

FROGS

Event-driven Simulator

Version 1.2

STATOBJ – Programmer's Manual

July 2001

StObject (extends StProto)

PURPOSE

This superclass implements the basic behavior of a statistical object. The **StProto** class is inherited to connect each measurement object to a display front-end through a protocol pilot. In other words, a statistical object's value can be exported to a front-end for display (usually an external application), and receive specific directives interactively that may alter its value and/or behavior.

A given statistical object can be used in two separate contexts:

- ⑩ As a backend object, it performs statistical measurements; It may also send the collected data to its front-end counterpart through its protocol pilot, and receive directives from it through the same channel. This operating mode may be seen as the object's master role.
- ⑩ As a front-end object, it is updated by data received from its protocol pilot, which was produced by its backend counterpart. This operating mode may be seen as the object's slave role.

The protocol pilot should be seen as a communication object aimed at exchanging messages back and forth between a measurement object and its display.

A statistical object has five major properties:

- ⑩ A current value (floating-point).
- ⑩ A count of received values since the object was last reset or created.
- ⑩ Two varying bounds (lower and upper) that are updated each time a new value is logged.
- ⑩ A current count of samples.

Subclassed statistical objects may need a time reference as part of their input. In such a case, the global variable **Clock** is used to retrieve the current time value.

This class is usually not instantiated directly, but rather subclassed to provide some specific statistical behavior.

CONSTRUCTORS

StObject(const char *name =0, StProtoPilot *pilot =0, int pflags =0)

Creates a statistical object. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StObject(StProtoExportMessage *pex, StProtoPilot *pilot)

Creates a slave statistical object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the `protoExport()` method being called for a backend object.

METHODS

virtual void add(double value)

Stores the current object's value. The object's lower and upper bounds are updated accordingly, and the number of collected values is incremented. Of course, this method is expected to be specialized by subclasses that performs more complex statistical processing.

void add(int value)

Invokes `add((double)value)`.

virtual void double getValue (StValueType vtype =VAL)

Gets the object's value. Because a statistical object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last added value to be returned.
- ⑩ NUM represents the number of collected values since the object was created or reset.
- ⑩ MINVAL represents the object's lower bound value. In other words, it is the lowest value collected through a call to `add()`, since the object was created or reset.
- ⑩ MAXVAL represents the object's upper bound value. In other words, it is the greatest value collected through a call to `add()`, since the object was created or reset.

Subclasses usually provide more value types as they hold additional statistical results.

virtual void resetValues()

Resets the object. The count of collected values and samples are set to zero. The lower and higher bounds are reset so as they bounce to the next collected value when available.

virtual void sample()

Performs a sampling operation on the object. At this class level, sampling only causes the internal count of samples to be incremented. Subclasses usually provide more complexity to this action, but they still should increment this count.

virtual void result()

This method is aimed at computing the statistical object's final value, just before it is inquired. At this class level, this method does nothing.

StObjectGroup (extends StObject)

PURPOSE

Instances of this class gather multiple statistical objects under a common name to perform grouped computation. Held objects are referred to by an index into an internal array. An extra cell is automatically reserved to store the group's global value.

The global value index is always #0. Individual object indices range from the specified index base to the sum of this base with the number of held objects. Negative bases are allowed.

This class should never be instantiated directly, but rather extended by subclasses implementing specific group behavior.

PROTECTED DATA MEMBERS

int ncells

The number of cells in the group's internal array (i.e. number of held objects + 1).

int iscale

The group's index scaling value (i.e. 1 – iBase).

StObject **vector

The array of pointers to the held objects. Index #0 is reserved for the global measure.

CONSTRUCTORS

StObjectGroup(const char *name, const char *indexName, int nitems, int iBase =0, StProtoPilot *pilot =0, int pflags =0)

Creates a statistical object group. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

indexName is the logical name of the index that will be used in composing the held objects names.

nitems is the number of statistical objects the group constructor will hold.

iBase is the the lowest index value for the group. Indices passed to methods requiring them will be scaled to take in account this threshold. For instance, passing an index base of -2 for a 4-elements group allows using indices -2, -1, 0 (global value), 1 and 2.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

This method allocates an array of pointers which can subsequently be used by the subclass constructor.

StObjectGroup(StProtoExportMessage *pex, StProtoPilot *pilot, const char *indexName, int nitems, int iBase =0)

Creates a slave group object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the **protoExport()** method being called for a backend object.

indexName, **nitems** and **iBase** have the same meaning than previously.

METHODS

StObject *getAddress(int nth)

Returns the address of the **nth** object from the group. The index scaling is applied.

virtual void add(double value)

Adds **value** to the global object's value.

void add(double value, int nth)

Adds **value** to the **nth** object from the group, then to the global object's value. The index scaling is applied.

void add(int value, int nth)

Invokes add((double)value,nth).

virtual void resetValues()

Resets the values in turn for each held object, including the global one.

virtual void sample()

Samples each held object in turn, including the global one.

virtual void result()

Computes the held objects' final results in turn, including the global object's one.

StCounter (extends StObject)

PURPOSE

This class implements a measurement object aimed at summing numerical values, with a sampling capability.

CONSTRUCTORS

StCounter(const char *name, StProtoPilot *pilot =0, int pflags =0)

Creates a statistical object group. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StCounter(StProtoExportMessage *pex, StProtoPilot *pilot)

Creates a slave group object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the protoExport() method being called for a backend object.

METHODS

virtual void add(double value)

Adds **value** to the counter's sum of values. The object's lower and upper bounds are updated accordingly, and the number of summed values is incremented.

virtual void double getValue (StValueType vtype =VAL)

Gets the object's value. Because a counter object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last added value to be returned.
- ⑩ CMES returns the sum on the current sample.
- ⑩ NUM represents the number of summed values since the object was created or reset.
- ⑩ SUM returns the global sum of values since the beginning of the measure.
- ⑩ SUM2 returns the square sum of values since the beginning of the measure.

- ⑩ MINVAL represents the object's lower bound value. In other words, it is the lowest value collected through a call to `add()`, since the object was created or reset.
- ⑩ MAXVAL represents the object's upper bound value. In other words, it is the greatest value collected through a call to `add()`, since the object was created or reset.
- ⑩ MEAN return the mean of values since the beginning of the measure.
- ⑩ STDEV returns the standard deviation since the beginning of the measure.

`void inc()`

Invokes `add(1.0)`.

`virtual void resetValues()`

Resets the object. The count of summed values and samples are set to zero. The lower and higher bounds are reset so as they bounce to the next added value when available.

`virtual void sample()`

Samples the counter, updating the sum and square sums for the current sample.

`virtual void result()`

No action.

StCounterGroup (extends StObjectGroup)

PURPOSE

Instances of this class gather multiple counter objects under a common name to perform grouped computation. Held counters are referred to by an index into an internal array. An extra cell is automatically reserved to store the group's global counter.

The global counter index is always #0. Individual counter indices range from a given index base to the sum of this base with the number of held counters. Negative bases are allowed.

CONSTRUCTORS

StCounterGroup(const char *name, const char *indexName, int nitems, int iBase =0, StProtoPilot *pilot =0, int pflags =0)

Creates a counter group. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

indexName is the logical name of the index that will be used in composing the held counters names.

nitems is the number of counters the group constructor will create.

iBase is the the lowest index value for the group. Indices passed to methods requiring them will be scaled to take in account this threshold. For instance, passing an index base of -2 for a 4-elements group allows using indices -2, -1, 0 (global counter), 1 and 2.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StObjectGroup(StProtoExportMessage *pex, StProtoPilot *pilot, const char *indexName, int nitems, int iBase =0)

Creates a slave group object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the **protoExport()** method being called for a backend object.

indexName, **nitems** and **iBase** have the same meaning than previously.

StIntegrator (extends StObject)

PURPOSE

This class implements a time integrator object which is aimed at computing the integration of a given variable with respect to time. The current time is obtained from the contents of the global variable `Clock`.

CONSTRUCTORS

`StIntegrator(const char *name, const ITime& tStart, const ITime& tEnd, const ITime& dtSample, StProtoPilot *pilot=0, int pflags=0)`

Creates a time integrator. **name** is a null-terminated character string identifying the new object; `NULL` is an acceptable value, preventing the anonymous object from being visible from the display front-end.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends `StProtoFrontPilot`) or master (if **pilot** extends `StProtoBackPilot`). If `NULL`, the object has no interface channel.

tStart is the starting time of survey, **tEnd** specifies its end. **dtSample** is the sampling period that applies for this object.

pflags is a set of protocol flags such as defined by the `StProto` interface.

`StIntegrator(StProtoExportMessage *pex, StProtoPilot *pilot, const ITime& tStart, const ITime& tEnd, const ITime& dtSample)`

Creates a slave time integrator, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the `protoExport()` method being called for a backend object.

tStart, **tEnd** and **dtSample** have the same meaning than previously.

METHODS

`virtual void add(double value)`

Integrates the last entered value up to the current time (0.0 if none), then sets the last value to **value**.

`void inc()`

Adds the last entered value incremented from 1.0 to the object.

`void dec()`

Adds the last entered value decremented from 1.0 to the object.

virtual double getValue(StValueType vtype =VAL)

Gets the object's value. Because an integrator object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last added value to be returned.
- ⑩ CMES returns the mean of the variable on the current sample.
- ⑩ NUM represents the number of summed values since the object was created or reset.
- ⑩ SUM returns the global integral since the beginning of the measure.

virtual void resetValues()

Resets the object. The count of collected values and samples are set to zero. The lower and higher bounds are reset so as they bounce to the next collected value when available.

virtual void sample()

Performs a sampling operation on the object.

virtual void result()

No action.

StIntegratorGroup (extends StObjectGroup)

PURPOSE

Instances of this class gather multiple time integrator objects under a common name to perform grouped computation. Held integrators are referred to by an index into an internal array. An extra cell is automatically reserved to store the group's global integrator.

The global integrator index is always #0. Individual integrator indices range from a given index base to the sum of this base with the number of held integrators. Negative bases are allowed.

CONSTRUCTORS

StIntegratorGroup(const char *name, const char *indexName, int nitems, const ITime& tStart, const ITime& tEnd, const ITime& dtSample, int iBase =0, StProtoPilot *pilot =0, int pflags =0)

Creates a time integrator group. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

indexName is the logical name of the index that will be used in composing the held integrators names.

nitems is the number of integrators the group constructor will create.

tStart is the starting time of survey, while **tEnd** specifies its end. **dtSample** is the sampling period that applies for this object.

iBase is the the lowest index value for the group. Indices passed to methods requiring them will be scaled to take in account this threshold. For instance, passing an index base of -2 for a 4-elements group allows using indices -2, -1, 0 (global integrator), 1 and 2.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StIntegratorGroup(StProtoExportMessage *pex, StProtoPilot *pilot, const char *indexName, int nitems, const ITime& tStart, const ITime& tEnd, const ITime& dtSample, int iBase =0)

Creates a slave group object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the protoExport() method being called for a backend object.

indexName, **nitems**, **tStart**, **tEnd**, **dtSample** and **iBase** have the same meaning than previously.

StHistogram (extends StObject)

PURPOSE

This measurement object is aimed at determining the probability density of any statistic law, and computes the mean and standard deviation and accuracy evaluation for a given confidence interval.

CONSTRUCTORS

StHistogram(const char *name, int nbins, double leftb, double rightb, StHistAdjustMode mode =MULTIPLY, StProtoPilot *pilot =0, int pflags =0)

Creates an histogram with floating-point bounds, namely **leftb** and **rightb**. **name** is a null-terminated character string identifying the new object that can be used during tracing and monitoring; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end. The number of counting bins is specified by **nbins**.

Histograms can automatically adjust themselves to the actual range of entered values. The expected behavior when a value falls outside the current range of an histogram is selectable by an adjustment **mode**:

- ⊗ **MULTIPLY** causes the histogram range to be adjusted by successive multiplications by 2, either to the left or to the right side according to the bound that is overshooted. This is the default mode.
- ⊗ **GARBAGE** prevents the histogram range to be adjusted, all the values falling outside this range are collected in the leftmost and rightmost bins.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StHistogram(const char *name, int nbins, int leftb, int rightb, StHistAdjustMode mode =MULTIPLY, StProtoPilot *pilot =0, int pflags =0)

Creates an histogram with integer bounds, namely **leftb** and **rightb**. Other parameters have the same meaning than previously.

The right bound and the number of counting bins are adjusted to obtain an integer bin size.

StHistogram(StHistogramExportMessage *pex, StProtoPilot *pilot =0)

Builds the histogram from values stored in the **pex** export message. If not NULL, **pilot** should point to a front-end protocol pilot object. This constructor is usually invoked on behalf of the **StProtoFrontPilot::createDisplay()** method to build the graphical counterpart of a statistic object that has just been exported by the backend program.

virtual void add(double value)

Adds **value** to the proper histogram's bin. If **value** is outside the current range of the histogram, then perform the adjustment according to the histogram's adjustment mode (whether MULTIPLY or GARBAGE).

virtual double getValue(StValueType vtype =VAL)

Gets the object's value. Because a histogram object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last added value to be returned.
- ⑩ NUM represents the number of summed values since the histogram was created or reset.
- ⑩ SUM returns the total sum of values.
- ⑩ SUM2 returns the square sum of values.

virtual void resetValues()

Resets the histogram, flushing its internal log. The count of collected values and samples are set to zero. The lower and higher bounds are reset so as they bounce to the next collected value when available.

virtual void sample()

Performs a sampling operation on the histogram.

virtual void result()

This method is designed to compute the statistical object's final value, just before it is inquired. At this class level, this method does nothing.

StHistogramGroup (extends StObjectGroup)

PURPOSE

Instances of this class gather multiple histograms under a common name to perform grouped computation. Held histograms are referred to by an index into an internal array. An extra cell is automatically reserved to store the group's global histogram.

The global histogram index is always #0. Individual histogram indices range from a given index base to the sum of this base with the number of held histograms. Negative bases are allowed.

CONSTRUCTORS

StHistogramGroup(const char *name, const char *indexName, int nitems, int nbins, double leftb, double rightb, StHistAdjustMode mode =MULTIPLY, int iBase =0, StProtoPilot *pilot =0, int pflags =0)

StHistogramGroup(const char *name, const char *indexName, int nitems, int nbins, int leftb, int rightb, StHistAdjustMode mode =MULTIPLY, int iBase =0, StProtoPilot *pilot =0, int pflags =0)

Creates an histogram group. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

indexName is the logical name of the index that will be used in composing the held histograms names.

nitems is the number of histograms the group constructor will create.

Each histogram from the set is created either with floating-point or integer bounds, namely **leftb** and **rightb**. The number of counting bins is specified by **nbins**. For integer bounded histograms, the right bound and the number of counting bins are adjusted to obtain an integer bin size.

Histograms can automatically adjust themselves to the actual range of entered values. The expected behavior when a value falls outside the current range of an histogram is selectable by an adjustment **mode** setting:

- Ⓢ MULTIPLY causes the histogram range to be adjusted by successive multiplications by 2, either to the left or to the right side according to the bound that is overshooted. This is the default mode.
- Ⓢ GARBAGE prevents the histogram range to be adjusted, all the values falling outside this range are collected in the leftmost and rightmost bins.

iBase is the the lowest index value for the group. Indices passed to methods requiring them will be scaled to take in account this threshold. For instance, passing an index base of -2 for a 4-elements group allows using indices -2, -1, 0 (global histogram), 1 and 2.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot)

or master (if **pilot** extends **StProtoBackPilot**). If **NULL**, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StHistogramGroup(**StProtoExportMessage** *pex, **StProtoPilot** *pilot, const char *indexName, int nitems, int nbins, double leftb, double rightb, **StHistAdjustMode** mode =MULTIPLY, int iBase =0)

StHistogramGroup(**StProtoExportMessage** *pex, **StProtoPilot** *pilot, const char *indexName, int nitems, int nbins, int leftb, int rightb, **StHistAdjustMode** mode =MULTIPLY, int iBase =0)

Creates a slave group object, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the **protoExport()** method being called for a backend object.

indexName, **nitems**, **nbins**, **leftb**, **rightb**, **mode** and **iBase** have the same meaning than previously.

StScaler (extends StObject)

PURPOSE

StScaler instances are statistical measurement objects aimed at computing the ratio of a given value by an other. A scaler establishes a relationship between a *scaled* object and a *scaling* one.

This pure class should be extended by subclasses in order to implement the way the scaling is done.

CONSTRUCTORS

StScaler(const char *name, StObject *scaledObject, StValueType vtype =VAL, StProtoPilot *pilot =0, int pflags =0)

Creates a scaler object. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

scaledObject is a pointer to a valid statistical object to scale.

vtype is used to specify which value type should be obtained from the scaled object to compute the ratio. Valid types are defined by the scaled object's `getValue()` method.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StScaler(StProtoExportMessage *pex, StProtoPilot *pilot, StObject *scaledObject, StValueType vtype =VAL)

Creates a slave scaler, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the `protoExport()` method being called for a backend object.

scaledObject and **vtype** have the same meaning than previously.

METHODS

virtual double getValue(StValueType vtype =VAL)

Gets the object's value. Because a scaler object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last entered value to be returned. This operation implies the retrieval of the current scaled and scaling values to compute the ratio.
- ⑩ NUM represents the number of times the ratio has been computed.

StTimeScaler (extends StScaler)

PURPOSE

StTimeScaler instances are statistical measurement objects aimed at computing the division of a scaled object's value by the multiplication of the current clock value by a given factor.

Each time the inherited StScaler::getValue() method is invoked for a time scaler, the following actions take place:

- ⑩ First, the current value of the scaled object is obtained by a call to its getValue() method, with the appropriate value type.
- ⑩ Then, the divisor is computed by multiplying the current clock value by the object's time factor.
- ⑩ Finally, the current time scaler's value is set to the result of dividing the scaled object's value by the divisor.

CONSTRUCTORS

StTimeScaler(const char *name, StObject *scaledObject, StValueType vtype =VAL, double timeFactor=1.0, StProtoPilot *pilot =0, int pflags =0)

Creates a time scaler object. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

scaledObject is a pointer to a valid statistical object to scale.

vtype is used to specify which scaled value type should be obtained from the object to compute the ratio. Valid types are defined by the scaled object's getValue() method.

timeFactor is the multiplication factor applied to the clock value.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StTimeScaler(StProtoExportMessage *pex, StProtoPilot *pilot, StObject *scaledObject, StValueType vtype =VAL, double timeFactor =1.0)

Creates a slave scaler, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the protoExport() method being called for a backend object.

scaledObject, **vtype** and **timeFactor** have the same meaning than previously.

StObjectScaler (extends StScaler)

PURPOSE

StObjectScaler instances are statistical measurement objects aimed at computing the ratio of a *scaled* object's value by the value of a *scaling* object.

CONSTRUCTORS

StScaler(const char *name, StObject *scaledObject, StObject *scalingObject, StValueType scaledVtype =VAL, StValueType scalingVtype =VAL, StProtoPilot *pilot =0, int pflags =0)

Creates an object scaler. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

scaledObject is a pointer to a valid statistical object, that will be scaled by the scaling object pointed to by **scalingObject**.

scaledVtype and **scalingVtype** are respectively used to specify which *scaled* and *scaling* value types should be obtained from the objects to compute the ratio. Valid types are:

- ⑩ VAL, that is the last stored value.
- ⑩ NUM, representing the number of aggregated values since the object was created or reset.
- ⑩ SUM, that is the global aggregated value since the beginning of the measure.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StScaler(StProtoExportMessage *pex, StProtoPilot *pilot, StObject *scaledObject, StObject *scalingObject, StValueType scaledVtype =VAL, StValueType scalingVtype =VAL)

Creates a slave scaler, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the protoExport() method being called for a backend object.

scaledObject, **scalingObject**, **scaledVtype** and **scalingVtype** have the same meaning than previously.

StFilter (extends StObject)

PURPOSE

StFilter instances can be applied to any statistical object to build derivative or first order filter on any measurement performed by such object.

The filter class does not implement the update trigger that should invoke the StFilter::update() method periodically, according to the time reference stored in the global variable Clock. An example of a filter management class named SxSampler can be found in FROGS' SIMEX simulation kernel.

CONSTRUCTORS

StFilter(const char *name, StObject *filteredObject, ITime dtUpdate, StValueType vtype, int logSize =2, StProtoPilot *pilot =0, int pflags =0)

Creates a filter. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

filteredObject is a pointer to the statistical object to filter each **dtUpdate** time units.

vtype is used to specify which object's value type should be filtered. Valid types are defined by the filtered object's getValue() method.

logSize specifies the number of computed values that should be kept in the object's internal log. The log is circular, causing the oldest values to be overwritten when the overflow limit is reached. Each cell of the log contains the value computed during an update, up to **logSize * dtUpdate** back in time.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends StProtoFrontPilot) or master (if **pilot** extends StProtoBackPilot). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the StProto interface.

StFilter(StProtoExportMessage *pex, StProtoPilot *pilot, StObject *filteredObject, ITime dtUpdate, StValueType vtype =VAL, int logSize =2)

Creates a slave filter, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the protoExport() method being called for a backend object.

filteredObject, **dtUpdate**, **vtype** and **logSize** have the same meaning than previously.

METHODS

virtual void update()

Obtains the current filtered object's value, then stores it into the filter's log. This method should be called on a periodical basis, according to the time reference value given by the global variable **Clock**. The time elapsed between two invocations of this method should be equal to the **dtUpdate** parameter passed to the filter constructor.

virtual void resetValues()

This method leads to a null-effect. A filter cannot be reset.

virtual void add(double value)

Stores **value** in the filter's log. The lower and upper object's value bounds are maintained by this method. The current time value (i.e. **Clock** contents) is remembered.

virtual double getValue(StValueType vtype =VAL)

Gets the object's value. Because a filter object actually stores different values, **vtype** is used to specify the kind of desired value on return. At this class level, the method can be asked the following value types:

- ⑩ VAL causes the last stored value to be returned.
- ⑩ NUM represents the total number of measures performed.
- ⑩ DMES is derivative over the full range of the filter.

virtual void derive(ITime dt)

Computes and returns the derivative of the value serie for a time interval of **dt**. **dt** must be inside the range **dtUpdate** .. **logSize * dtUpdate**. If **dt** is outside of this range, a value for **dt** is taken as the closest of these bounds; otherwise, **dt** is rounded to the closest multiple of **dtUpdate**.

virtual void sift()

Computes and returns the current value sifted through a first order filter of time constant **dt**. **dt** must be inside the range **dtUpdate** .. **logSize * dtUpdate**. If **dt** is outside of this range, a value for **dt** is taken as the closest of these bounds; otherwise, **dt** is rounded to the closest multiple of **dtUpdate**.

StTimeGraph (extends StFilter)

PURPOSE

This class implements a measurement object designed to grasp the temporal evolution of a given statistic object. A time graph can be connected to a plotter drawing its graphical representation through the **StProto** interface.

CONSTRUCTORS

StTimeGraph(const char *name, StObject *filteredObject, ITime dtUpdate, StValueType vtype, int logSize =NTGVALUEDEF, StProtoPilot *pilot =0, int pflags =0)

Creates a filter. **name** is a null-terminated character string identifying the new object; NULL is an acceptable value, preventing the anonymous object from being visible from the display front-end.

filteredObject is a pointer to the statistical object to filter each **dtUpdate** time units.

vtype is used to specify which object's value type should be filtered. Valid types are defined by the filtered object's **getValue()** method.

logSize specifies the number of computed values that should be kept in the object's internal log. The log is circular, causing the oldest values to be overwritten when the overflow limit is reached. Each cell of the log contains the value computed during an update, up to **logSize * dtUpdate** back in time. The default value, NTGVALUEDEF, is currently 500.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StTimeGraph(StProtoExportMessage *pex, StProtoPilot *pilot)

Creates a slave time graph, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the **protoExport()** method being called for a backend object.

StStateDiagram (extends StObject)

PURPOSE

This class implements a backend object designed to log and export state information to a drawing program. A state diagram can be connected to a plotter drawing the transitions between states through the **StProto** interface.

Basically, a state diagram is used to log state transitions along with the current **Clock** value at which they occur. It makes easy to propagate the state changes to a graphical display for monitoring those transitions.

Known states are stored in an array of strings. A state diagram object's current value is the array index of the last state entered by a call to the **add()** method.

CONSTRUCTORS

StStateDiagram(const char *name, int nstates, const char *sarray, int logSize =NSTVALUEDEF, StProtoPilot *pilot =0, int pflags =0)

Creates a state diagram. **name** is a null-terminated character string identifying the new object; **NULL** is an acceptable value, preventing the anonymous object from being visible from the display front-end.

sarray is a pointer to an array of null-terminated strings naming the states the object can go through. **nstates** is the number of valid cells in the array. **nstates** can be negative or null, in which case the valid state names will remain unidentified until **defineStates()** is called for the object. If **nstates** is negative or null, **sarray** remains unused and thus can be passed as **NULL**.

logSize specifies the number of states that should be kept in the object's internal log. The log is circular, causing the oldest values to be overwritten when the overflow limit is reached. The default value, **NSTVALUEDEF**, is currently 100.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If **NULL**, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StStateDiagram(const char *name, int logSize =NSTVALUEDEF, StProtoPilot *pilot =0, int pflags =0)

Creates a state diagram, with no defined states yet. The existing states should be defined by a subsequent invocation of the **defineStates()** method with the proper arguments.

name is a null-terminated character string identifying the new object; **NULL** is an acceptable value, preventing the anonymous object from being visible from the display front-end.

logSize specifies the number of states that should be kept in the object's internal log. The log is circular, causing the oldest values to be overwritten when the overflow limit is reached. The default value, NSTVALUEDEF, is currently 100.

pilot is the address of the object's protocol pilot. The current object's operating mode is determined by the nature of its pilot, whether slave (if **pilot** extends **StProtoFrontPilot**) or master (if **pilot** extends **StProtoBackPilot**). If NULL, the object has no interface channel.

pflags is a set of protocol flags such as defined by the **StProto** interface.

StStateDiagram(StProtoExportMessage *pex, int logSize =NSTVALUEDEF StProtoPilot *pilot =0)

Creates a slave state diagram, whose specifications are given in the export message pointed to by **pex**. **pilot** is the address of the new object's protocol pilot, that should belong to the display front-end application.

An export message is received as a result of the **protoExport()** method being called for a backend object.

logSize specifies the number of states that should be kept in the object's internal log. The log is circular, causing the oldest values to be overwritten when the overflow limit is reached. The default value, NSTVALUEDEF, is currently 100.

METHODS

void defineStates(int nstates, const char *const *sarray)

Defines the object's states. This method should be called once, whenever no states were passed to the constructor.

sarray is a pointer to an array of null-terminated strings naming the states the object can go through. **nstates** is the number of valid cells in the array.

int getNStates() const

Returns the number of defined states.

const char *getStateName(int nth) const

Returns the **nth** state name from the array. **nth** is zero-based.

virtual void resetValues()

Resets the object. The count of collected values and samples are set to zero. The lower and higher bounds are reset so as they bounce to the next collected value when available.

virtual void add(double stateno)

Enters a new state. **stateno** is converted to an integer index referring to the name of the new state into the state array. This index is remembered into the object's log.

If the state diagram is exported by the **StProto** object protocol, a message is immediately sent to the display front-end propagating the entered state value.

PURPOSE

This class implements a basic export protocol feature for statistical object's to send their current results and/or values to a display front-end. **StProto** is the superclass of all exportable objects defined in the STATOBJ library. This class is documented in the hope it will help implementing additional statistical objects.

The purpose of this class is to define a conventional way of sending and receiving data between objects communicating remotely from a computation back-end and a display front-end. It makes subclassed objects exhibit a simple interface to accomplish this task in a well-defined, centralized manner.

The **StProto** class does not define the actual mean of exchanging data, such as using sockets, shared memory, files or simple function calls. It rather defines a protocol to follow for each side involved in the communication. The actual i/o operations are performed by protocol pilots supplied by the programmer. Messages are composed of a type identifier and an optional dynamically sized data buffer. FROGS SIMEX's **SxTcp** illustrates how to implement a TCP-based communication channel one can use to connect protocol pilots.

It is important to understand that the object classes involved in the communication (i.e. from back-end and front-end code) should both extend the **StProto** class. This is the reason why all exportable objects defined in this library inherit the **StProto** object, and have two construction modes, each defining a distinct behavior. A statistical object has a *master* operation mode in which it actually performs computation and produces results. It also has a *slave* operation mode receiving data produced by a *master* object, logging it for display purpose. The slave object should be seen as an image of the master object used by a remote display code. An illustration of such code can be found in the TKPLOT library, that conforms to the **StProto** protocol for building a graphical plotter of STATOBJ classes.

A protocol object has two major states, whether it is *exported* or not. To be eligible for export, a protocol object must have a valid name. A protocol object can be explicitly hidden, preventing any further export. When exported, a protocol object can be displayed by the front-end or not.

Each protocol object has a unique internal identifier passed along with the messages it sends and receives. This identifier keeps both sides in sync while allowing to destroy a back-end object without jeopardizing the application.

The major steps of the protocol are:

- ⑩ A protocol object can be given a name at creation time, or by using the `protoSetName()` method. If the object does not need to be exported, it can be hidden using `protoSetHidden()` or given a NULL name, which leads to the same effect.
- ⑩ Once the communication channel is established between the back-end and the front-end code (maybe in the same application), a master object should be exported by sending an appropriate message to the front-end through the

`protoExport()` method. It should be noted that statistical objects tend to provide this code in their respective `protoInit()` method. When received by the front-end pilot, the message should cause a slave counterpart object to be created.

- ⑩ Both sides can send messages to their respective counterpart using the `protoSend()` method.
- ⑩ As messages are received and dispatched by the protocol pilots, the `protoProcess()` method is invoked to forward the input to the recipient protocol object.

Messages can be of six types, all extending the `StProtoMessage` type:

- ⑩ Export messages relay the characteristics of a master object to the display front-end, expecting a slave counterpart object to be built as a result of processing such messages. Export messages contain the master object's name and an optional type field.
- ⑩ Unexport (deletion) message is sent to the peer whenever an object is destroyed, whichever side deleted it.
- ⑩ Display messages are used to forward an object's display status to the peer. A boolean value indicates whether the object has been put to or removed from display.
- ⑩ Informational messages are sent using the `protoInfo()` method to the peer object. They should relay replies to general information requests back to the peer. This kind of information is passed as a character string, instead of a raw data buffer.
- ⑩ Breakpoint messages can flow on the communication channel as a result of setting or clearing internal breakpoints one can set on statistical objects supporting this feature. The `StProto` class implements a simple breakpoint management scheme that is used by time graphs and state diagrams to provide graphical breakpoint capabilities to the external plotter.
- ⑩ Other messages are defined by the application code.

CONSTRUCTORS

`StProto(const char *name =0, StProtoPilot *pilot =0, int pflags =0)`

Creates a master protocol object. **name** is a null-terminated character string identifying the new object; `NULL` is an acceptable value, preventing the anonymous object from being visible from the display front-end. A new named object is implicitly exportable, unless `STPROTO_HIDDEN` appears in the **pflags** parameter.

pilot is a pointer to the protocol pilot that is in charge of handling the communication for the back-end side. `NULL` is an acceptable value telling the object is not currently exportable. Setting this information may be postponed until a subsequent call to `protoSetPilot()` is issued for the object.

pflags should be zero, or `STPROTO_HIDDEN` if the object should not be exported to the display front-end.

`StProto(const StProtoExportMessage *pex, StProtoPilot *pilot)`

Creates a slave protocol object. **pex** is a pointer to the information block passed by the master object to its slave, defining the characteristics of the image it should build.

pilot is a pointer to the protocol pilot that is in charge of handling the communication for the front–end side. Because a slave object makes no sense without a master creating it, it should always have a valid pilot. However, setting this information may be postponed until a subsequent call to `protoSetPilot()` is issued for the object.

METHODS

`StProtoHandle protoGetHandle() const`

Returns the object's protocol handle. A valid handle is set whenever a master object is exported, or a slave object is created using the information stored in the export message. This handle is unique among all objects.

`void protoSetPilot(StProtoPilot *pilot)`

Sets the **pilot** information for the target object.

`void protoSetName(const char *name)`

Sets the object's external name. If **name** is NULL, the object becomes unexportable, unless it has already been exported. `protoSetExportable()` needs to be called next in order to mark the object as exportable.

`void protoSetExportable()`

Mark the object as exportable. This method should be invoked to validate the ability to export the object after a name has been set through `protoSetName()`, and before `protoInit()` is called.

`const char *protoGetName() const`

Returns the object's name, or NULL if none.

`void protoSend(int mtype, STProtoMessage *pm =0, int msize =0)`

Sends a message to the peer object. The data is relayed by the protocol pilot currently active for the object through its `send()` method. This method leads to a null effect if the current object has no defined pilot.

mtype is the message's type identifier serving as a tag for the receiver to interpret it. This identifier should be strictly positive.

pm is a pointer to an optional data buffer whose size is given by **msize**. If **pm** is NULL, the recipient gets notified of a basic `StProtoMessage` arrival of type **mtype**. **msize** should be zero whenever **pm** is invalid.

When received, the message is dispatched to the peer object identified by its protocol handle. Related master and slave objects share the same handle.

`int protolsDisplayed() const`

`int protolsConcealed() const`

Returns the object's display status. Whenever the display front-end has informed the back-end that the current object has come to display (through the `protoDisplay()` method), `protolsDisplayed()` returns true. Conversely, `protolsConcealed()` returns true when the previous assertion is false.

`int protolsExportable() const`

`int protolsHidden() const`

Tells whether the object is exportable or not. An object must have a valid name, and should not have been explicitly hidden to be exportable.

`int protolsExported() const`

Returns the object's export status. Whenever the object has already been exported to the display front-end, this method returns true.

`void protoExport(StProtoExportMessage *pex, int msize)`

Sends an export notification message of type `STPROTO_EXPORT_OBJECT` to the peer. The message is relayed by the protocol pilot currently active for the object through its `send()` method. This method is reserved for use with the master object role and is typically called on behalf of `protoInit()`.

pex is a pointer to an optional information buffer whose size is given by **msize**. If **pex** is NULL, the recipient gets notified of a basic `StProtoMessage` arrival of type `STPROTO_EXPORT_OBJECT`. **msize** should be zero whenever **pex** is invalid.

The message is sent to the peer only if all of the following conditions are met:

- ⑩ The current object has a valid name, and was not explicitly hidden (i.e. is exportable);
- ⑩ A valid back-end protocol pilot has been set for the current object;
- ⑩ The object has not already been exported by a previous call to `protoExport()`.

`void protoDestroy()`

Sends an unexport notification message of type `STPROTO_UNEXPORT_OBJECT` to the peer. The message is relayed by the protocol pilot currently active for the object through its `send()` method. This method can be called either for the master or slave objects.

The message is dispatched by the recipient's protocol pilot to the proper destination, whether the deletion was requested from the master or slave side. The following steps may be taken as a result of receiving this kind of notification:

- ⑩ If the message is sent from the master object side, the notification should be processed by the `StProtoFrontPilot::dispatch()` method, which in turn calls the virtual `StProtoFrontPilot::destroyDisplay()` method passing the corresponding slave object address. The latter is expected to handle the deletion for the front-end side of the protocol, doing all the necessary housekeeping chores in order to remove the destroyed object from the current display as needed. The `StProtoFrontPilot::destroyDisplay()` method is pure, thus must be implemented by subclasses.

- ⑩ If the message is sent from the slave object side, the notification should be processed by the `StProtoBackPilot::dispatch()` method, which in turn calls the virtual `StProtoFrontPilot::destroyObject()` method passing the corresponding slave object address. The latter is expected to handle the deletion for the front-end side of the protocol. Its base implementation causes the slave object to be deleted (i.e. the C++ *delete* operator is called for it), but can be specialized by subclasses to perform extended deletion processing.

`void protoDisplay()`

Sends a display notification message of type `STPROTO_DISPLAY_TOGGLE` to the peer, along with a `StProtoDisplayMessage` buffer. This type of message contains a boolean value telling the new display status, which in this case is set to true. The message is relayed by the protocol pilot currently active for the object through its `send()` method. This method is reserved for the slave object role.

This kind of notification is used by the front-end code to inform the back-end application that the corresponding slave object has just been put on display. As a result of receiving this message, the master object should start sending data to its peer.

`void protoConceal()`

Sends a display notification message of type `STPROTO_DISPLAY_TOGGLE` to the peer, along with a `StProtoDisplayMessage` buffer. This type of message contains a boolean value telling the new display status, which in this case is set to false. The message is relayed by the protocol pilot currently active for the object through its `send()` method. This method is reserved for the slave object role.

This kind of notification is used by the front-end code to inform the back-end application that the corresponding slave object has just been removed from display. As a result of receiving this message, the master object should stop sending data to its peer.

`void protoInfo(int mtype, const char *info, int slen == -1)`

Sends an information message to the peer object. The data is relayed by the protocol pilot currently active for the object through its `send()` method. This method is similar to the `protoSend()` method, except the message body that must be a character string instead of a raw data buffer. The peer may assume the received buffer is always a null-terminated character string.

mtype is the message's type identifier serving as a tag for the receiver to interpret it. This identifier should be strictly positive.

info is a pointer to character string whose length is given by **slen**. If **slen** is negative, the actual string length is determined by a call to `strlen()`. This means that **info** may not be null-terminated, provided that **slen** has a correct value. However, the received buffer will always be null-terminated. There is no static limit imposed to the string length.

When received, the message is dispatched to the peer object identified by its protocol handle. Related master and slave objects share the same handle.

`virtual void protoInit()`

Initializes a master object for export. This method should be implemented by subclasses to call the `protoExport()` method appropriately. This is not an internal service, but rather a conventional location where to put an object's export directive. This method should be called for each object extending the `StProto` superclass after the communication channel has been established between the back-end and the front-end sides.

The default implementation does nothing.

`virtual void protoSignal(int signo)`

Processes a `StProto` signal sent to the current object. This service is not currently used by the `StProto` interface. It is rather defined for classes extending the `StProto` superclass as a conventional signal handler. If of any use, subclasses should implement this method in order to catch existing signals, and behave appropriately.

signo can be:

- ⑩ `STPROTO_SIGBREAK` telling that a breakpoint has been reached. The signal handler should take the appropriate steps to react to this condition. For instance, statistical objects from the FROGS executive suspend the simulation process upon receipt of such signal.
- ⑩ Any other user-defined signal.

`virtual void protoProcess(int mtype, const StProtoMessage *pm, int msize)`

This method should be implemented by subclasses to process the messages received from the peer object.

mtype is the message's type identifier serving as a tag for the receiver to interpret it. This identifier is strictly positive.

pm is a pointer to a data buffer whose size is given by **msize**. A valid buffer is always passed to the method, even if the peer gave no message body. In such a case, the address of a basic `StProtoMessage` is passed.

The layout of the `StProtoMessage` struct is as follow:

```
struct StProtoMessage {  
    StProtoHandle handle;  
    u_long seqNum;  
}
```

handle is the sender object's identifier. This value is local to the back-end side, thus remote from the front-end side.

seqNum is the message sequence number computed by the sender. This value can be used to uniquely identify a message in a flow. All back-end objects share a single counter for numbering.

The base implementation of `protoProcess()` does nothing.

`void protoSetBreak(double threshold)`

Records a breakpoint on the current object. The **StProto** interface maintains a simple breakpoint list on a per-object basis for subclasses implementing breakpoints on numerical values, such as **StTimeGraph** and **StStateDiagram**.

threshold is the numerical value the object should reach or cross for the breakpoint to be hit. The routine gracefully and silently ignores duplicate breakpoints with identical thresholds. Otherwise, a breakpoint for the given threshold is attached to the object's breakpoint list.

When run on behalf of a slave object, this method attempts to send a breakpoint notification to the master peer. This notification consists of a **STPROTO_BREAK_TOGGLE** message. The message body is a **StProtoBreakpoint** struct containing the given threshold and a boolean status telling whether the breakpoint should be set or removed. In this case, this boolean value is set to true. Upon receipt, the master object is expected to start monitoring the breakpoint condition and react as needed when it is reached or crossed.

Triggering breakpoints is not in the scope of the **StProto** superclass. Implementing the defined breakpoints is definitely a task devoluted to its subclasses, as they process events and state transitions. The superclass only maintains the breakpoint information.

void protoClrBreak(double threshold)

Removes a breakpoint from the current object. The **StProto** interface maintains a simple breakpoint list on a per-object basis for subclasses implementing breakpoints on numerical values, such as **StTimeGraph** and **StStateDiagram**.

threshold is the numerical value for which a breakpoint has been set using a previous call to **protoSetBreak()**. The routine gracefully and silently ignores inexistent breakpoints for the given threshold. Otherwise, the breakpoint for the given threshold is removed from the object's breakpoint list.

When run on behalf of a slave object, this method attempts to send a breakpoint notification to the master peer. This notification consists of a **STPROTO_BREAK_TOGGLE** message. The message body is a **StProtoBreakpoint** struct containing the given threshold and a boolean status telling whether the breakpoint should be set or removed. In this case, this boolean value is set to false. Upon receipt, the master object is expected to stop monitoring the breakpoint condition.

PURPOSE

This class implements the statistical objects interface protocol pilot for the STATOBJ library. One should first have a look to the discussion about the **StProto** class before reading the following information.

Protocol pilots are objects performing the actual i/o and dispatching tasks to establish a communication channel between related master and slave objects.

The **StProtoPilot** class is a pure superclass that must be subclassed to implement the actual protocol processing.

CONSTRUCTOR

StProtoPilot(**StProtoPilotType** ptype)

Creates a protocol pilot. **ptype** indicates the kind of service provided by the object, whether **StProtoBACKEND** for a back-end pilot managing master objects, or **StProtoFRONTEND** for a front-end pilot managing slave objects.

METHODS

StProto *search(**StProtoHandle** handle)

Searches for a protocol object identified by **handle** and managed by the current pilot, then returns its address when found. Otherwise, returns **NULL**.

void remap(**StProto ***object, **StProtoHandle** handle)

Sets an object's internal identifier to **handle**. This method should be used with extreme care as it may jeopardize the library consistency if misused. Anyway, it is sometimes useful when implementing specific protocol pilots.

A unique identifier is set for each object when it is exported. This method allows to change it for internal management purposes.

virtual int dispatch(**int** mtype, **const void ***mbuf, **int** msize)

Dispatches a message received from the i/o channel to the proper recipient. At this class level, the method searches for the object identified by the handle received in the message body which is interpreted as a buffer of type **StProtoMessage**.

If the object is not found, the method returns a boolean false value.

If the object is found and **mtype** is equal to **STPROTO_BREAK_TOGGLE**, the breakpoint notification is processed locally. Otherwise, the message is passed to the **StProto::protoProcess()** method for the recipient object. The method always returns a boolean true value, indicating that the message has been successfully dispatched.

This method should be called by the underlying i/o management code in order to dispatch the received messages.

```
virtual void send(int mtype, const void *mbuf, int msize) =0
```

This pure virtual method is called to send messages to the peer object. It should be implemented by the subclassed protocol pilot to perform the appropriate tasks in order to relay the message identifier and data to the remote peer.

StProtoBackPilot (extends StProtoPilot)

PURPOSE

This pure class implements the STATOBJ protocol pilot handling the backend side of the communication between master and slave statistical objects.

CONSTRUCTOR

StProtoBackPilot()

Creates a backend protocol pilot. This object should be used to handle the communication for the master statistical objects.

virtual int dispatch(int mtype, const void *mbuf, int msize)

Dispatches a message received from the i/o channel to the proper recipient. At this class level, the method traps messages of type STPROTO_DESTROY_OBJECT, searching for the object identified by the handle received in the message body which is interpreted as a buffer of type StProtoMessage. If the object is not found, the method returns a boolean false value. Otherwise, destroyObject() is called and given the address of the target object.

If **mtype** is different from STPROTO_DESTROY_OBJECT, the message is simply relayed to StProtoPilot::dispatch (). The method always returns a boolean true value whenever the message has been successfully dispatched.

This method should be called by the underlying i/o management code in order to dispatch the received messages.

virtual void destroyObject(StProto *object)

Destroys **object** by calling the C++ *delete* operator. This method should be implemented by subclasses whenever additional housekeeping chores should be performed upon master object deletion by the frontend application.

StProtoFrontPilot (extends StProtoPilot)

PURPOSE

This pure class implements the STATOBJ protocol pilot handling the front–end side of the communication between slave and master statistical objects.

CONSTRUCTOR

StProtoFrontPilot()

Creates a front–end protocol pilot. This object should be used to handle the communication for the slave statistical objects.

virtual int dispatch(int mtype, const void *mbuf, int msize)

Dispatches a message received from the i/o channel to the proper recipient. At this class level, the action taken depends on **mtype**'s value:

- ⑩ if **mtype** is STPROTO_EXPORT_OBJECT, createDisplay() is called and given the export message's address and size. This method should return a valid new StProto object handling the slave side of the protocol, or NULL if an error occurred. In the latter case, the method returns a boolean false value.
- ⑩ if **mtype** is STPROTO_UNEXPORT_OBJECT and the target object is found, destroyDisplay() is called and given the address of that object. If the target object cannot be found, the method returns a boolean false value.
- ⑩ If **mtype** has another value from the above, the message is simply relayed to StProtoPilot::dispatch ().

StProtoFrontPilot::dispatch() always returns a boolean true value whenever the message has been successfully dispatched.

This method should be called by the underlying i/o management code in order to dispatch the received messages.

virtual StProto *createDisplay(const StProtoExportMessage *pex, int msize)

This pure method must be implemented by subclasses to create a slave counter–part to the master object whose export has just been notified through an incoming STPROTO_EXPORT_MESSAGE.

The information needed for creation should be contained in the export struct of size **msize** pointed to by **pex**.

This method should return a valid slave object on success, otherwise NULL on failure.

virtual void destroyDisplay(StProto *object) =0

This pure method must be implemented by subclasses to perform the needed housekeeping chores (such as removing the object from the display) upon slave object deletion by the backend application.

StNumericLaw

PURPOSE

This class implements the basic object for all numeric laws and number generators. Whilst it can be instantiated directly, it is usually subclassed to implement specific behaviors.

`StNumericLaw(double x =0.0)`

Creates a number generator with **x** as seed.

`int iget()`

Calls `get()` to obtain the next value and returns it as an integer.

`virtual double get()`

Draws and returns the next value. At this class level, this method always returns the initial seed.