

Document of ClassLib OscilloscopeKernel

Index

- [Foreword](#)
- [OscilloscopeKernel](#)
- [SingleThreadOscilloscope](#)
- [SimpleOscilloscope](#)
- [TimeCountedOscilloscope](#)
- [MultiThreadOscilloscope](#)
- [UndrivedOscilloscope](#)
- [DrivedOscilloscope](#)
- [Wave](#)

Foreword

- if the method or attribute of a certain class that behave the same as the super-class or behave just as the implemented interface requires, it will still be listed again in the document of this certain class, but no details except a `see also`.
- `private` attribute, field, or method will not be listed. `protected` attribute and method will be special marked at the class's attribute-list or method-list. So, the attributes and methods that are listed without special mark are all `public`.
- `protected` and `public` has no difference when it comes to the constructor of a abstract class, so `protected` will not be special marked on this occasion.
- static method and static attribute will be special marked, except the methods and attributes of a static class.
- the time unit is defined with [Waves.UNIT_NUMBER_PRO_SECOND](#). the defaute time unit is μs but most of System functions use ms as the time unit, be careful!.
- for attributes-list of classes or interfaces, in accessor row:

symbol	meaning
G	only has a public Getter
S	only has a public Setter
g	only has a protected Getter
s	only has a protected Setter
GS	has both public Getter and public Setter
gS	has protected Getter and public Setter
Gs	has public Getter and protected Setter
readonly	is a readonly field (if a field is not readonly, this classlib make sure it is private)

- parameter's type, but name, will not provided in the method-list in a class or interface. if you need the name of parameters click the method name or scroll down to see the details of this method.

OscilloscopeKernel

1 | namespace OscilloscopeKernel

Summary:

- main part of oscilloscope-simulation.
- developed in .NET Standard.

type	name	description
abstract class	SingleThreadOscilloscope	an abstract class that describe an oscilloscope that cannot start a new draw-task while the old one has not finish
class	SimpleOscilloscope	a SingleThreadOscilloscope with public Draw() .
class	TimeCountedOscilloscope	a SingleThreadOscilloscope with public Draw() and a built-in watch, which means it doesn't need to delta_time as input.
abstract class	MultiThreadOscilloscope	an abstract class that describe an oscilloscope that can start a new draw-task while the old one has not finish
class	UndrivedOscilloscope	a MultiThreadOscilloscope with public Draw() .
class	DrivedOscilloscope	a MultiThreadOscilloscope that can produce graphs periodically.
namespace	Wave	tools to describe electric waves with time and voltage.
namespace	Tools	
namespace	Drawing	
namespace	Producer	
namespace	Exceptions	

SingleThreadOscilloscope

```
1 public abstract class SingleThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: none
- interfaces: none
- summary:
 - an oscilloscope that cannot start a new draw-task while the old one has not finished.
 - T is the output type of this oscilloscope.
- remarks
 - this is a abstract class, if you want to use it, please try [SimpleOscilloscope](#) or [TimeCountedOscilloscope](#).
 - calling [Draw\(\)](#) to produce and get a new graph.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- constructors:

name	description
SingleThreadOscilloscope (ICanvas <T>, IPointDrawer , IGraphProducer , IControlPanel)	

- methods:

name	description
protected T Draw (double)	produce and get a new graph.

constructors:

```
1 protected SingleThreadOscilloscope(  
2     ICanvas<T> canvas,  
3     IPointDrawer point_drawer,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel)
```

- Summary:
 - create a new oscilloscope.
- Remarks:
 - every input objects should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:
 - [ICanvas](#)<T> canvas: the canvas that produce the graph.
 - [IPointDrawer](#) point_drawer: the point-drawer the producer will produce the graph with.
 - [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.

- [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
 - ConcurrentQueue<T> buffer: the buffer of this oscilloscope, if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
 - Normal-Behaviour:
 - Pre-Condition:
 - canvas.GraphSize == point_drawer.GraphSize
 - !graph_producer.RequireConcurrentDrawer || point_drawer.IsConcurrent
 - Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas.GraphSize != point_drawer.GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent
-

methods:

```
1 | protected T Draw(double delta_time);
```

- Summary:
 - get the current state of the panel and produce a new graph according to this. then return the graph while finish.
 - Params:
 - double delta_time: the time during which the point will be drawn on the graph. in short you'd better delivery the time span from the latest call of this method. it should not be negative.
 - Normal-Behaviour:
 - Pre-condition:
 - delta_time >= 0.
 - Post-Condition:
 - a new graph with type T will be produced and return.
-

SimpleOscilloscope

```
1 public class SimpleOscilloscope<T> : SingleThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: [SingleThreadOscilloscope](#)
- interfaces: none
- summary:
 - the only difference with [SingleThreadOscilloscope](#) is the method [Draw\(\)](#) is public.
- constructors:

name	description
SingleThreadOscilloscope (ICanvas <T>, IPointDrawer , IGraphProducer , IControlPanel)	

- methods:

name	description
protected T Draw (double)	produce and get a new graph.

constructors:

```
1 protected SimpleOscilloscope(  
2     ICanvas<T> canvas,  
3     IPointDrawer point_drawer,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel)
```

- Summary:
 - create a new oscilloscope.
- Remarks:
 - every input objects should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:
 - [ICanvas](#)<T> canvas: the canvas that produce the graph.
 - [IPointDrawer](#) point_drawer: the point-drawer the producer will produce the graph with.
 - [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.
 - [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
 - ConcurrentQueue<T> buffer: the buffer of this oscilloscope, if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
- Normal-Behaviour:
 - Pre-Condition:
 - canvas.GraphSize == point_drawer.GraphSize

- !graph_producer.RequireConcurrentDrawer || point_drawer.IsConcurrent
 - Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas.GraphSize != point_drawer.GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent
-

methods:

```
1 | public T Draw(double delta_time);
```

- Summary:
 - it will call and return the result of [SingleThreadOscilloscope.Draw](#) directly.
 - get the current state of the panel and produce a new graph according to this, then return the graph while finish.
 - Params:
 - double delta_time: the time during which the point will be drawn on the graph. in short you'd better delivery the time span from the latest call of this method. it should not be negative.
 - Normal-Behaviour:
 - Pre-condition:
 - delta_time >= 0.
 - Post-Condition:
 - a new graph with type T will be produced and return.
-

TimeCountedOscilloscope

```
1 public class TimeCountedOscilloscope<T> : SingleThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: [SingleThreadOscilloscope](#)
- interfaces: none
- summary:
 - the only difference with [SimpleOscilloscope](#) is the method [Draw\(\)](#) will use a built-in watch to get delta-time.
- constructors:

name	description
SingleThreadOscilloscope (ICanvas <T>, IPointDrawer , IGraphProducer , IControlPanel)	

- methods:

name	description
protected T Draw ()	produce and get a new graph.

constructors:

```
1 protected TimeCountedOscilloscope(  
2     ICanvas<T> canvas,  
3     IPointDrawer point_drawer,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel)
```

- Summary:
 - create a new oscilloscope.
- Remarks:
 - every input objects should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:
 - [ICanvas](#)<T> canvas: the canvas that produce the graph.
 - [IPointDrawer](#) point_drawer: the point-drawer the producer will produce the graph with.
 - [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.
 - [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
 - ConcurrentQueue<T> buffer: the buffer of this oscilloscope, if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
- Normal-Behaviour:
 - Pre-Condition:

- canvas.GraphSize == point_drawer.GraphSize
 - !graph_producer.RequireConcurrentDrawer || point_drawer.IsConcurrent
 - Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas.GraphSize != point_drawer.GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent
-

methods:

```
1 | public T Draw();
```

- Summary:
 - it will get delta_time with built-in watch.
 - it will call and return the result of [SingleThreadOscilloscope.Draw](#) directly.
 - get the current state of the panel and produce a new graph according to this, then return the graph when finished.
 - Params:
 - double delta_time: the time during which the point will be drawn on the graph. in short you'd better deliver the time span from the latest call of this method.
 - Normal-Behaviour:
 - Post-Condition:
 - a new graph with type T will be produced and return.
-

MultiThreadOscilloscope

```
1 public abstract class MultiThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: none
- interfaces: none
- summary:
 - an oscilloscope that can start a new draw-task while the old one has not finished.
 - T is the output type of this oscilloscope.
- remarks
 - this is a abstract class, if you want to use it, please try [UndrivedOscilloscope](#) or [DrivenOscilloscope](#).
 - calling [Draw\(\)](#) to start a draw-task, and after the draw-task is complete, a new graph will be put into [Buffer](#).
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- constructors:

name	description
MultiThreadOscilloscope (ConstructorTuple<ICanvas<T>>, ConstructorTuple<IPointDrawer>, IGraphProducer, IControlPanel[, ConcurrentQueue<T>=null])	

- attributes:

type	name	accessor	description
ConcurrentQueue<T>	Buffer	G	the productions of this oscilloscope will be put into this buffer.

- methods:

name	description
protected void Draw (double)	get the current state of the panel and produce a new graph accoding to this.then put the new graph into Buffer

constructors:

```
1 public MultiThreadOscilloscope(  
2     ConstructorTuple<ICanvas<T>> canvas_constructor,  
3     ConstructorTuple<IPointDrawer> point_drawer_constructor,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel,  
6     ConcurrentQueue<T> buffer = null)
```

- Summary:
 - create a new Oscilloscope.

- Remarks:
 - the control_panel and graph_producer should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:
 - [ConstructorTuple<ICanvas>](#) canvas_constructor: a ConstructorTuple that can create new ICanvas.
 - [ConstructorTuple<IPointDrawer>](#) point_drawer_constructor: a ConstructorTuple that can create new IPointDrawer.
 - [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.
 - [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
 - ConcurrentQueue<T> buffer: the buffer of this oscilloscope. if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
- Normal-Behaviour:
 - Pre-Condition:
 - canvas_constructor.NewInstance().GraphSize == point_drawer_constructor.NewInstance().GraphSize
 - !graph_producer.RequireConcurrentDrawer || point_drawer_constructor.NewInstance().IsConcurrent
- Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent

attributes:

```
1 | public ConcurrentQueue<T> Buffer { get; }
```

- Summary:
 - the productions of this oscilloscope will be put into this buffer.
 - the reference of buffer will never change.
-

methods:

```
1 | protected void Draw(double delta_time);
```

- Summary:

- get the current state of the panel and produce a new graph according to this. then put the new graph into [Buffer](#)
- Params:
 - double delta_time: the time during which the point will be drawn on the graph. in short you'd better deliver the time span from the latest call of this method. it should not be negative.
- Normal-Behaviour:
 - Pre-condition:
 - $\text{delta_time} \geq 0$.
 - Post-Condition:
 - a new graph with type T will be produced and put into [Buffer](#)

UndrivedOscilloscope

```
1 public class UndrivedOscilloscope<T> : MultiThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: `MultiThreadOscilloscope<T>`
- interfaces: none
- summary:
 - the only difference with [MultiThreadOscilloscope](#) is that the [Draw\(\)](#) of [UndrivedOscilloscope](#) is public.
- constructors:

name	description
UndrivedOscilloscope (ConstructorTuple<ICanvas<T>>, ConstructorTuple<IPointDrawer>, IGraphProducer, IControlPanel[, ConcurrentQueue<T>=null])	

- methods:

name	description
void Draw (double)	call MultiThreadOscilloscope.Draw() directly.

constructors:

```
1 public UndrivedOscilloscope(  
2     ConstructorTuple<ICanvas<T>> canvas_constructor,  
3     ConstructorTuple<IPointDrawer> point_drawer_constructor,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel,  
6     ConcurrentQueue<T> buffer = null)
```

- Summary:
 - create a new Oscilloscope.
 - the same as [MultiThreadOscilloscope](#).
- Remarks:
 - the control_panel and graph_producer should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:
 - [ConstructorTuple](#)<[ICanvas](#)<T>> canvas_constructor: a ConstructorTuple that can create new ICanvas.
 - [ConstructorTuple](#)<[IPointDrawer](#)> point_drawer_constructor: a ConstructorTuple that can create new IPointDrawer.
 - [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.

- [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
 - ConcurrentQueue<T> buffer: the buffer of this oscilloscope, if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
 - Normal-Behaviour:
 - Pre-Condition:
 - canvas_constructor.NewInstance().GraphSize == point_drawer_constructor.NewInstance().GraphSize
 - !graph_producer.RequireConcurrentDrawer || point_drawer_constructor.NewInstance().IsConcurrent
 - Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent
-

methods:

```
1 | public void Draw(double delta_time);
```

- Summary:
 - it will call [MultiThreadOscilloscope.Draw\(\)](#) directly.
 - get the current state of the panel and produce a new graph according to this. then put the new graph into [Buffer](#)
- Params:
 - double delta_time: the time during which the point will be drawn on the graph. in short you'd better delivery the time span from the latest call of this method. it should not be negative.
- Normal-Behaviour:
 - Pre-condition:
 - delta_time >= 0.
 - Post-Condition:
 - a new graph with type T will be produced and put into [Buffer](#)

DrivedOscilloscope

```
1 public class DrivedOscilloscope<T> : MultiThreadOscilloscope<T>;
```

- namespace: [OscilloscopeKernel](#)
- supers: MultiThreadOscilloscope<T>
- interfaces: none
- summary:
 - a multi-thread oscilloscope that contains a built-in timer.
 - it will produce graphs periodically and put them into the [Buffer](#).
- constructors:

name	description
DrivedOscilloscope (ConstructorTuple<ICanvas<T>>, ConstructorTuple<IPointDrawer>, IGraphProducer, IControlPanel[, ConcurrentQueue<T>=null])	

- attributes:

type	name	accessor	description
bool	IsRunning	G	marks wheather this oscilloscope is running

- methods:

name	description
void Start (int)	start to produce graphs periodically.
void End ()	stop this oscilloscope.

constructors:

```
1 public DrivedOscilloscope(  
2     ConstructorTuple<ICanvas<T>> canvas_constructor,  
3     ConstructorTuple<IPointDrawer> point_drawer_constructor,  
4     IGraphProducer graph_producer,  
5     IControlPanel control_panel,  
6     ConcurrentQueue<T> buffer = null)
```

- Summary:
 - create a new Oscilloscope.
 - the same as [MultiThreadOscilloscope](#).
- Remarks:
 - the control_panel and graph_producer should not be used by other oscilloscope at the same time.
 - no attribute or method will be provided to get the panel that this oscilloscope is using, so you need to handle the reference of it by yourself.
- Params:

- [ConstructorTuple<ICanvas>](#) canvas_constructor: a ConstructorTuple that can create new ICanvas.
- [ConstructorTuple<IPointDrawer>](#) point_drawer_constructor: a ConstructorTuple that can create new IPointDrawer.
- [IGraphProducer](#) graph_producer: a certain GraphProducer, MultiThreadOscilloscope requires a concurrent producer, which means producer.[Produce\(\)](#) can be called by different thread.
- [IControlPanel](#) control_panel: the user-interface of this oscilloscope.
- ConcurrentQueue<T> buffer: the buffer of this oscilloscope, if null, a new ConcurrentQueue will be created as the buffer, and then you could get it with attribute [Buffer](#).
- Normal-Behaviour:
 - Pre-Condition:
 - canvas_constructor.NewInstance().GraphSize == point_drawer_constructor.NewInstance().GraphSize
 - !graph_producer.RequireConcurrentDrawer || point_drawer_constructor.NewInstance().IsConcurrent
- Exception-Behaviour:
 - Exception: OscilloscopeBuildException with inner-exception: DifferentGraphSizeException
 - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
 - Exception: OscilloscopeBuildException
 - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent

attributes:

```
1 | public bool IsRunning { get; }
```

- Summary:
 - marks wheather this oscilloscope is running
- Remarks
 - while IsRunning is true, ths oscilloscope will produce a new graph and put it into the [Buffer](#) periodically.
- Getter

methods:

```
1 | public void start(int delta_time);
```

- Summary:
 - the oscilloscope start to run, which means it will put a new graph into the [Buffer](#) every `delta_time`.
- Remarks:
 - be careful about the time unit of delta_time. the time unit is still difined with [Waves.UNIT_NUMBER PRO SECOND](#).

- Params:
 - int delta_time: the period that this oscilloscope produce a new graph and put into the [Buffer](#).
 - Normal-Behaviour:
 - Pre-Condition:
 - IsRunning == true
 - Post-Condition:
 - stop and then restart to run.
 - IsRunning == true
 - Normal-Behaviour:
 - Pre-Condition:
 - IsRunning == false
 - Post-Condition:
 - start to run.
 - IsRunning == true
-

```
1 | public void End()
```

- Summary:
 - stop this oscilloscope.
 - Remarks:
 - if the oscilloscope is not running, nothing will happen.
 - Normal-Behaviour:
 - Pre-Condition:
 - IsRunning == true
 - Post-Condition:
 - the oscilloscope will stop producing graphs periodically
 - IsRunning == false
 - Normal-Behaviour:
 - Pre-Condition:
 - IsRunning == false
 - Post-Condition:
 - nothing will happen
-

Wave

```
1 namespace OscilloscopeKernel.Wave
```

Summary:

- tools to describe electric waves with time and voltage.

[illegible]

IWave

```
1 public interface IWave
2 {
3     double MeanVoltage { get; }
4
5     int Period { get; }
6
7     double Voltage(double phase);
8 }
```

- namespace: [OscilloscopeKernel.Wave](#)
- supers: none
- interfaces: none
- summary:
 - describe a periodic wave with time, phase and voltage.
- remarks
 - every object that implement this interface should be **immutable** object.
 - if you want to change a wave, you can build a special class implementing IWave, whose constructor receive an IWave object as origin-wave. just like how [WaveReverser](#) do.
 - if you want a wave variable, you'd better not let it implement IWave. you could add a `GetStateshot()` method to return an IWave at certain time, just like how [WaveFixer](#) do.
 - this wave can be described with a function $f(t)$. the voltage at time t is $f(t)$. $\exists S_T, s. t. \forall T \in S_T, f(t) = f(t + T)$, then we define the period of this wave as $T = \min(S_T)$, define the phase of this wave at time t as $p = \frac{t}{T} \bmod 1$. in `IWave`, we use [Period](#) to describe T and use [Voltage](#)(double phase) to describe $f_p(p) = f(p \cdot T)$.
- attributes:

type	name	accessor	description
double	MeanVoltage	G	the mean voltage
int	Period	G	the period of this wave

- methods:

name	description
double Voltage (double)	return the voltage of this wave with certain phase

attributes:

```
1 double MeanVoltage { get; }
```

- Summary:
 - the mean voltage of this wave.
- Remarks
 - definition: $\text{MeanVoltage} = \int_0^1 \text{Voltage}(p) dp$
 - [Waves.CalculateMeanVoltage\(\)](#) can calculate the meanvoltage with difinition.

- Invariant:
 - $\text{MeanVoltage} = \int_0^1 \text{Voltage}(p) dp$
 - Getter
-

```
1 | int Period { get; }
```

- Summary:
 - the period of this wave.
 - Remarks
 - the voltage at time t is the same as the voltage at time $t + \text{Period}$
 - Getter
-

methods:

```
1 | double Voltage(double phase);
```

- Summary:
 - the voltage at certain phase.
 - Params:
 - double phase: $\text{phase} \in [0, 1)$. no exception will be raise if not, but it is still an undifined behavior.
 - Return:
 - double: $f_p(p) = f(p \cdot T)$
 - Normal-Behaviour:
 - Pre-Condition:
 - $\text{phase} \in [0, 1)$
 - Post-Condition:
 - return $f_p(p) = f(p \cdot T)$
 - Exception-Behaviour:
 - Exception null (no Exception will be throw out but this is undefined behavior):
 - $\text{phase} < 0 \ || \ \text{phase} \geq 1$
-

Waves

```
1 | public static class waves
```

- namespace: [OscilloscopeKernel.Wave](#)
- supers: none
- interfaces: none
- summary:
 - a static class providing basics operations for IWave.
- attributes:

type	name	accessor	description
ConstantWave	NONE	readonly	GND signal
int	UNIT_NUMBER_PRO_SECOND	readonly	time-unit of this classlib is $\frac{1}{\text{UNIT_NUMBER_PRO_SECOND}} s$

- methods:

name	description
double GetFrequency (IWave)	get the frequency of certain wave.
double CalculateMeanVoltage (IWave[, int=1000])	calculate the mean voltage of certain wave according to difination.
IWave Add (IWave,IWave)	add two wave

attributes:

```
1 | public static readonly ConstantWave NONE = new ConstantWave(0);
```

- Summary:
 - GND signal
- readonly

```
1 | public static readonly int UNIT_NUMBER_PRO_SECOND = 1000_000;
```

- Summary:
 - time-unit of this classlib is $\frac{1}{\text{UNIT_NUMBER_PRO_SECOND}} s$
- Remarks
 - UNIT_NUMBER_PRO_SECOND = 1000_000 means the time-unit of this classlib is μs .
- readonly

methods:

```
1 | public static double GetFrequency(IWave wave);
```

- Summary:
 - get the frequency of certain wave.
- Remarks:
 - return [UNIT NUMBER PRO SECOND](#) / (double)(wave.Period);
- Params:
 - IWave wave: the wave to calculate frequency.
- Return:
 - double: the frequency of this wave. frequency-unit is Hz.
- Normal-Behaviour:
 - Post-Condition:
 - return [UNIT NUMBER PRO SECOND](#) / (double)(wave.Period);

```
1 | public static double CalculateMeanVoltage(IWave wave, int calculate_times = 1000);
```

- Summary:
 - calculate the mean voltage of certain wave according to definition.
- Remarks:
 - this function is time-consuming, you'd better use `wave.MeanVoltage` to get the mean-voltage of wave if possible.
 - this function is mainly used to help the constructor of a wave calculating the mean-voltage.
- Params:
 - [IWave](#) wave: the wave that need to calculate mean_voltage.
 - int calculate_times: the bigger calculate_times, the more precise the result will be, but the more time it will cost.
- Return:
 - double : $\frac{1}{calculate_times} \sum_{i=0}^{calculate_times} wave.Voltage(\frac{i}{calculate_times})$
- Normal-Behaviour:
 - Pre-Condition:
 - wave can be partly initialized, but make sure wave.[Voltage\(\)](#) can work correctly.
 - Post-Condition
 - return $\frac{1}{calculate_times} \sum_{i=0}^{calculate_times} wave.Voltage(\frac{i}{calculate_times})$

```
1 | public static IWave Add(IWave left, IWave right)
```

- Summary:
 - add two wave.
- Remarks:
 - suggest we describe left-wave by function $f_1(t)$, and right-wave by function $f_2(t)$, this function will return a new wave described by function $f_3(t) = f_1(t) + f_2(t)$.

- the Period of the output wave will be the LCM (lowest common multiple) of the Period of each input wave.
 - Params:
 - [IWave](#) left: a wave that need to be add.
 - [IWave](#) right: a wave that need to be add.
 - Return:
 - IWave: a wave that observe the rules in Remarks.
-

Tools

```
1 namespace OscilloscopeKernel.Tools
```

Summary:

-

Drawing

```
1 namespace OscilloscopeKernel.Drawing
```

Summary:

-

Producer

```
1 namespace OscilloscopeKernel.Producer
```

Summary:

-

Exceptions

```
1 namespace OscilloscopeKernel.Exceptions
```

Summary:

-

OscilloscopeFramework

```
1 namespace OscilloscopeFramework
```

Summary:

-