVIATION ends
```

Structure Displacement

(Equates related to the position of each member and size of the structure)

```
CT_STANDARD_DEVIATION.MeanDis 0
CT_STANDARD_DEVIATION.PopulationStd.MaxDis 8
CT_STANDARD_DEVIATION.PopulationStd.MinDis 16
CT_STANDARD_DEVIATION.PopulationStd.VarianceDis 24
CT_STANDARD_DEVIATION.PopulationStd.StandardDeviationDis 32
CT_STANDARD_DEVIATION.SampleStd.MaxDis 40
CT_STANDARD_DEVIATION.SampleStd.MinDis 48
CT_STANDARD_DEVIATION.Sample.VarianceDis 56
CT_STANDARD_DEVIATION.Sample.StandardDeviationDis 64

Size_Of_CT_STANDARD_DEVIATION 72
```

Members

Mean

The mean is the average of the calculated timings. In nanoseconds

PopulationStd.Max

Computes the Maximum value of a population standard deviation. In nanoseconds

PopulationStd.Min

Computes the Minimum value of a population standard deviation. In nanoseconds

PopulationStd.Variance

Computes the Variance of a population standard deviation. In nanoseconds

PopulationStd.StandardDeviation

Computes the Standard Deviation of a entire population of the measured the timings. In nanoseconds

SampleStd.Max

Computes the Maximum value of a sample standard deviation. In nanoseconds

SampleStd.Min

Computes the Minimum value of a sample standard deviation. In nanoseconds

Sample.Variance

Computes the Variance of a sample standard deviation. In nanoseconds

Sample.StandardDeviation

Computes the Standard Deviation of a sample of the measured the timings. In nanoseconds

Remarks

The values on each member are expressed in nanoseconds. Also, the length of the data is a 64 byte value represented as a double Float type (Real).

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h

Examples of Usage:

See [CreateTimeProfile](#) function

See also

[CreateTimeProfile](#)

[Apendix1 - Understanding Mean, Variance and Standard Deviation](#)

[Apendix2 - Standard Deviation and Variance](#)

CT_STDEx structure

Displays extended information of Standard Deviation values measured from [CreateTimeProfileEx](#) function. The values are displayed in nanoseconds.

This structure is formed by a double data type containing the fastest time measured from [CreateTimeProfileEx](#) function, a dword value containing the Algo method Id where the fastest time was retrieved, and two [CT_STANDARD_DEVIATION](#) structures (*one where the fastest Mean was found, and other where the Fastest Time was found*).

The Standard Deviation is a measure of how spread out numbers are. Its symbol is σ (the Greek letter sigma)

When only a sample of data from a population is available, the term standard deviation of the sample or sample standard deviation can refer to either the above-mentioned quantity as applied to those data or to a modified quantity that is a better estimate of the population standard deviation (the standard deviation of the entire population).

By using standard deviations, a minimum and maximum value can be calculated that the averaged weight will be within some very high percentage of the time (99.9% or more).

Syntax

C++

a) Simplified

```
typedef struct _CT_STDEx {
    double          BestTiming;
    uint            IDFound;
    CT_STANDARD_DEVIATION Avg;
    CT_STANDARD_DEVIATION Fast;
} CT_STDEx, *PCT_STDEx
```

b) Complete

```
typedef struct _CT_STDEx {
    double          BestTiming;
    uint            IDFound;
    double          Avg.Mean;
    double          Avg.PopulationStd.Max;
    double          Avg.PopulationStd.Min;
    double          Avg.PopulationStd.Variance;
    double          Avg.PopulationStd.StandardDeviation;
    double          Avg.SampleStd.Max;
    double          Avg.SampleStd.Min;
    double          Avg.Sample.Variance;
    double          Avg.Sample.StandardDeviation;
```

CodeTune Reference Guide

```
double    Fast.Mean;  
double    Fast.PopulationStd.Max;  
double    Fast.PopulationStd.Min;  
double    Fast.PopulationStd.Variance;  
double    Fast.PopulationStd.StandardDeviation;  
double    Fast.SampleStd.Max;  
double    Fast.SampleStd.Min;  
double    Fast.Sample.Variance;  
double    Fast.Sample.StandardDeviation;  
} CT_STDEx, *PCT_STDEx
```

RosAsm

```
[CT_STDEx:  
CT_STDEx.BestTiming: R$ 0  
CT_STDEx.IDFound: D$ 0  
CT_STDEx.Avg.AlgoMethod: D$ 0  
CT_STDEx.Avg.Mean: R$ 0  
CT_STDEx.Avg.PopulationStd.Max: R$ 0  
CT_STDEx.Avg.PopulationStd.Min: R$ 0  
CT_STDEx.Avg.PopulationStd.Variance: R$ 0  
CT_STDEx.Avg.PopulationStd.StandardDeviation: R$ 0  
CT_STDEx.Avg.SampleStd.Max: R$ 0  
CT_STDEx.Avg.SampleStd.Min: R$ 0  
CT_STDEx.Avg.Sample.Variance: R$ 0  
CT_STDEx.Avg.Sample.StandardDeviation: R$ 0  
CT_STDEx.Fast.AlgoMethod: D$ 0  
CT_STDEx.Fast.Mean: R$ 0  
CT_STDEx.Fast.PopulationStd.Max: R$ 0  
CT_STDEx.Fast.PopulationStd.Min: R$ 0  
CT_STDEx.Fast.PopulationStd.Variance: R$ 0  
CT_STDEx.Fast.PopulationStd.StandardDeviation: R$ 0  
CT_STDEx.Fast.SampleStd.Max: R$ 0  
CT_STDEx.Fast.SampleStd.Min: R$ 0  
CT_STDEx.Fast.Sample.Variance: R$ 0  
CT_STDEx.Fast.Sample.StandardDeviation: R$ 0]
```

Masm

a) Simplified

```
CT_STDEx struct  
    BestTiming REAL8 ?  
    IDFound DWORD ?  
    Avg CT_STANDARD_DEVIATION <?>  
    Fast CT_STANDARD_DEVIATION <?>  
CT_STDEx ends
```

b) Complete

```
CT_STDEx struct  
    BestTiming REAL8 ?
```

CodeTune Reference Guide

```
IDFound DWORD ?  
Avg_Mean REAL8 ?  
Avg_PopulationStd_Max REAL8 ?  
Avg_PopulationStd_Min REAL8 ?  
Avg_PopulationStd_Variance REAL8 ?  
Avg_PopulationStd_StandardDeviation REAL8 ?  
Avg_SampleStd_Max REAL8 ?  
Avg_SampleStd_Min REAL8 ?  
Avg_Sample_Variance REAL8 ?  
Avg_Sample_StandardDeviation REAL8 ?  
Fast_Mean REAL8 ?  
Fast_PopulationStd_Max REAL8 ?  
Fast_PopulationStd_Min REAL8 ?  
Fast_PopulationStd_Variance REAL8 ?  
Fast_PopulationStd_StandardDeviation REAL8 ?  
Fast_SampleStd_Max REAL8 ?  
Fast_SampleStd_Min REAL8 ?  
Fast_Sample_Variance REAL8 ?  
Fast_Sample_StandardDeviation REAL8 ?  
CT_STDEx ends
```

Structure Displacement

(Equates related to the position of each member and size of the structure)

```
CT_STDEx.BestTimingDis 0  
CT_STDEx.IDFoundDis 8  
CT_STDEx.Avg.AlgoMethodDis 12  
CT_STDEx.Avg.MeanDis 16  
CT_STDEx.Avg.PopulationStd.MaxDis 24  
CT_STDEx.Avg.PopulationStd.MinDis 32  
CT_STDEx.Avg.PopulationStd.VarianceDis 40  
CT_STDEx.Avg.PopulationStd.StandardDeviationDis 48  
CT_STDEx.Avg.SampleStd.MaxDis 56  
CT_STDEx.Avg.SampleStd.MinDis 64  
CT_STDEx.Avg.Sample.VarianceDis 72  
CT_STDEx.Avg.Sample.StandardDeviationDis 80  
CT_STDEx.Fast.AlgoMethodDis 88  
CT_STDEx.Fast.MeanDis 92  
CT_STDEx.Fast.PopulationStd.MaxDis 100  
CT_STDEx.Fast.PopulationStd.MinDis 108  
CT_STDEx.Fast.PopulationStd.VarianceDis 116  
CT_STDEx.Fast.PopulationStd.StandardDeviationDis 124  
CT_STDEx.Fast.SampleStd.MaxDis 132  
CT_STDEx.Fast.SampleStd.MinDis 140  
CT_STDEx.Fast.Sample.VarianceDis 148  
CT_STDEx.Fast.Sample.StandardDeviationDis 156  
  
Size_Of_CT_STDEx 164
```

Members

BestTiming

CodeTune Reference Guide

The value of the fastest timing collected from the function [CreateTimeProfileEx](#). It is the one that best represents the real timing from which the user's code was running. It is the minimum value measured through all available algo methods. In nanoseconds

IDFound

The algo method ID from where the value of the BestTiming member was retrieved on function [CreateTimeProfileEx](#).

The values are the same ones as in parameter "algoMethod" in [CreateTimeProfile](#) function.

It can be one of the following equates.

Name	Value	Description
CT_ALGO1	1	This method uses cpuid + rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) cpuid rdtsc (...)
CT_ALGO2	2	This method uses lfence + rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtsc (...)
CT_ALGO3	3	This method uses lfence + rdtsc + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtsc lfence (...)
CT_ALGO4	4	This method uses rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) rdtscp (...)
CT_ALGO5	5	This method uses lfence + rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtscp

CodeTune Reference Guide

		(...)
CT_ALGO6	6	This method uses cpuid + rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) cpuid rdtscp (...)
CT_ALGO7	7	This method uses rdtscp + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) rdtscp lfence (...)
CT_ALGO8	8	This method uses rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) rdtsc (...)

Avg

A [CT_STANDARD_DEVIATION](#) structure containing the values of the fastest Mean value found among all Algo Methods available. This structure is collected after all methods are tested. Once [CreateTmeprofileEx](#) finishes the initial analysis it selects the structure with the fastest mean value found.

For more information about the members of this structure, see [CT_STANDARD_DEVIATION](#).

Fast

A [CT_STANDARD_DEVIATION](#) structure containing the values of the fastest value ever found among all Algo Methods available. This structure is collected after all methods are tested. Once [CreateTmeprofileEx](#) finishes the initial analysis it selects the structure with the fastest value found.

The fastest value, in general is found in the SampleStd.Min member of [CT_STANDARD_DEVIATION](#).structure.

But, not necessarily the value of Avg and Fast were found under the same Algo Method. [CreateTmeprofileEx](#) function may have found the fastest mean from Algo method 3 and the Fastest value (*The one with the smallest STD.Min*) in Algo Method 1.

This happens because of the variances that are being found. One Algo Method can have a Variance (or Standard Deviation) bigger then the other. The mean, is like the name says an average o the results, and depends the values found in STD.Max and STD.Min. So if a value have a low STDMin and high STDMax, the value of the Mean tends to be bigger too.

Since what we are collecting is a measure of the time of the samples, it is safe to say that the value found on this “Fast” member is the one that is more close to the real time of the tested function.

For more information about the members of this structure, see [CT_STANDARD_DEVIATION](#).

Remarks

The values on each member are expressed in nanoseconds. Also, the length of the data is a 64 byte value represented as a double Float type (Real).

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h

Examples of Usage:

See [CreateTimeProfileEx](#) function

See also

[CreateTimeProfile](#)

[CreateTimeProfileEx](#)

[CT_STANDARD_DEVIATION](#)

[Apendix1 - Understanding Mean, Variance and Standard Deviation](#)

[Apendix2 - Standard Deviation and Variance](#)

CT_Nfo structure

Displays information about the Algorithm method used, amount of overheads and Good Samples that are being retrieved from [CreateTimeProfile](#) function.

Syntax

C++

```
typedef struct _CT_Nfo {  
    DWORD   AlgoID;  
    DWORD   Overheads;  
    DWORD   GoodSamples;  
} CT_Nfo, *PCT_Nfo
```

RosAsm

```
[CT_Nfo:  
    CT_Nfo.AlgoID: D$ 0  
    CT_Nfo.Overheads: D$ 0  
    CT_Nfo.GoodSamples: D$ 0 ]
```

Masm

```
CT_Nfo struct  
    AlgoID DWORD ?  
    Overheads DWORD ?  
    GoodSamples DWORD ?  
CT_Nfo ends
```

Structure Displacement

(Equates related to the position of each member and size of the structure)

```
CT_Nfo.AlgoIDDis 0  
CT_Nfo.OverheadsDis 4  
CT_Nfo.GoodSamplesDis 8  
  
Size_Of_CT_Nfo 12
```

Members

AlgoId

A flag corresponding to the Algo Method chosen by the user. For a complete description of the Flags, please see [CreateTimeProfile](#) function.

CodeTune Reference Guide

The supported flags are:

Name	Value
CT_ALGO1	1
CT_ALGO2	2
CT_ALGO3	3
CT_ALGO4	4
CT_ALGO5	5
CT_ALGO6	6
CT_ALGO7	7
CT_ALGO8	8

Overheads

The total amount of overheads and problems found on [CreateTimeProfile](#) function. The amount of overheads are calculated on both stages: Calibration and benchmarking. So, while it is calibrating, this member is storing the amount of overheads for the user display it if he wants. Once calibration is done, this member is zeroed for the next stage (Benchmark). So, when Benchmark begins, it starts calculating the amount of overheads on this stage too.

GoodSamples

The total amount of Good samples found on [CreateTimeProfile](#) function. The amount of Good Samples is calculated on both stages: Calibration and benchmarking. So, while it is calibrating, this member is storing the amount of samples considered “good” for the user display it if he wants. Once calibration is done, this member is zeroed for the next stage (Benchmark). So, when Benchmark begins, it starts calculating the amount of good samples on this stage too.

Remarks

none

Requirements

Minimum supported client Windows2000

Header CodeTune.h

Examples of Usage:

See [CreateTimeProfile](#) function

See also

[CreateTimeProfile](#)
[RunTimeDataProc](#)

CodeTune Reference Guide

III- EQUATES

Here is a list of equates used by CodeTune Api

Name	Value	Description
CPU_CPUID_AVALIABLE	1	The CPU supports cpuid instruction
CPU_RDTSC_AVALIABLE	4	The CPU supports rdtsc instruction
CPU_RDTSCP_AVALIABLE	2	The CPU supports rdtscp instruction
CT_ALGO1	1	This method uses cpuid + rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) cpuid rdtsc (...)
CT_ALGO2	2	This method uses lfence + rdtsc opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtsc (...)
CT_ALGO3	3	This method uses lfence + rdtsc + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtsc lfence (...)
CT_ALGO4	4	This method uses rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) rdtscp (...)
CT_ALGO5	5	This method uses lfence + rdtscp opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures: (...) lfence rdtscp (...)
CT_ALGO6	6	This method uses cpuid + rdtscp opcode to calculate the total amount of ticks and the

CodeTune Reference Guide

		<p>amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) cpuid rdtscp (...)</pre>
CT_ALGO7	7	<p>This method uses rdtscp + lfence opcode to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) rdtscp lfence (...)</pre>
CT_ALGO8	8	<p>This method uses QueryPerformanceCounter Windows Api to calculate the total amount of ticks and the amount of time it takes to run. A pseudo code involving this method is basically this while starting and ending the time measures:</p> <pre>(...) invoke QueryPerformanceCounter, ADDR pcCount (...)</pre>
CT_ALGO_METHOD_ERROR	0-8	Invalid algo method. The chosen algorithm doesn't exists on the user CPU
CT_ANALYSIS_ERROR1	0-3	This happens when all samples collected in benchmark are smaller or equal to the calibration. So, the user code is too short to perform a full analysis, because the app can distinguish between both. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions
CT_ANALYSIS_ERROR2	0-5	Sample too short and only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions.
CT_ANALYSIS_ERROR3	0-6	Although the sample fits the required range, not all of them fit the required characteristics to be a good sample. The resultant Standard deviation has negative values.
CT_ANALYSIS_ERROR4	0-7	Happens on too short code or non existent. Not all of them fit the required characteristics to be a good sample. The resultant Standard deviation has negative values. This is a common error found on short user's code. Such as analysis of isolated instructions inside lpStartAddress, such as "xor eax eax", "mov ebx 5" etc. so, it happens if your function consists on short instructions

CodeTune Reference Guide

CT_ANALYSIS_START	7	Analysis Started
CT_ANALYSIS_SUCESS	8	Analysis Completed Successfully
CT_BENCHMARK_FINISHED	6	Benchmark Finished
CT_BENCHMARK_RUNNING	5	Benchmark is Running
CT_BENCHMARK_START	4	Benchmark Started
CT_CALIBRATION_FINISHED	3	Calibration Finished
CT_CALIBRATION_RUNNING	2	Calibration is Running
CT_CALIBRATION_START	1	Calibration Started
CT_ERROR_BENCHMARK_OVERHEAD	0-2	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during Benchmarking.
CT_ERROR_CALIBRATION_OVERHEAD	0-1	The maximum overhead limit (300000 as defined in OVERHEAD_LIMIT) was reached during calibration.
CT_ERROR_INPUT_VALUE	0	The user inputted an incorrect value for Samples or Iterations. The amount of each one of them are limited to 1 to 100000 as defined in MAX_SAMPLES and MAX_ITERATIONS
CT_INCONCLUSIVE	0-4	Benchmark time fits the required range to be considered a good sample, but only 1 sample was found to be compared. Cannot make a Standard Deviation measure with this.
CT_INSUFFICIENT_FEATURES	0-10	Your processor does not supports the minimum instructions to perform a timing measure.
CT_STATUSCODE_ERROR	0-9	The user did not inputted a pointer to the Status code
MAX_ITERATIONS	100000	The maximum amount of Iterations
MAX_SAMPLES	100000	The maximum amount of samples
OVERHEAD_LIMIT	300000	The maximum amount of overheads performed by CreateTimeProfile function. When this limit is reached the function exits unsuccessfully, to avoid hangs.

Requirements

Minimum supported client Windows2000
Header CodeTune.h
DLL CodeTune.dll

Examples of Usage:

See [CreateTimeProfile](#) , [CpuSettings](#) functions

See also

[CreateTimeProfile](#)
[CpuSettings](#)
[CpuData](#)

IV- TYPE DEFINITIONS

LP_RUNTIMEDATA_CALLBACK_ROUTINE Pointer

Function

Points to a function that notify the host when collecting data has started in [CreateTimeProfile](#) function.

Syntax

```
typedef DWORD (__stdcall * LP_RUNTIMEDATA_CALLBACK_ROUTINE) (
    _Out_ PCT_Nfo      pUserStruct,
    _Out_ INT         dwStatusCode,
);
```

Parameters

pUserStruct [out]

A pointer to a CT_Nfo structure that contains the values internally stored from [CreateTimeProfile](#) such as: overheads, the amount of good samples that are being collected while the main function is operating in the calibration mode and benchmark mode.

dwStatusCode [out]

A reference to a status code. Status codes indicate the success or failure of the function. It is a INT (32 bits: Dword) variable. For more information on the used values and equates, see: [CreateTimeProfile](#)

Return value

The function does not return any value

Remarks

The function to which LP_RUNTIMEDATA_CALLBACK_ROUTINE points is a callback function and must be implemented by the writer of the hosting application.

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
DLL	CodeTune.dll

Examples of Usage:

See [CreateTimeProfile](#) function

See also

[CreateTimeProfile](#)

[RunTimeDataProc](#)

LP_USERTARGET_CALLBACK_ROUTINE Function Pointer

Points to a function that notify the host when he must use the code inside this function to start timing it with [CreateTimeProfile](#) function.

Syntax

```
typedef DWORD (__stdcall * LP_USERTARGET_CALLBACK_ROUTINE) (
);
```

Parameters

None

Return value

The function does not return any value

Remarks

The function to which LP_USERTARGET_CALLBACK_ROUTINE point is a callback function and must be implemented by the writer of the hosting application.

This is the most important type definition used as parameter in [CreateTimeProfile](#) function. From here the actual timing is being measure, since this is where the user must insert his code to be analyzed.

Requirements

Minimum supported client	Windows2000
Header	CodeTune.h
DLL	CodeTune.dll

Examples of Usage:

See [CreateTimeProfile](#) function

See also

[CreateTimeProfile](#)
[RunTimeDataProc](#)

APENDIX I

Understanding Mean, Variance and Standard Deviation

1) How to Find the Mean

The mean is the **average** of the numbers.

It is easy to calculate: **add up** all the numbers, then **divide by how many** numbers there are.

In other words it is the **sum** divided by the **count**.

Example 1: What is the Mean of these numbers?

6, 11, 7

- Add the numbers: $6 + 11 + 7 = 24$
- Divide by *how many* numbers (there are 3 numbers): $24 / 3 = 8$

The Mean is 8

Why Does This Work?

It is because 6, 11 and 7 added together is the same as 3 lots of 8:



It is like you are "flattening out" the numbers

Example 2: Look at these numbers:

3, 7, 5, 13, 20, 23, 39, 23, 40, 23, 14, 12, 56, 23, 29

The sum of these numbers is 330

There are fifteen numbers.

The mean is equal to $330 / 15 = 22$

The mean of the above numbers is 22

Negative Numbers

How do you handle negative numbers? Adding a negative number is the same as subtracting the number (without the negative). For example $3 + (-2) = 3 - 2 = 1$.

Knowing this, let us try an example:

Example 3: Find the mean of these numbers:

$$3, -7, 5, 13, -2$$

- The sum of these numbers is $3 - 7 + 5 + 13 - 2 = 12$
- There are **5** numbers.
- The mean is equal to $12 \div 5 = 2.4$

The mean of the above numbers is 2.4

Here is how to do it one line:

$$\text{Mean} = \frac{3 - 7 + 5 + 13 - 2}{5} = \frac{12}{5} = 2.4$$

2) Standard Deviation and Variance

Deviation just means how far from the normal

Standard Deviation

The Standard Deviation is a measure of how spreads out numbers are. Its symbol is σ (the Greek letter sigma)

The formula is easy: it is the **square root** of the **Variance**. So now you ask, "What is the Variance?"

Variance

The Variance is defined as:

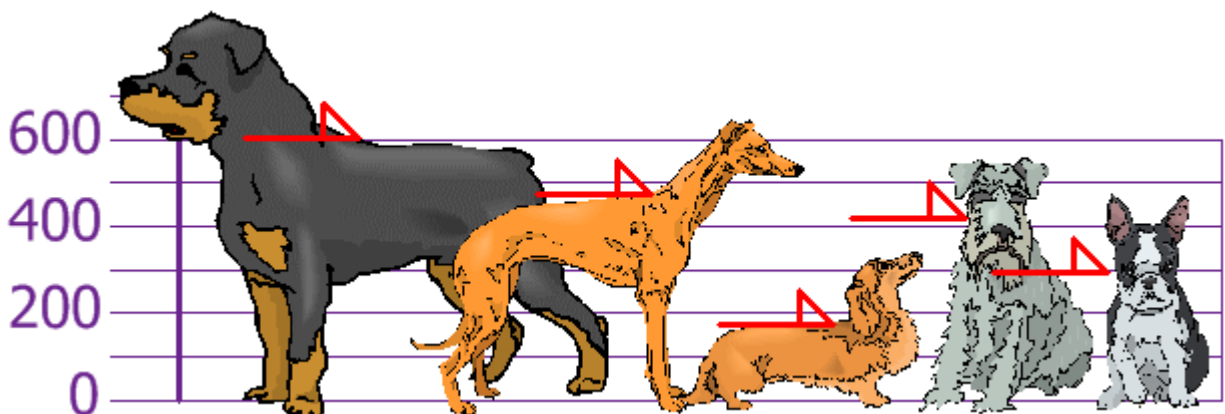
The average of the **squared** differences from the Mean.

To calculate the variance follow these steps:

- Work out the [Mean](#) (the simple average of the numbers)
- Then for each number: subtract the Mean and square the result (the *squared difference*).
- Then work out the average of those squared differences. ([Why Square?](#))

Example

You and your friends have just measured the heights of your dogs (in millimeters):



The heights (at the shoulders) are: 600mm, 470mm, 170mm, 430mm and 300mm.

Find out the Mean, the Variance, and the Standard Deviation.

Your first step is to find the Mean:

Answer:

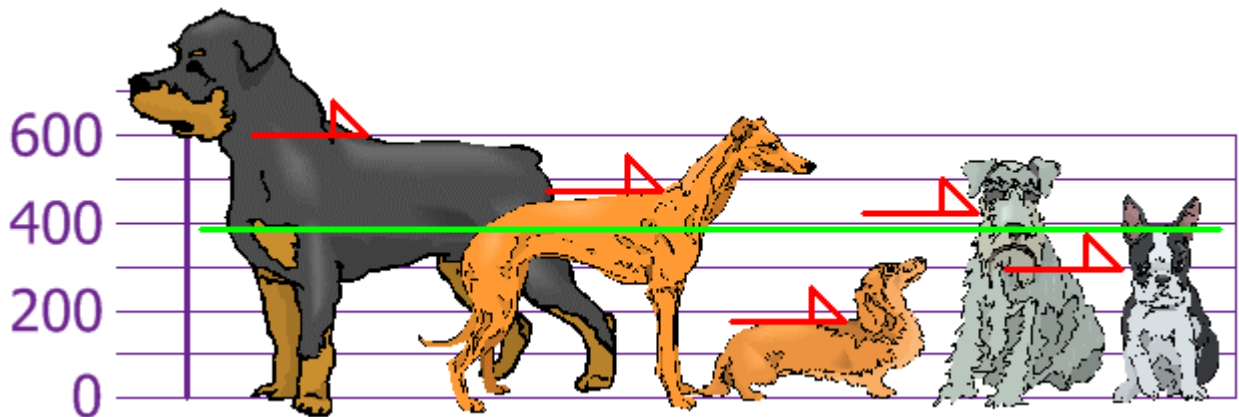
$$\text{Mean} = 600 + 470 + 170 + 430 + 300 = 1970 \div 5 = 394$$

CodeTune Reference Guide

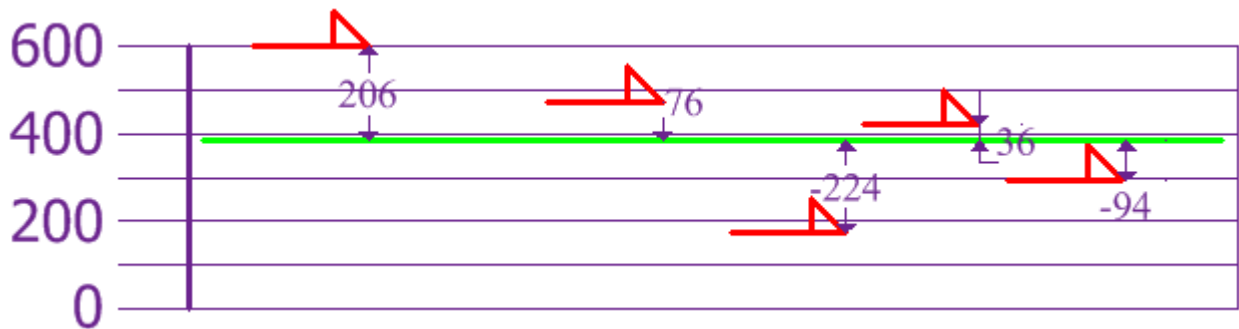
5

5

So the mean (average) height is 394 mm. Let's plot this on the chart:



Now we calculate each dog's difference from the Mean:



To calculate the Variance, take each difference, square it, and then average the result:

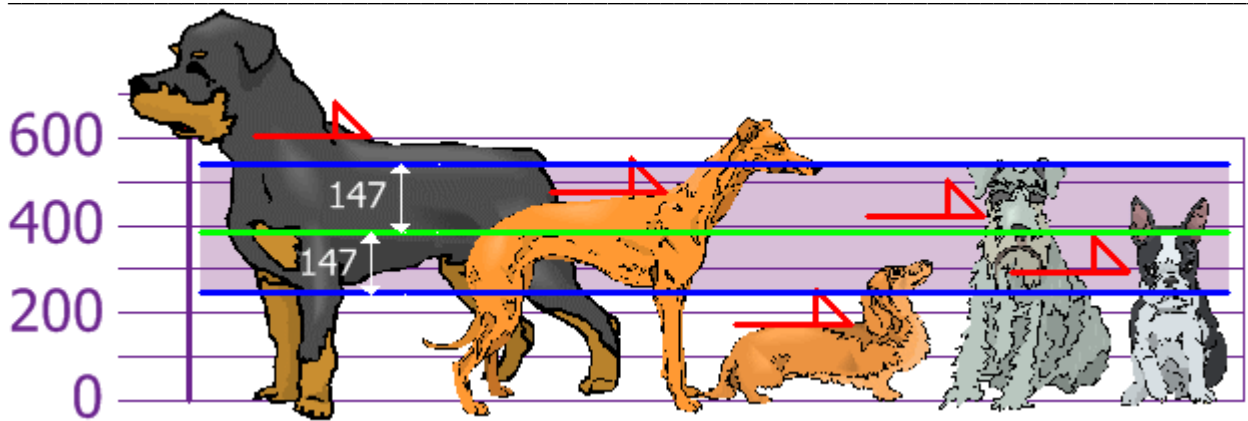
$$\begin{aligned}
 \text{Variance: } \sigma^2 &= \frac{206^2 + 76^2 + (-224)^2 + 36^2 + (-94)^2}{5} \\
 &= \frac{42,436 + 5,776 + 50,176 + 1,296 + 8,836}{5} \\
 &= \frac{108,520}{5} = 21,704
 \end{aligned}$$

So, the Variance is **21,704**.

And the Standard Deviation is just the square root of Variance, so:

$$\text{Standard Deviation: } \sigma = \sqrt{21,704} = 147.32... = 147 \text{ (to the nearest mm)}$$

And the good thing about the Standard Deviation is that it is useful. Now we can show which heights are within one Standard Deviation (147mm) of the Mean:



So, using the Standard Deviation we have a "standard" way of knowing what is normal, and what is extra large or extra small.

Rottweilers **are** tall dogs. And Dachshunds **are** a bit short ... but don't tell them!

But ... there is a small change with Sample Data

Our example was for a **Population** (the 5 dogs were the only dogs we were interested in).

But if the data is a **Sample** (a selection taken from a bigger Population), then the calculation changes!

When you have "N" data values that are:

- **The Population:** divide by **N** when calculating Variance (like we did)
- **A Sample:** divide by **N-1** when calculating Variance

All other calculations stay the same, including how we calculated the mean.

Example: if our 5 dogs were just a **sample** of a bigger population of dogs, we would divide by **4** instead of 5 like this:

Sample Variance = $108,520 / 4 = 27,130$

Sample Standard Deviation = $\sqrt{27,130} = 164$ (to the nearest mm)

Think of it as a "correction" when your data is only a sample.

Formulas

Here are the two formulas, explained at [Standard Deviation Formulas](#) if you want to know more:

The "**Population** Standard Deviation":
$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

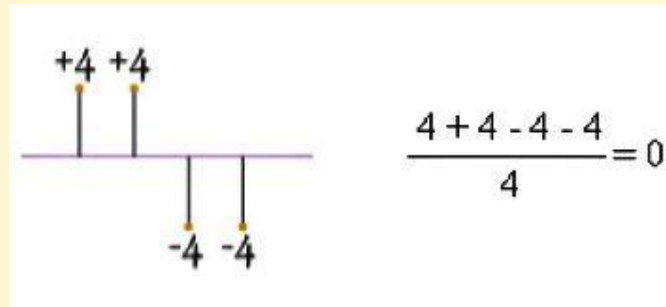
The "**Sample** Standard Deviation":
$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$$

CodeTune Reference Guide

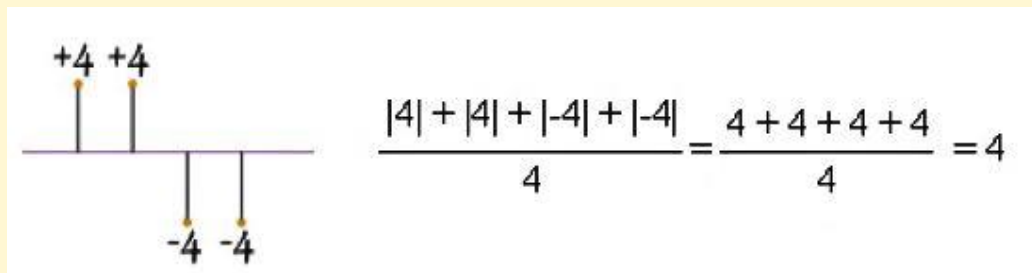
Looks complicated, but the important change is to divide by $N-1$ (instead of N) when calculating a Sample Variance.

*Footnote: Why *square* the differences?

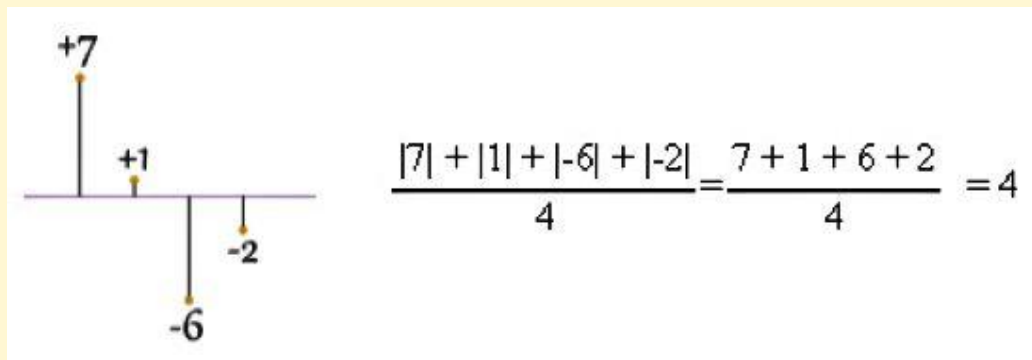
If we just added up the differences from the mean ... the negatives would cancel the positives:



So that won't work. How about we use absolute values?

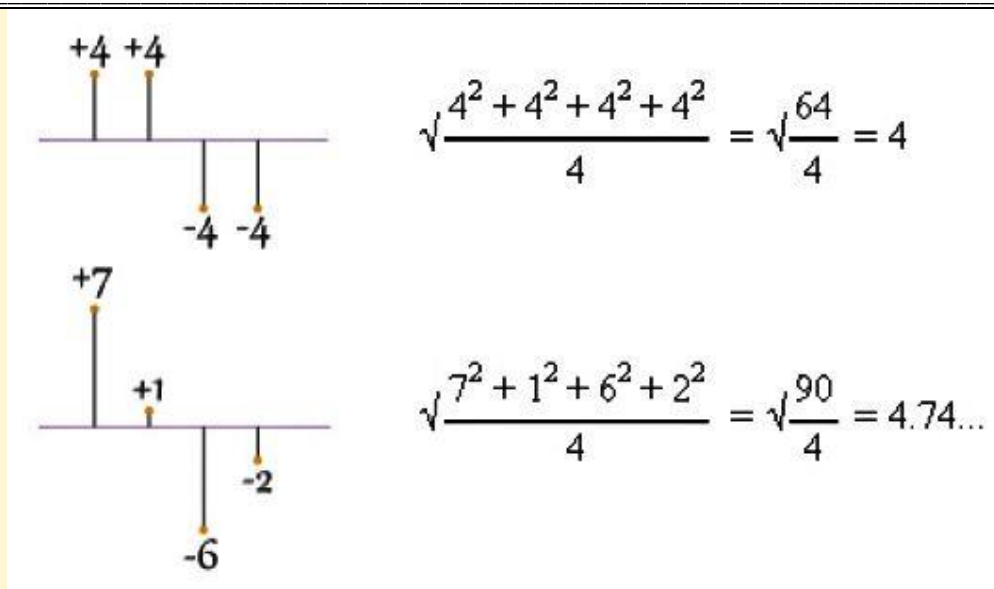


That looks good (and is the [Mean Deviation](#)), but what about this case:



Oh No! It also gives a value of 4, Even though the differences are more spread out! So let us try squaring each difference (and taking the square root at the end):

CodeTune Reference Guide



That is nice! The Standard Deviation is bigger when the differences are more spread out ... just what we want!

In fact this method is a similar idea to distance between points, just applied in a different way.

And it is easier to use algebra on squares and square roots than absolute values, which makes the standard deviation easy to use in other areas of mathematics.

3) Standard Deviation Formulas

Deviation just means how far from the normal

Standard Deviation

The Standard Deviation is a measure of **how spread out numbers are**.

Here we explain **the formulas**.

The symbol for Standard Deviation is σ (the Greek letter sigma). This is the formula for Standard Deviation:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

Say what? Please explain!

OK. Let us explain it step by step.

Say we have a bunch of numbers like 9, 2, 5, 4, 12, 7, 8, 11.

To calculate the standard deviation of those numbers:

1. Work out the [Mean](#) (the simple average of the numbers)
2. Then for each number: subtract the Mean and square the result
3. Then work out the mean of **those** squared differences.
4. Take the square root of that and we are done!

The formula actually says all of that, and I will show you how.

The Formula Explained

First, let us have some example values to work on:

Example: Sam has 20 Rose Bushes.

The number of flowers on each bush is

9, 2, 5, 4, 12, 7, 8, 11, 9, 3, 7, 4, 12, 5, 4, 10, 9, 6, 9, 4

Work out the Standard Deviation.

Step 1. Work out the mean

In the formula above μ (the greek letter "mu") is the [mean](#) of all our values ...

Example: 9, 2, 5, 4, 12, 7, 8, 11, 9, 3, 7, 4, 12, 5, 4, 10, 9, 6, 9, 4

CodeTune Reference Guide

The mean is:

$$9+2+5+4+12+7+8+11+9+3+7+4+12+5+4+10+9+6+9+4 \ 20 = 140 / 20 = 7$$

So:

$$\mu = 7$$

Step 2. Then for each number: subtract the Mean and square the result

This is the part of the formula that says:

$$(x_i - \mu)^2$$

So what is x_i ? They are the individual x values 9, 2, 5, 4, 12, 7, etc...

In other words $x_1 = 9$, $x_2 = 2$, $x_3 = 5$, etc.

So it says "for each value, subtract the mean and square the result", like this

Example (continued):

$$\begin{aligned}(9 - 7)^2 &= (2)^2 = 4 \\(2 - 7)^2 &= (-5)^2 = 25 \\(5 - 7)^2 &= (-2)^2 = 4 \\(4 - 7)^2 &= (-3)^2 = 9 \\(12 - 7)^2 &= (5)^2 = 25 \\(7 - 7)^2 &= (0)^2 = 0 \\(8 - 7)^2 &= (1)^2 = 1 \\&\dots \text{ etc } \dots\end{aligned}$$

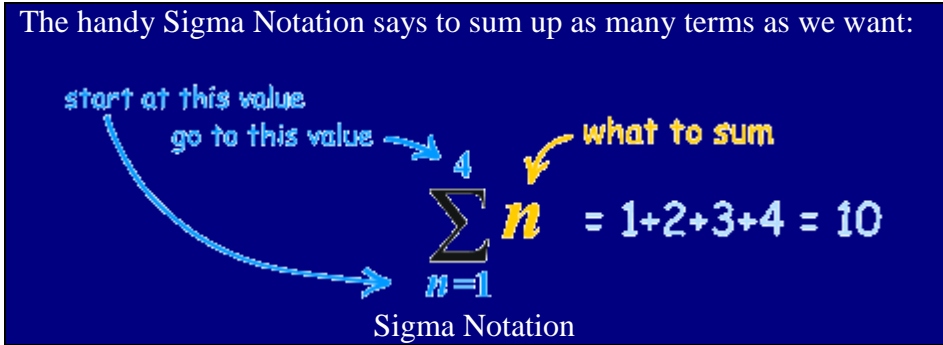
Step 3. Then work out the mean of those squared differences.

To work out the mean, **add up all the values** then **divide by how many**.

First add up all the values from the previous step.

But how do we say "add them all up" in mathematics? We use "Sigma": Σ

The handy Sigma Notation says to sum up as many terms as we want:



The diagram shows the formula $\sum_{n=1}^4 n = 1+2+3+4 = 10$ on a dark blue background. Annotations include: a blue arrow pointing to the '4' above the sigma symbol labeled 'go to this value'; a blue arrow pointing to the 'n=1' below the sigma symbol labeled 'start at this value'; and a yellow arrow pointing to the 'n' labeled 'what to sum'. The text 'Sigma Notation' is written below the formula.

We want to add up all the values from 1 to N, where $N=20$ in our case because there are 20 values:

Example (continued):

CodeTune Reference Guide

$$\sum_{i=1}^N (x_i - \mu)^2$$

Which means: Sum all values from $(x_1-7)^2$ to $(x_N-7)^2$

We already calculated $(x_1-7)^2=4$ etc. in the previous step, so just sum them up:=
 $4+25+4+9+25+0+1+16+4+16+0+9+25+4+9+9+4+1+4+9 = 178$

But that isn't the mean yet, we need to **divide by how many**, which is simply done by multiplying by "1/N":

Example (continued):

$$\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2$$

Mean of squared differences = $(1/20) \times 178 = 8.9$
(Note: this value is called the "Variance")

Step 4. Take the square root of that:

Example (concluded):

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

$\sigma = \sqrt{(8.9)} = 2.983...$

DONE!

Sample Standard Deviation

But wait, there is more ...

... sometimes our data is only a **sample** of the whole population.

Example: Sam has 20 rose bushes, but only counted the flowers on 6 of them!

The "population" is all 20 rose bushes, and the "sample" is the 6 that were counted. Let us say they are:

9, 2, 5, 4, 12, 7

We can still **estimate** the Standard Deviation.

But when we use the sample as an **estimate of the whole population**, the Standard Deviation formula changes to this:

The formula for **Sample Standard Deviation**:

CodeTune Reference Guide

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$$

The important change is "**N-1**" instead of "**N**" (which is called "Bessel's correction").

The symbols also change to reflect that we are working on a sample instead of the whole population:

- The mean is now \bar{x} (for sample mean) instead of μ (the population mean),
- And the answer is s (for Sample Standard Deviation) instead of σ .

But that does not affect the calculations. **Only N-1 instead of N changes the calculations.**

OK, let us now calculate the **Sample Standard Deviation**:

Step 1. Work out the mean

Example 2: Using sampled values 9, 2, 5, 4, 12, 7

The mean is $(9+2+5+4+12+7) / 6 = 39/6 = 6.5$

So:

$$\bar{x} = 6.5$$

Step 2. Then for each number: subtract the Mean and square the result

Example 2 (continued):

$$(9 - 6.5)^2 = (2.5)^2 = 6.25$$

$$(2 - 6.5)^2 = (-4.5)^2 = 20.25$$

$$(5 - 6.5)^2 = (-1.5)^2 = 2.25$$

$$(4 - 6.5)^2 = (-2.5)^2 = 6.25$$

$$(12 - 6.5)^2 = (5.5)^2 = 30.25$$

$$(7 - 6.5)^2 = (0.5)^2 = 0.25$$

Step 3. Then work out the mean of those squared differences.

To work out the mean, **add up all the values** then **divide by how many**.

But hang on ... we are calculating the **Sample** Standard Deviation, so instead of dividing by how many (N), we will divide by **N-1**

Example 2 (continued):

$$\text{Sum} = 6.25 + 20.25 + 2.25 + 6.25 + 30.25 + 0.25 = \mathbf{65.5}$$

$$\text{Divide by N-1: } (1/5) \times 65.5 = \mathbf{13.1}$$

(This value is called the "Sample Variance")

Step 4. Take the square root of that:

Example 2 (concluded):

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$$

$s = \sqrt{13.1} = 3.619...$

DONE!

Comparing

When we used the whole **population** we got: Mean = 7, Standard Deviation = 2.983...

When we used the **sample** we got: Sample Mean = 6.5, Sample Standard Deviation = 3.619...

Our Sample Mean was wrong by 7%, and our Sample Standard Deviation was wrong by 21%.

Why Would We Take a Sample?

Mostly because it is easier and cheaper.

Imagine you want to know what the whole country thinks ... you can't ask millions of people, so instead you ask maybe 1,000 people.

There is a nice quote (supposed to be by Samuel Johnson):

"You don't have to eat the whole ox to know that the meat is tough."

This is the essential idea of sampling. To find out information about the population (such as mean and standard deviation), we do not need to look at **all** members of the population; we only need a sample.

But when we take a sample, we lose some accuracy.

Summary

The **Population** Standard Deviation: $\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$

The **Sample** Standard Deviation: $s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2}$

APENDIX II

Standard Deviation and Variance

The variance and the closely-related standard deviation are measures of how spread out a distribution is. In other words, they are measures of variability.

The variance is computed as the average squared deviation of each number from its mean. For example, for the numbers 1, 2, and 3, the mean is 2 and the variance is:

$$\sigma^2 = \frac{(1-2)^2 + (2-2)^2 + (3-2)^2}{3} = 0.667 \cdot$$

The formula (in summation notation) for the variance in a population is

$$\sigma^2 = \frac{\sum (X - \mu)^2}{N}$$

where μ is the mean and N is the number of scores.

When the variance is computed in a sample, the statistic

$$S^2 = \frac{\sum (X - M)^2}{N}$$

(where M is the mean of the sample) can be used. S^2 is a biased estimate of σ^2 , however. By far the most common formula for computing variance in a sample is:

$$s^2 = \frac{\sum (X - M)^2}{N - 1}$$

which gives an unbiased estimate of σ^2 . Since samples are usually used to estimate parameters, s^2 is the most commonly used measure of variance. Calculating the variance is an important part of many statistical applications and analyses. It is the first step in calculating the standard deviation.

Standard Deviation

The standard deviation formula is very simple: it is the square root of the variance. It is the most commonly used measure of spread.

An important attribute of the standard deviation as a measure of spread is that if the mean and standard deviation of a normal distribution are known, it is possible to compute the percentile rank associated with any given score. In a normal distribution, about 68% of the scores are within one standard deviation of the mean and about 95% of the scores are within two standard deviations of the mean.

The standard deviation has proven to be an extremely useful measure of spread in part because it is mathematically tractable. Many formulas in statistics use the standard deviation.