Document of ClassLib OscilloscopeKernel

Index

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 Systerm 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

```
namespace OscilloscopeKernel
```

Summary:

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

type	name	description
abstract class	SingleThreadOscilloscope	an abstract class thar 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 thar 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

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

- namespace: OscilloscopeKernel
- inheritance: Object → SingleThreadOscilloscope<T>
- 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	describtion
SingleThreadOscilloscope(ICanvas<T>, IPointDrawer,IGraphProducer,IControlPanel)	

- methods:

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

constructors:

```
protected SingleThreadOscilloscope(
            ICanvas<T> canvas,
            IPointDrawer point_drawer,
            IGraphProducer graph_producer,
            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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
    - canvas.GraphSize != point_drawer.GraphSize
  - Exception: OscillocopeBuildException
    - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent

---

methods:

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

- Summary:
  - get the current state of the panel and produce a new graph accoding 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

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

- namespace: OscilloscopeKernel
- inheritance: Object → SingleThreadOscilloscope<T> → SimpleOscilloscope<T>
- interfaces: none
- summary:
  - the only difference with SingleThreadOscilloscope is the method Draw() is puiblic.
- constructors:

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

- methods:

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

constructors:

```
protected SimpleOscilloscope(
            ICanvas<T> canvas,
            IPointDrawer point_drawer,
            IGraphProducer graph_producer,
            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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
        - canvas.GraphSize != point_drawer.GraphSize
    - Exception: OscillocopeBuildException
        - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent

---

methods:

```
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 accoding 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

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

- namespace: OscilloscopeKernel
- inheritance: Object → SingleThreadOscilloscope<T> → TimeCountedOscilloscope<T>
- interfaces: none
- summary:
  - the only difference with SimpleOscilloscope is the method Draw() will use a built-in watch to get delta-time.
- constructors:

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

- methods:

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

constructors:

```
protected TimeCountedOscilloscope(
            ICanvas<T> canvas,
            IPointDrawer point_drawer,
            IGraphProducer graph_producer,
            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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
        - canvas.GraphSize != point_drawer.GraphSize
    - Exception: OscillocopeBuildException
        - graph_producer.RequireConcurrentDrawer && !point_drawer.IsConcurrent

---

methods:

```
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 accoding 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.
- Normal-Behaviour:
    - Post-Condition:
        - a new graph with type T will be produced and return.

MultiThreadOscilloscope

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

- namespace: OscilloscopeKernel
- inheritance: Object → MultiThreadOscilloscope<T>
- 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 DrivedOscilloscope.
  - 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	describtion
MultiThreadOscilloscope(ConstructorTuple<ICanvas<T>>, ConstructorTuple<IPointDrawer>, IGraphProducer, IControlPanel[, ConcurrentQueue<T>=null])	

- attributes:

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

- methods:

name	describtion
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:

```
public MultiThreadOscilloscope(
    ConstructorTuple<ICanvas<T>> canvas_constructor,
    ConstructorTuple<IPointDrawer> point_drawer_constructor,
    IGraphProducer graph_producer,
    IControlPanel control_panel,
    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<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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
        - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
    - Exception: OscillocopeBuildException
        - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent

attributes:

```
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:

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

- Summary:

    - get the current state of the panel and produce a new graph accoding 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

UndrivedOscilloscope

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

- namespace: OscilloscopeKernel
- inheritance: Object → MultiThreadOscilloscope<T> → UndrivedOscilloscope<T>
- interfaces: none
- summary:
  - the only difference with MultiThreadOscilloscope is that the Draw() of UndrivedOscilloscope is public.
- constructors:

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

- methods:

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

constructors:

```
public UndrivedOscilloscope(
    ConstructorTuple<ICanvas<T>> canvas_constructor,
    ConstructorTuple<IPointDrawer> point_drawer_constructor,
    IGraphProducer graph_producer,
    IControlPanel control_panel,
    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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
        - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
    - Exception: OscillocopeBuildException
        - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent

---

methods:

```
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 accoding 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

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

- namespace: OscilloscopeKernel
- inheritance: Object → MultiThreadOscilloscope<T> → DrivedOscilloscope<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	describtion
DrivedOscilloscope(ConstructorTuple<ICanvas<T>>, ConstructorTuple<IPointDrawer>, IGraphProducer, IControlPanel[, ConcurrentQueue<T>=null])	

- attributes:

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

- methods:

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

constructors:

```
public DrivedOscilloscope(
    ConstructorTuple<ICanvas<T>> canvas_constructor,
    ConstructorTuple<IPointDrawer> point_drawer_constructor,
    IGraphProducer graph_producer,
    IControlPanel control_panel,
    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 requirs 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: OscillocopeBuildException with inner-exception: DifferentGraphSizeException
    - canvas_constructor.NewInstance().GraphSize != point_drawer_constructor.NewInstance().GraphSize
  - Exception: OscillocopeBuildException
    - graph_producer.RequireConcurrentDrawer && !point_drawer_constructor.NewInstance().IsConcurrent

---

attributes:

```
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:

```
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

```
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

```
namespace OscilloscopeKernel.Wave
```

Summary:

- tools to describe electric waves with time and voltage.

type	name	description
interface	IWave	describe a periodic wave with time, phase and voltage.
static class	Waves	providing basics operations for IWave.
abstract class	AbstractWave	a better IWave providing base operations for waves.
class	FunctionWave	a wave created with a $f_p(p)$.
class	SinWave	a wave described as $f_p(p)$ = max_voltage $\cdot \sin(2\pi p)$.
class	SawToothWave	a wave described as $f_p(p)$ = max_voltage $\cdot (2p - 1)$.
class	SquareWave	a wave described as $f_p(p)$ = max_voltage · (p < $\frac{1}{2}$? -max_voltage : max_voltage)
class	ConstantWave	a wave described as $f_p(p)$ = voltage. A DC wave.
class	WaveFixer	a mutable wave.

IWave

```
    public interface IWave
    {
        double MeanVoltage { get; }

        int Period { get; }

        double Voltage(double phase);
    }
```

- namespace: OscilloscopeKernel.Wave
- 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	describtion
double	MeanVoltage	G	the mean voltage
int	Period	G	the period of this wave

- methods:

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

attributes:

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

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

    - $\mathrm{MeanVoltage} = \int_0^1 \mathrm{Voltage}(p)\mathrm{d}p$
- Getter

```
int Period { get; }
```

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

---

methods:

```
double Voltage(double phase);
```

- Summary:
    - the voltage at certain phase.
- Params:
    - double phase: $\mathrm{phase} \in [0, 1)$. 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:
        - $\mathrm{phase} \in [0, 1)$
    - 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):
        - phase < 0 || phase >= 1

Waves

```
public static class Waves
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object $\longrightarrow$ Waves
- interfaces: none
- summary:
  - a static class providing basics operations for IWave.
- attributes:

type	name	accessor	describtion
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	describtion
double GetFrequence(IWave)	get the frequence of certain wave.
double CalculateMeanVoltage(IWave[, int=1000])	calculate the mean voltage of certain wave accoding to difination.
AbstractWave Add(IWave,IWave)	add two wave, $g(t) = f_1(t) + f_2(t)$
AbstractWave Negative(IWave)	return a wave $g(t) = -f(t)$
AbstractWave Reverse(IWave)	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \text{mod } 1) = f_p(1 - (\frac{t}{T} \text{mod } 1)) = f(T - t)$
AbstractWave Decorate(IWave)	decorate a IWave as an AbstractWave

attributes:

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

- Summary:
  - GND signal
- readonly

```
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:

```
public static double GetFrequence(IWave wave);
```

- Summary:
    - get the frequence of certain wave.
- Remarks:
    - return UNIT_NUMBER_PRO_SECOND / (double)(wave.Period);
- Params:
    - IWave wave: the wave to calculate frequence.
- Return:
    - double: the frequence of this wave. frequence-unit is Hz.
- Normal-Behaviour:
    - Post-Condition:
        - return UNIT_NUMBER_PRO_SECOND / (double)(wave.Period);

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

- Summary:
    - calculate the mean voltage of certain wave accoding to difination.
- 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 calcutate_times, the more precise the result will be, but the more time it will cost.
- Return:
    - double : $\frac{1}{calculate\_times}\sum_{i=0}^{\text{calculate\_times}} \text{wave. Voltage}(\frac{\text{i}}{\text{calculate\_times}})$
- Normal-Behaviour:
    - Pre-Condition:
        - wave can be partly initialized, but make sure wave.Voltage() can work correctly.
    - Post-Condition
        - return $\frac{1}{calculate\_times}\sum_{i=0}^{\text{calculate\_times}} \text{wave. Voltage}(\frac{\text{i}}{\text{calculate\_times}})$

```
public static AbstractWave Add(IWave left, IWave right);
```

- Summary:
    - add two wave, $g(t) = f_1(t) + f_2(t)$.
- Remarks:
    - suggest we discribe left-wave by function $f_1(t)$, and right-wave by function $f_2(t)$, this function will return a new wave discribed 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:
    - AbstractWave: a wave that observe the rules in Remarks.

```
public static AbstractWave Negative(IWave origin);
```

- Summary:
    - return a wave $g(t) = -f(t)$.
- Params:
    - IWave origin: origin wave;
- Return:
    - AbstractWave: a new AbstractWave;
- Normal-Behaviour:
    - Pre-Condition:
        - origin is an `immutable` object;
    - Post-Condition:
        - return AbstractWave new_wave;
        - new_wave.MeanVoltage + origin.MeanVoltage == 0;
        - new_wave.Period == origin.Period;
        - $\forall$ double p $\in [0, 1)$, new_wave.Voltage(p) + origin.Voltage(p) == 0;

```
public static AbstractWave Reverse(IWave origin);
```

- Summary:
    - reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \bmod 1) = f_p(1 - (\frac{t}{T} \bmod 1)) = f(T - t)$
- Params:
    - IWave origin: origin wave;
- Return:
    - AbstractWave: a new AbstractWave;
- Normal-Behaviour:
    - Pre-Condition:
        - origin is an `immutable` object;
    - Post-Condition:

        - return AbstractWave new_wave;
        - new_wave.MeanVoltage == origin.MeanVoltage;
        - new_wave.Period == origin.Period;
        - $\forall$ double p $\in [0, 1)$, new_wave.Voltage(p) == origin.Voltage(1 - p);

```
public static AbstractWave Decorate(IWave origin);
```

- Summary:
    - decorate a IWave as an AbstractWave
- Params:
    - IWave origin: origin wave;
- Return:
    - AbstractWave: a new AbstractWave;
- Normal-Behaviour:
    - Pre-Condition:
        - origin is an `immutable` object;
    - Post-Condition:
        - return AbstractWave new_wave;
        - new_wave.MeanVoltage == origin.MeanVoltage;
        - new_wave.Period == origin.Period;
        - $\forall$ double p $\in [0, 1)$, new_wave.Voltage(p) == origin.Voltage(p);

AbstractWave

```
public abstract class AbstractWave : IWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object $\longrightarrow$ AbstractWave
- interfaces: IWave
- summary:
  - a better IWave providing base operations for waves.
- remarks
  - Each `AbstractWave` should be an immutable object.
  - There is no fields in this class, so there is only default constructor.
  - The only reason why this class is designed is that, in .NET Standard 2.0, I cannot use C# 8.0, so I cannot add those operations to IWave derectly.
  - operator suntraction of 2 element is not provided, you can use `wave1 + (-wave2)` instead of `wave1 - wave2`, the latter is wrong.
- attributes:

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

- methods:

name	describtion
abstract double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \text{mod } 1) = f_p(1 - (\frac{t}{T} \text{mod } 1)) = f(T - t)$

- operators:

name	describtion
AbstractWave Subtraction(AbstractWave)	return a wave $g(t) = -f(t)$
AbstractWave Addition(AbstractWave, IWave)	add two wave, $g(t) = f_1(t) + f_2(t)$
AbstractWave Addition(IWave, AbstractWave)	add two wave, $g(t) = f_1(t) + f_2(t)$

attributes:

```
public abstract double MeanVoltage { get; }
```

- see also:
  - Wave.IWave.MeanVoltage.

```
public abstract int Period { get; }
```

- see also:
  - Wave.IWave.Period.

---

methods:

```
public abstract double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse();
```

- Summary:
  - reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \mod 1) = f_p(1 - (\frac{t}{T} \mod 1)) = f(T - t)$
- Remarks:
  - it behave the same as Waves.Reverse(this).
- Return:
  - AbstractWave: a new AbstractWave;
- Normal-Behaviour:
  - Post-Condition:
    - return AbstractWave new_wave;
    - new_wave.MeanVoltage == this.MeanVoltage
    - new_wave.Period == this.Period;
    - $\forall$ double p $\in [0, 1)$, new_wave.Voltage(p) == this.Voltage(1 - p);

---

operators:

```
public static AbstractWave operator -(AbstractWave origin);
```

- Summary:
  - return a wave $g(t) = -f(t)$.
- Remarks:
  - it behave the save as Waves.Negative(this).
- Params:
  - IWave origin: origin wave;
- Return:
  - AbstractWave: a new AbstractWave;
- Normal-Behaviour:
  - Post-Condition:
    - return AbstractWave new_wave;

        - new_wave.MeanVoltage + origin.MeanVoltage == 0;
        - new_wave.Period == origin.Period;
        - ∀ double p $\in [0, 1)$, new_wave.Voltage(p) + origin.Voltage(p) == 0;

```
public static AbstractWave operator +(AbstractWave left, IWave right);
```

- Summary:
    - add two wave, $g(t) = f_1(t) + f_2(t)$.
- Remarks:
    - it behave the save as Waves.Add(left, right).
    - suggest we discribe left-wave by function $f_1(t)$, and right-wave by function $f_2(t)$, this function will return a new wave discribed 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:
    - AbstractWave left: a wave that need to be add.
    - IWave right: a wave that need to be add.
- Return:
    - AbstractWave: a wave that observe the rules in Remarks.

```
public static AbstractWave operator +(IWave left, AbstractWave right);
```

- Summary:
    - add two wave, $g(t) = f_1(t) + f_2(t)$.
- Remarks:
    - it behave the save as Waves.Add(left, right).
    - suggest we discribe left-wave by function $f_1(t)$, and right-wave by function $f_2(t)$, this function will return a new wave discribed 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.
    - AbstractWave right: a wave that need to be add.
- Return:
    - AbstractWave: a wave that observe the rules in Remarks.

FunctionWave

```
public class FunctionWave : AbstractWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object → Abstractwave → FunctionWave
- interfaces: IWave
- summary:
    - a wave created with a $f_p(p)$.
- remarks
    -
- delegates:

name	describtion
double WaveFunction(double).	describtion of $f_p(p)$

- constructors:

name	describtion
FunctionWave(WaveFunction, int[, double=1])	create a FunctionWave and MeanVoltage will be calculated automatically.
FunctionWave(WaveFunction, int, double, double)	create a FunctionWave, using given MeanVoltage.

- attributes:

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

- methods:

name	describtion
double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \bmod 1) = f_p(1 - (\frac{t}{T} \bmod 1)) = f(T - t)$

delegates

```
public delegate double WaveFunction(double phase);
```

- Summary:
    - describtion of $f_p(p)$

- Remark:
    - phase $\in [0, 1)$.

constructors:

```
public FunctionWave(WaveFunction function, int period, double voltage_times = 1);
```

- Summary:
    - create a FunctionWave and MeanVoltage will be calculated automatically.
- Remarks:
    - It may take some time to calculate the mean_voltage. If you want to make it faster, try to use another constructor.
- Params:
    - WaveFunction function: the describtion of $f_p(p)$;
    - int period: the Period;
    - double voltage_times: this.Voltage(p) == voltage_times $\cdot$ function(p).

```
public FunctionWave(WaveFunction function, int period, double voltage_times, double function_mean);
```

- Summary:
    - Create a FunctionWave, using given MeanVoltage.
- Remarks:
    - Please make sure function_mean is correct. No check will be provided. function_mean == $\int_0^1 \mathrm{function}(p)\mathrm{d}p$.
- Params:
    - WaveFunction function: the describtion of $f_p(p)$;
    - int period: the Period;
    - double voltage_times: this.Voltage(p) == voltage_times $\cdot$ function(p).
    - double function_mean: the mean of param function, which means this.MeanVoltage == voltage_times $\cdot$ function_mean.
- Normal-Behaviour:
    - Pre-Condition:
        - function_mean == $\int_0^1 \mathrm{function}(p)\mathrm{d}p$.

---

attributes:

```
public double MeanVoltage { get; }
```

- see also:
    - Wave.IWave.MeanVoltage.

```
public int Period { get; }
```

- see also:
  - Wave.IWave.Period.

methods:

```
public double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse()
```

- see also:
  - Wave.AbstractWave.Reverse().

SinWave

```
public class SinWave : FunctionWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object $\longrightarrow$ Abstractwave $\longrightarrow$ FinctionWave $\longrightarrow$ SinWave
- interfaces: IWave
- summary:
  - a wave described as $f_p(p)$ = max_voltage $\cdot \sin(2\pi p)$
- remarks
  - just like FunctionWave(phase => Math.Sin(2 * Math.PI * phase), period, max_voltage, 0);
- constructors:

name	describtion
SinWave(int, double)	create a sin-wave with given period and max_voltage

- attributes:

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

- methods:

name	describtion
double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \mathrm{mod}\ 1) = f_p(1 - (\frac{t}{T} \mathrm{mod}\ 1)) = f(T - t)$

constructors:

```
public SinWave(int period, double max_voltage);
```

- Summary:
  - create a sin-wave with given period and max_voltage.
- Params:
  - int period: the Period of this wave.
  - double max_voltage: the max voltage of this wave. In other way, $f_p(\frac{1}{4}) =$ max_voltage.

---

attributes:

```
public double MeanVoltage { get; }
```

- see also:
  - Wave.IWave.MeanVoltage.

```
public int Period { get; }
```

- see also:
  - Wave.IWave.Period.

---

methods:

```
public double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse()
```

- see also:
  - Wave.AbstractWave.Reverse().

SawToothWave

```
public class SawToothWave : FunctionWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object $\longrightarrow$ Abstractwave $\longrightarrow$ FinctionWave $\longrightarrow$ SawToothWave
- interfaces: IWave
- summary:
    - a wave described as $f_p(p)$ = max_voltage $\cdot(2p-1)$.
- remarks
    - just like FunctionWave(phase => 2 * phase - 1, period, max_voltage, 0);
- constructors:

name	describtion
SawToothWave(int, double)	create a sin-wave with given period and max_voltage

- attributes:

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

- methods:

name	describtion
double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \bmod 1) = f_p(1-(\frac{t}{T} \bmod 1)) = f(T-t)$

constructors:

```
public SawToothWave(int period, double max_voltage);
```

- Summary:
    - create a sin-wave with given period and max_voltage.
- Params:
    - int period: the Period of this wave.
    - double max_voltage: the max voltage of this wave. In other way, $\forall p \in [0,1), f_p(p) =$ max_voltage $\cdot(2p-1)$.

---

attributes:

```
public double MeanVoltage { get; }
```

- see also:
  - Wave.IWave.MeanVoltage.

```
public int Period { get; }
```

- see also:
  - Wave.IWave.Period.

---

methods:

```
public double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse()
```

- see also:
  - Wave.AbstractWave.Reverse().

SquareWave

```
public class SquareWave : FunctionWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object $\longrightarrow$ Abstractwave $\longrightarrow$ FinctionWave $\longrightarrow$ SquareWave
- interfaces: IWave
- summary:
  - a wave described as $f_p(p)$ = max_voltage · (p < $\frac{1}{2}$? -max_voltage : max_voltage)
- remarks
  - just like FunctionWave(phase => phase < 0.5 ? -1 : 1, period, max_voltage, 0);
- constructors:

name	describtion
SquareWave(int, double)	create a sin-wave with given period and max_voltage

- attributes:

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

- methods:

name	describtion
double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \bmod 1) = f_p(1 - (\frac{t}{T} \bmod 1)) = f(T - t)$

constructors:

```
public SquareWave(int period, double max_voltage);
```

- Summary:
  - create a sin-wave with given period and max_voltage.
- Params:
  - int period: the Period of this wave.
  - double max_voltage: the max voltage of this wave. In other way, $\forall p \in [0, \frac{1}{2}), f_p(p) =$ -max_voltage, $\forall p \in [\frac{1}{2}, 1), f_p(p) =$ max_voltage.

---

attributes:

```
public double MeanVoltage { get; }
```

- see also:
  - Wave.IWave.MeanVoltage.

```
public int Period { get; }
```

- see also:
  - Wave.IWave.Period.

---

methods:

```
public double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse()
```

- see also:
  - Wave.AbstractWave.Reverse().

ConstantWave

```
public class ConstantWave : AbstractWave
```

- namespace: OscilloscopeKernel.Wave
- inheritance: Object → Abstractwave → ConstantWave
- interfaces: IWave
- summary:
    - a wave described as $f_p(p)$ = voltage.
    - In other way, it is a DC wave.
- remarks
    - just like FunctionWave(phase => 1, 1, voltage, voltage);
- constructors:

name	describtion
ConstantWave(double)	create a DC-wave with given voltage

- attributes:

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

- methods:

name	describtion
double Voltage(double)	return the voltage of this wave with certain phase
AbstractWave Reverse()	reverse the phase of a wave, $g(t) = g_p(\frac{t}{T} \text{mod } 1) = f_p(1 - (\frac{t}{T} \text{mod } 1)) = f(T - t)$

constructors:

```
public ConstantWave(int period, double max_voltage);
```

- Summary:
    - create a DC-wave with given voltage.
- Params:
    - double voltage: the voltage of this wave. In other way, $\forall p \in [0, 1)$, $f_p(p) =$ voltage.

attributes:

```
public double MeanVoltage { get; }
```

- see also:
  - Wave.IWave.MeanVoltage.

```
public int Period { get; }
```

- see also:
  - Wave.IWave.Period.

---

methods:

```
public double Voltage(double phase);
```

- see also:
  - Wave.IWave.Voltage().

```
public AbstractWave Reverse()
```

- see also:
  - Wave.AbstractWave.Reverse().

WaveFixer

```
public class WaveFixer
```

- namespace: OscilloscopeKernel.
- inheritance: Object → WaveFixer
- interfaces: none
- summary:
  - a mutable wave.
- remarks
  - use GetStateShot() to get an AbstractWave as the shot of fixed wave now.
- constructors:

name	describtion
WaveFixer()	create a WaveFixer with GND wave
WaveFixer(IWave)	create a WaveFixer with given wave

- attributes:

type	name	accessor	describtion
double	VoltageTimes	GS	the times of voltage
double	PeriodTimes	GS	the times of period
IWave	Wave	GS	the base wave

- methods:

name	describtion
AbstractWave GetStateShot()	get the shot of the fixed wave now

constructors:

```
public WaveFixer();
```

- Summary:
  - create a WaveFixer with GND wave.
- Remarks:
  - the same as WaveFixer(Waves.NONE).

```
public WaveFixer(IWave wave);
```

- Summary:
  - create a WaveFixer with given wave
- Params:

  - IWave wave: given wave that this WaveFixer will use.

---

attributes:

```
public double VoltageTimes { get; set; }
```

- Summary:
  - the Voltage of fixed wave at phase p is Wave.Voltage(p) · VoltageTimes.
- Invarient:
  - $\forall p \in [0, 1)$, GetStateShot().Voltage(p) == Wave.Period * VoltageTimes.
- Getter
- Setter

```
public double PeriodTimes { get; set; }
```

- Summary:
  - the Period of fixed wave is (int)(Wave.Period · PeriodTimes).
- Invarient:
  - GetStateShot().Period == (int)(Wave.Period * PeriodTimes)
- Getter
- Setter

```
public IWave Wave { get; set; }
```

- Summary:
  - the base wave.
- Invarient:
  - GetStateShot().Period == (int)(Wave.Period * PeriodTimes)
  - $\forall p \in [0, 1)$, GetStateShot().Voltage(p) == Wave.Period * VoltageTimes.
- Getter
- Setter:
  - if value is null, Wave will be set to Waves.NONE.

---

methods:

```
public AbstractWave GetStateShot();
```

- Summary:
  - get the shot of the fixed wave now.
- Return:
  - AbstractWave: a new wave that can describe the wave now.

- Normal-Behaviour:
  - Post-Condition:
    - AbstractWave new_wave;
    - new_wave.Period == (int)(Wave.Period * PeriodTimes)
    - $\forall p \in [0, 1)$, new_wave.Voltage(p) == Wave.Period * VoltageTimes.
    - return new_wave.

Tools

```
namespace OscilloscopeKernel.Tools
```

Summary:

-

type	name	description

Drawing

```
namespace OscilloscopeKernel.Drawing
```

Summary:

-

type	name	description

Producer

```
namespace OscilloscopeKernel.Producer
```

Summary:

-

type	name	description

Exceptions

```
namespace OscilloscopeKernel.Exceptions
```

Summary:

-

type	name	description

OscilloscopeFramework

```
namespace OscilloscopeFramework
```

Summary:

-

type	name	description