iqr User-defined Types Documentation

Ulysses Bernardet
bernuly@gmail.com

January 7, 2008

Contents

Concepts

This document gives an overview on how to write your own neurons, synapses, and modules.

The first part will explain the basic concepts, the second part will provide walk-through example implementations, and the appendix lists the definition of the relevant member variables and functions for the different types.

iqr does not make a distinction between types that are defined by the user, and those that come with the installation; both are implemented in the same way.

The base-classes for all three types, neurons, synapses, and modules, are derived from `ClsItem` (fig. 1.1).

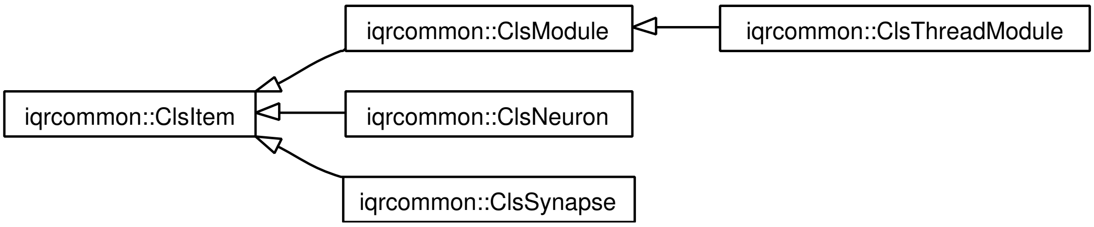


fig. 1.1: Class diagram for types

The classes derived from `ClsItem` are in turn the parent classes for the specific types; a specific neuron type will be derived from `ClsNeuron`, a specific synapse type from `ClsSynapse`. In case of modules, a distinction is made between threaded and non-threaded modules. Modules that are not threaded are derived from `ClsModule`, threaded ones from `ClsThreadModule`.

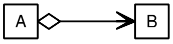
The inheritance schema defines the framework, in which

- parameters are defined
- data is represented and accessed
- input, output, and feedback is added

All types are defined in the namespace `iqrcommon`.

1.1 Object model

To write user-defined types, it is vital to understand the object model iqr uses. Figure 1.2 shows a simplified version of the class diagram for an iqr system.

The lines connecting the objects represent the relation between the objects. The arrow heads and tails have a specific meaning:  stands for a relation where **A** contains **B**.

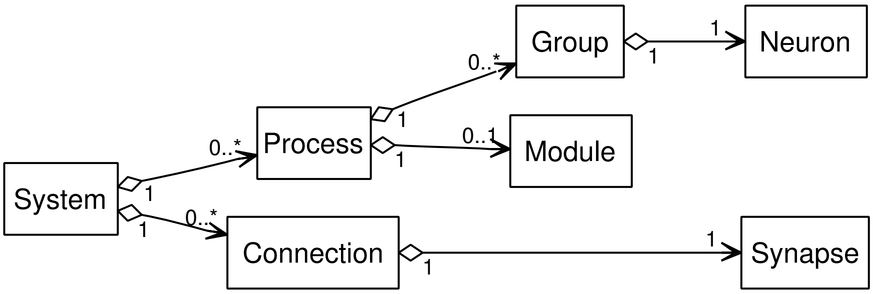


fig. 1.2: Simplified class diagram of an iqr system

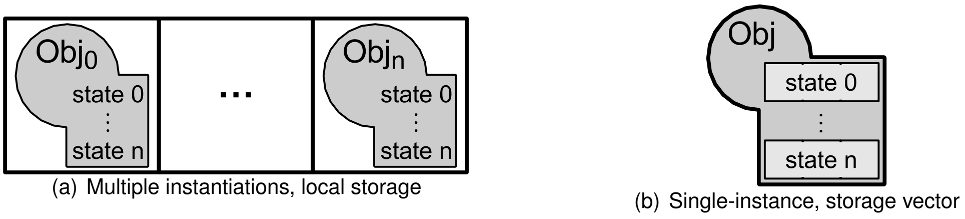
The multiplicity relation between the objects is denoted by the numbers at the start and the end of the line. E.g. a system can have 0 or more processes ($0 \ldots *$ near the arrow head). A process in turn can only exist in one system (1 near the $\Diamond$).

On the phenomenological level, to a user, a group consists of a $n \geq 1$ neuron(s), and a connection of $n \geq 1$ synapse(s). In terms of the implementation though, as can be seen in fig. 1.2, a group contains only **one instance** of a neuron object, and a connection only **one instance** of a synapse object. This is independent of the size of the group or the number of synapses in a connection.

1.2 Data representation

In the concept of iqr, neurons and synapses do have individual values for parameters like the membrane potential or weight associated to them. In this document, type-associated values, that change during the simulation, are referred to as “states”.

There are essentially two ways in which individual value association can be implemented: • multiple instantiations of objects with local data storage (fig. 1.3a), or • single-instance with states for individual “objects” in vector like structure (fig. 1.3b).



(a) Multiple instantiations, local storage

(b) Single-instance, storage vector

fig. 1.3:

For reasons of efficiency, iqr uses the single-instance implementation (see also fig. 1.2). For authors of types, the drawback is a somewhat more demanding handling of states and update functions. To compensate for this, great care was taken to provide an easy to use framework for writing types.

1.2.1 The `StateArray`

The data structure used to store states of neurons and synapses is the `StateArray`. The structure of a `StateArray` is depicted in fig. 1.4. It is used like a two-dimensional matrix, with the first dimension being the time and the second dimension the index of the individual item (neuron or synapse). Hence `StateArray`$[t][n]$ is the value of item n at time t.

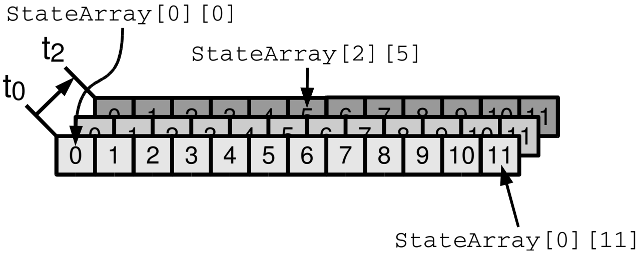


fig. 1.4: Concept of `StateArray`

Internally `StateArrays` make use of the **valarray** class from the standard C++ library.

To extract a valarray containing the values for all the items at time $t - d$ use

```
std::valarray<double> va(n);
va = StateArray[d];
```

The convention is that `StateArray[0]` is the current valarray, whereas `StateArray[2]` denotes the valarray that is 2 simulation cycles back in time.

Valarrays provide a compact syntax for operations on each element in the vector, the possibility to apply masks to select specific elements, and a couple of other useful features. A good reference on the topic is Josuttis (1999).

1.2.2 States in neurons and synapses

The subsequently discussed functions are the main functions used to deal with states. Additional functions for the individual types are listed in the appendix.

Adding an internal state to neurons and synapses is done via the wrapper class `ClsStateVariable`:

```
ClsStateVariable *pStateVariable;
pStateVariable = addStateVariable("st" /* identifier */,
                                  "A state variable" /* visible name*/);
```

To manipulate the state we first extract the `StateArray`, where after we can address and change the state as described above:

```
StateArray &sa = pStateVariable->getStateArray();
sa[0][1] = .5;
```

The output state is a special state for neurons and synapses. For most neurons the output state will be the activity. The neuronal output state is used as input to synapses and modules. For synapses, the output state acts as an input to neurons. A neuron or synapse can only have one output state.

An output state is defined by means of the `addOutputState(...)` function:

```
ClsStateVariable *pActivity;
pActivity   = addOutputState("act" /*identifier*/,
                             "Activity" /*name*/);
```

States created in this framework are accessible in the GUI for graphing and saving.

The full description of these functions can be found in documentation for `ClsNeuron` (section 3.2 on page 25) and `ClsSynapse` (section 3.3 on page 28).

State related functions in neurons

The base-class for the neuron type automatically creates three input states: the excitatory, inhibitory, and modulatory. Therefore, you do not create any input state when implementing a neuron type. To access the existing ones, use the following functions, which return a reference to a `StateArray`:

```
StateArray &excitation = getExcitatoryInput();
StateArray &inhibition = getInhibitoryInput();
StateArray &modulation = getModulatoryInput();
```

The user is free as to which of these functions to use.

State related functions in synapses

Synapses also must have access to the input state, which is actually the output state of the **pre-synaptic** neuron. The implementation of the pre-synaptic neuron type thus defines the input state of the synapse.

To access the input state the function `getInputState()` is employed, which returns a pointer to a `ClsStateVariable`:

```
StateArray &synIn     = getInputState()->getStateArray();
```

To use feedback from the **post-synaptic** neuron use the `addFeedbackInput()` function:

```
ClsStateVariable *pApicalShunt;
pApicalShunt = addFeedbackInput("apicalShunt" /*identifier*/,
                                "Apical dendrite shunt" /*description*/);
StateArray &shunt     = pApicalShunt->getStateArray();
```

1.2.3 Using history

To be able to make use of previous states, i.e. to use the history of a state, you explicitly have to declare a history length when you create the `StateArray` using the `addStateVariable(...)`, `addOutputState(...)`, or `addFeedbackInput(...)` functions. The reference for the syntax is given in the appendix (neurons: 3.2.2, synapses: 3.3.2)

1.2.4 Modules and access to states

Unlike neurons and synapses, modules do not need to use internal states to represent multiple elements. Modules require to read states from neurons, and feed data into states of neurons. The functions provided for this purpose are using a naming schema that is module centered: data that is read from the group is referred to as “input from group”, data fed into a group is pointed to as “output to group”. The references for theses states will be set in the module properties of the process (see iqr Manual section 2.17 on page 35)

Defining the output to a group is done with the `addOutputToGroup(...)` function:

```
ClsStateVariable* outputStateVariable;
outputStateVariable = addOutputToGroup("output0" /*identifier*/,
                                       "Output 0 description" /*description*/);
StateArray &clsStateArrayOut = outputStateVariable->getStateArray();
```

Specifying input from a group into a module employs a slightly different syntax using the `StateVariablePtr` class:

```
StateVariablePtr* inputStateVariablePtr;
inputStateVariablePtr = addInputFromGroup("input0" /*identifier*/,
                                          "Input 0 description" /*description*/);
StateArray &clsStateArrayInput =
inputStateVariablePtr->getTarget()->getStateArray();
```

Once the `StateArray` references are created, the data can be manipulated as described above.

Please do not write into the input from group `StateArray`. The result might be catastrophic.

When adding output to groups, or input from groups, no size constraint for the state array can be defined. It is therefore advisable to either write the module in a way that it can cope with arbitrary sizes of state arrays, or to throw a `ModuleError` (see section 2.3.2 on page 18) in the `init()` function if the size criteria is not met.

Access protection

If a module is threaded, i.e. the access to the input and output states is not synchronized with the rest of the simulation, the read and write operations need to be protected by a mutex. The `ClsThreadedModule` provides the `qmutexThread` member class for this purpose. The procedure is to lock the mutex, to perform any read and write operations, and then to unlock the mutex:

```
 qmutexThread->lock();
 /*
   any operations that accesses the
   input or the output state
*/
 qmutexThread->unlock();
```

As the locked mutex is impairing the main simulation loop, as few as possible operations should be performed between locking and unlocking.

Failure to properly implement the locking, access, and unlocking schema will eventually lead to a crash of iqr.

1.3 Defining parameters

The functions inherited from `ClsItem` define the framework for adding parameters to the type. Parameters defined within this framework are accessible through the GUI and saved in the system file. To this end, iqr defines wrapper classes for parameters of type `double`, `int`, `bool`, `string`, and `options` (list of options).

Usage

The best way to use these parameters, is to define a pointer to the desired type in the header, and to instantiate the parameter-class in the constructor, using the `add[Double,Int,Bool,String,Options]Parameter` functions. The value of the parameter object can be retrieved at run-time by virtue of the `getValue()` function. Examples for the usage are given in sections 2.1, 2.2, and 2.3. The extensive list of these functions is provided in the documentation for the `ClsItem` class in section 3.1 on page 19.

1.4 Where to store the types

The location where iqr loads types from, is defined in the iqr settings NeuronPath, SynapsePath, and ModulePath (see iqr Manual section 2.2.1 on page 12). Best practice is to enable Use user defined nnn (where nnn stands for Neuron, Synapse, or Module), and to store the files in the location indicated by the Path to user defined nnn. As the neurons, synapses, and modules are read from disk when iqr starts up, any changes to the type, while iqr is running, has no effect; you will have to restart iqr if you make changes to the types.

Example implementations

2.1 Neurons

Header

Let us first have a look at the header file for a specific neuron type. As you can see in listing 2.1, the only functions that must be reimplemented are the constructor [11] and `update()` [13]. Hiding the copy-constructor [17] is an optional safety measure. Lines [20-21] declare pointers to parameter objects. Line [24] declares the two states of the neuron.

Listing 2.1: Neuron header example

```
 1 #ifndef NEURONINTEGRATEFIRE_HPP
 2 #define NEURONINTEGRATEFIRE_HPP
 3 
 4 #include <Common/Item/neuron.hpp>
 5 
 6 namespace iqrcommon {
 7 
 8     class ClsNeuronIntegrateFire : public ClsNeuron
 9     {
10     public:
11         ClsNeuronIntegrateFire();
12 
13         void update();
14 
15     private:
16         /* Hide copy constructor. */
17         ClsNeuronIntegrateFire(const ClsNeuronIntegrateFire&);
18 
19         /* Pointers to parameter objects */
20         ClsDoubleParameter *pVmPrs;
21         ClsDoubleParameter *pProbability, *pThreshold;
22 
23         /* Pointers to state variables. */
24         ClsStateVariable *pVmembrane, *pActivity;
25     };
26 }
27 
28 #endif
```

Source

Next we'll walk through the implementation of the neuron, which is shown in listing 2.2. Line [4-5] defines the precompile statement that iqr uses to identify the type of neuron (see section 3.2.3 on page 27). In the constructor we reset the two `StateVariables` [9,10]. On line [12-38] we instantiate the parameter objects (see section 3.1 on page 19), and at the end of the constructor we instantiate one internal state [41] with `addStateVariable(...)`, and the output state [42] with `addOutputState(...)`. As for all types, the constructor is only called once, when iqr starts, or the type is changed. The constructor is **not** called before each start of the simulation.

The other function being implemented is `update()` [46]. Firstly, we get a reference to the `StateArray` for the excitatory and inhibitory inputs [47-48] (see section 1.2.2 on page 8).

For clarity, we create a local reference to the state arrays [49-50]. Thus, the state array pointers need only be dereferenced once, which enhances performance.

For ease of use the parameter values can be extracted from parameter objects [52-55]. On line [58-60] we update the internal state, and the output state [64-65]. The calculation of the output state may seem strange, but becomes clearer when taking into account that `StateArray[0]` returns a `valarray`. The operation performed here is referred to as "subset masking" (Josuttis, 1999).

Listing 2.2: Neuron code example

```
 1 #include "neuronIntegrateFire.hpp"
 2 
 3 /* Interface for dynamic loading is built using a macro. */
 4 MAKE_NEURON_DLL_INTERFACE(iqrcommon::ClsNeuronIntegrateFire,
 5                           "Integrate & fire")
 6 
 7 iqrcommon::ClsNeuronIntegrateFire::ClsNeuronIntegrateFire()
 8     : ClsNeuron(),
 9       pVmembrane(0),
10       pActivity(0) {
11 
12     pExcGain = addDoubleParameter("excGain", "Excitatory gain",
13                                   1.0, 0.0, 10.0, 4,
14                                   "Gain of excitatory inputs.\n"
15                                   "The inputs are summed before\n"
16                                   "being multiplied by this gain.",
17                                   "Input");
18 
19     pInhGain = addDoubleParameter("inhGain", "Inhibitory gain",
20                                   1.0, 0.0, 10.0, 4,
21                                   "Gain of inhibitory inputs.\n"
22                                   "The inputs are summed before\n"
23                                   "being multiplied by this gain.",
24                                   "Input");
25 
26     /* Membrane persistence. */
27     pVmPrs = addDoubleParameter("vmPrs", "Membrane persistence",
28                                 0.0, 0.0, 1.0, 4,
29                                 "Proportion of the membrane potential\n"
30                                 "which remains after one time step\n"
31                              "if no input arrives.",
32                                 "Membrane");
33 
34     pProbability = addDoubleParameter("probability", "Probability",
35                                       0.0, 0.0, 1.0, 4,
                                     "Probability of output occuring\n"
                                     "during a single timestep.",
                                     "Membrane");

    /* Add state variables. */
    pVmembrane = addStateVariable("vm", "Membrane potential");
    pActivity  = addOutputState("act", "Activity");
}

void
iqrcommon::ClsNeuronIntegrateFire::update() {
    StateArray &excitation = getExcitatoryInput();
    StateArray &inhibition = getInhibitoryInput();
    StateArray &vm         = pVmembrane->getStateArray();
    StateArray &activity   = pActivity->getStateArray();

    double excGain     = pExcGain->getValue();
    double inhGain     = pInhGain->getValue();
    double vmPrs       = pVmPrs->getValue();
    double probability = pProbability->getValue();

    /* Calculate membrane potential */
    vm[0] *= vmPrs;
    vm[0] += excitation[0] * excGain;
    vm[0] -= inhibition[0] * inhGain;

    activity.fillProbabilityMask(probability);
    /* All neurons at threshold or above produce a spike. */
    activity[0][vm[0] >= 1.0] = 1.0;
    activity[0][vm[0] <  1.0] = 0.0;
}
```

2.2 Synapses

Header

The header file for the synapse shown in listing 2.3 is very similar to the one for the neuron. The major difference lies in the definition of a state variable that will be used for feedback input [20].

Listing 2.3: Synapse header example

```
 1 #ifndef SYNAPSEAPICALSHUNT_HPP
 2 #define SYNAPSEAPICALSHUNT_HPP
 3
 4 #include <Common/Item/synapse.hpp>
 5
 6 namespace iqrcommon {
 7
 8     class ClsSynapseApicalShunt : public ClsSynapse
 9     {
10     public:
11         ClsSynapseApicalShunt();
12
13         void update();
14
15     private:
16         /* Hide copy constructor. */
17         ClsSynapseApicalShunt(const ClsSynapseApicalShunt&);
18
19         /* Feedback input */
20         ClsStateVariable *pApicalShunt;
21
22         /* Pointer to output state. */
23         ClsStateVariable *pPostsynapticPotential;
24     };
25 }
26
27 #endif
```

Source

The source code for the synapse is shown in listing 2.4. The precompile statement [4-5] at the beginning of the file identifies the synapse type. In the constructor [7] we define the output state for the synapse [10]. In deviation to neurons, a definition of a feedback input [14] using `addFeedbackInput(...)` is introduced. The remains of the synapse code are essentially the same as for the neuron explained above.

Listing 2.4: Synapse code example

```
1 #include "synapseApicalShunt.hpp"
2
3 /* Interface for dynamic loading is built using a macro. */
4 MAKE_SYNAPSE_DLL_INTERFACE(iqrcommon::ClsSynapseApicalShunt,
5                            "Apical shunt")
6
7 iqrcommon::ClsSynapseApicalShunt::ClsSynapseApicalShunt()
8     : ClsSynapse() {
9
```

```
    /* Add state variables. */
    pPostsynapticPotential = addOutputState("psp", "Postsynaptic potential");

    /* Add feedback input */
    pApicalShunt = addFeedbackInput("apicalShunt", "Apical dendrite shunt");
}

void
iqrcommon::ClsSynapseApicalShunt::update() {
    StateArray &synIn     = getInputState()->getStateArray();
    StateArray &shunt     = pApicalShunt->getStateArray();
    StateArray &psp       = pPostsynapticPotential->getStateArray();

    psp[0] = synIn[0] * shunt[0];
}
```

2.3 Modules

Header

Listing 2.5 shows the header file for a module. As for the neurons and the synapses, the constructor for the module is only called once during start-up of iqr, or if the module type is changed. The constructor is **not** called before each start of the simulation. During the simulation iqr will call the `update()` function of the module at every simulation cycle.

During the process of starting the simulation, `init()` is called, at the end of the simulation `cleanup()`. Any opening of files and devices should therefore be put in `init()`, and not in the constructor. It is crucial to the working of the module, that `cleanup()` resets the module to a state in which `init()` can be called safely again. `cleanup()` must hence close any files and devices that were opened in `init()`.

Modules can receive information from group output state. This is achieved with a
`StateVariablePtr` as defined on line [21].

Listing 2.5: Module header example

```
 1 #ifndef MODULETEST_HPP
 2 #define MODULETEST_HPP
 3
 4 #include <Common/Item/module.hpp>
 5
 6 namespace iqrcommon {
 7
 8     class ClsModuleTest : public ClsModule
 9     {
10     public:
11         ClsModuleTest();
12
13         void init();
14         void update();
15         void cleanup();
16
17     private:
18         ClsModuleTest(const ClsModuleTest&);
19
20         /* input from group */
21         StateVariablePtr* inputStateVariablePtr;
22
23         /* output to group */
24         ClsStateVariable* outputStateVariable;
25
26         ClsDoubleParameter *pParam;
27     };
28 }
29
30 #endif
```

Source

In the implementation of the module (listing 2.6) we first define the precompile statement [3-4] to identify the module vis-à-vis iqr. As seen previously, a parameter is added [8-13] in the constructor. Using the function `addInputFromGroup(...)`, which returns a pointer to a

`StateVariablePtr`, we define one input from a group, and via `addOutputToGroup(...)` one output to a group.

In the `update()` function starting on line [28], we access the input state array with `getTarget()->getStateArray()` [31], and the output with `getStateArray()` [35].

Listing 2.6: Module code example

```
 1 #include "moduleTest.hpp"
 2 
 3 MAKE_MODULE_DLL_INTERFACE(iqrcommon::ClsModuleTest,
 4                           "test module 1")
 5 
 6 iqrcommon::ClsModuleTest::ClsModuleTest() :
 7     ClsModule() {
 8     pParam = addDoubleParameter("dummy Par0",
 9                        "short description",
10                        0.0, 0.0,
11                        1.0, 3,
12                        "Longer description",
13                        "Params");
14 
15     /* add input from group */
16     inputStateVariablePtr = addInputFromGroup("_nameIFG0", "IFG 0");
17 
18     /* add output to group */
19     outputStateVariable = addOutputToGroup("_nameOTG0", "OTG 0");
20 }
21 
22 
23 void
24 iqrcommon::ClsModuleTest::init(){
25     /* open any devices here */
26 };
27 
28 void
29 iqrcommon::ClsModuleTest::update(){
30     /* input from group */
31     StateArray &clsStateArrayInput =
32         inputStateVariablePtr->getTarget()->getStateArray();
33 
34     /* output to group */
35     StateArray &clsStateArrayOut = outputStateVariable->getStateArray();
36 
37     for(unsigned int ii=0; ii<clsStateArrayOut.getWidth(); ii++){
38         clsStateArrayOut[0][ii] = ii;
39     }
40 };
41 
42 void
43 iqrcommon::ClsModuleTest::cleanup(){
44     /* close any devices here */
45 };
```

2.3.1 Threaded modules

Threaded modules are derived from the `ClsThreadModule` class, as shown in the code fragment in listing 2.7.

Listing 2.7: Threaded module header fragment

```
#include <Common/Item/threadModule.hpp>

namespace iqrcommon {
    class moduleTTest : public ClsThreadModule {
    ...
```

The main difference in comparison with a non-threaded module is the protection of the access to the input and output data structures by means of a mutex as shown in listing 2.8. On line [3] we lock the mutex, then access the data structure, and unlock the mutex, when done [8]

Listing 2.8: Threaded module `update()` function

```
 1 void
 2 iqrcommon::moduleTTest::update(){
 3     qmutexThread->lock();
 4     StateArray &clsStateArrayOut = clsStateVariable0->getStateArray();
 5     for(unsigned int ii=0; ii<clsStateArrayOut.getWidth(); ii++){
 6         clsStateArrayOut[0][ii] = ii;
 7     }
 8     qmutexThread->unlock();
 9
10 };
```

2.3.2 Module errors

To have a standardized way of coping with errors occurring in modules the `ModuleError` class is used. Listing 2.9 shows a possible application of the error class for checking the size of the input and output states.

Listing 2.9: Throwing a `ModuleError` code example

```
void
iqrcommon::ClsModuleTest::init(){
    if(inputStateVariablePtr->getTarget()->getStateArray().getWidth()!=9 ){
        throw ModuleError(string("Module \"") +
                          label() +
                          "\": needs 9 cells for output");
    }

    if(outputStateVariable->getStateArray().getWidth()!=10){
        throw ModuleError(string("Module \"") +
                          label() +
                          "\": needs 10 cells for input");
    }
}
```

Appendix

3.1 iqrcommon::ClsItem Class Reference

Public Member Functions

- ClsBoolParameter ∗ **addBoolParameter** (string _strName, string _strLabel, string _strDescription="", string _strCategory="")
- ClsBoolParameter ∗ **addBoolParameter** (string _strName, string _strLabel, bool _bValue, string _strDescription="", string _strCategory="")
- ClsDoubleParameter ∗ **addDoubleParameter** (string _strName, string _strLabel, string _strDescription="", string _strCategory="")
- ClsDoubleParameter ∗ **addDoubleParameter** (string _strName, string _strLabel, double _dValue, double _dMinimum, double _dMaximum, int _iPrecision, string _strDescription="", string _strCategory="")
- ClsIntParameter ∗ **addIntParameter** (string _strName, string _strLabel, string _strDescription="", string _strCategory="")
- ClsIntParameter ∗ **addIntParameter** (string _strName, string _strLabel, int _iValue, int _iMinimum, int _iMaximum, string _strDescription="", string _strCategory="")
- ClsOptionsParameter ∗ **addOptionsParameter** (string _strName, string _strLabel, string _strDescription="", string _strCategory="")
- ClsOptionsParameter ∗ **addOptionsParameter** (string _strName, string _strLabel, bool _bReadOnly, string _strDescription="", string _strCategory="")
- ClsStringParameter ∗ **addStringParameter** (string _strName, string _strLabel, string _strDescription="", string _strCategory="")
- ClsStringParameter ∗ **addStringParameter** (string _strName, string _strLabel, string _strValue, bool _bEditable, bool _bLong, string _strDescription="", string _strCategory="")
- const ParameterList & **getListParameters** () const
- ClsParameter ∗ **getParameter** (string _strName)
- void **setParameter** (string _strName, string _strValue)
- void **setParameters** (const ParameterList &_lstParameters)

3.1.1 Member Function Documentation

iqrcommon::ClsBoolParameter ∗ iqrcommon::ClsItem::addBoolParameter (string *_strName*, string *_strLabel*, bool *_bValue*, string *_strDescription* = "", string *_strCategory* = "")

Add a new boolean parameter.

This function creates a parameter with a specified value.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_bValue Initial value of the parameter.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsBoolParameter ∗ iqrcommon::ClsItem::addBoolParameter (string *_strName*, string *_strLabel*, string *_strDescription* = "", string *_strCategory* = "")

Add a new boolean parameter.

This function creates a parameter with the default value.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsDoubleParameter ∗ iqrcommon::ClsItem::addDoubleParameter (string *_strName*, string *_strLabel*, double *_dValue*, double *_dMinimum*, double *_dMaximum*, int *_iPrecision*, string *_strDescription* = "", string *_strCategory* = "")

Add a new double parameter.

This function creates a parameter with a specified value and range.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_dValue Initial value of the parameter.

_dMinimum Minimum value of the parameter.

_dMaximum Maximum value of the parameter.

_iPrecision Precision (number of decimal places) of the parameter.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsDoubleParameter ∗ iqrcommon::ClsItem::addDoubleParameter (string *_strName*, string *_strLabel*, string *_strDescription* = "", string *_strCategory* = "")

Add a new double parameter.

This function creates a parameter with the default value and range.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsIntParameter ∗ iqrcommon::ClsItem::addIntParameter (string *_strName*, string *_strLabel*, int *_iValue*, int *_iMinimum*, int *_iMaximum*, string *_strDescription* = "", string *_strCategory* = "")

Add a new int parameter.

This function creates a parameter with a specified value and range.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_iValue Initial value of the parameter.

_iMinimum Minimum value of the parameter.

_iMaximum Maximum value of the parameter.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsIntParameter ∗ iqrcommon::ClsItem::addIntParameter (string *_strName*, string *_strLabel*, string *_strDescription* = "", string *_strCategory* = "")

Add a new int parameter.

This function creates a parameter with the default value and range.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsOptionsParameter ∗ iqrcommon::ClsItem::addOptionsParameter (string *_strName*, string *_strLabel*, bool *_bReadOnly*, string *_strDescription* = "", string *_strCategory* = "")

Add a new options parameter.

This function creates a parameter with an empty read-only options list. The options must be added explicitly using ClsOptionsParameter::addOption.

Returns:

Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_bReadOnly Sets the options list to read-only when true. For non-readonly lists, the corresponding parameter widget will allow the user to add new options to the list.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsOptionsParameter ∗ iqrcommon::ClsItem::addOptionsParameter (string *_strName*, string *_strLabel*, string *_strDescription* = "", string *_strCategory* = "")

Add a new options parameter.

This function creates a parameter with an empty read-only options list. The options must be added explicitly using ClsOptionsParameter::addOption.

Returns:

Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsStringParameter $*$ iqrcommon::ClsItem::addStringParameter (string *_strName*, string *_strLabel*, string *_strValue*, bool *_bEditable*, bool *_bLong*, string *_strDescription* = "", string *_strCategory* = "")

Add a new string parameter.

This function creates a parameter with the specified value and properties.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strValue Initial value of the parameter.

_bEditable Sets whether the string is editable (true) or readonly (false).

_bLong Sets whether the string is multiline (true) or single line (false).

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

iqrcommon::ClsStringParameter $*$ iqrcommon::ClsItem::addStringParameter (string *_strName*, string *_strLabel*, string *_strDescription* = "", string *_strCategory* = "")

Add a new string parameter.

This function creates a short editable empty parameter string.

Returns:
Pointer to the new parameter object.

Parameters:

_strName Name of the new parameter (used in the system file).

_strLabel Label to use for this parameter in a dialog.

_strDescription Description of the parameter, used for creating tooltips/help.

_strCategory Category of the parameter, used to arrange parameter widgets into categories in a dialog.

const iqrcommon::ParameterList & iqrcommon::ClsItem::getListParameters () const

Export the parameter list.

Returns:
Reference to list of parameter objects.

iqrcommon::ClsParameter * iqrcommon::ClsItem::getParameter (string *_strName*)

Find a named parameter.

This function searches the parameter map for the specified name and, if the name is found, returns the corresponding parameter pointer. WARNING: if the specified name is not found, this function returns 0. It is the responsibility of the caller to check the return value before use.

Returns:

Pointer to the requested parameter object, or 0 if the specified name was not found in the parameter map.

Parameters:

_strName Name of the desired parameter.

void iqrcommon::ClsItem::setParameter (string *_strName*, string *_strValue*)

Set the value of a named parameter.

If the parameter cannot be found, this function does nothing.

Parameters:

_strName Name of the parameter to set.

_strValue Value of the parameter as a std::string. The parameter object handles translation of the string into the relevant type.

void iqrcommon::ClsItem::setParameters (const ParameterList & *_lstParameters*)

Set the values of the parameters. If the parameters cannot be found, this function does nothing.

Parameters:

_lstParameters List of parameter objects.

3.2 iqrcommon::ClsNeuron Class Reference

Public Member Functions

- **ClsNeuron** ()
- virtual ∼**ClsNeuron** ()
- void **addExcitatoryInput** (StateArray ∗_pExcInput)
- void **addInhibitoryInput** (StateArray ∗_pInhInput)
- void **addModulatoryInput** (StateArray ∗_pModInput)
- ClsStateVariable ∗ **getOutputState** ()
- virtual void **update** ()=0
- const StateVariableList & **getListStates** () const
- ClsStateVariable ∗ **getState** (string _name)

Protected Member Functions

- ClsStateVariable ∗ **addStateVariable** (string _name, string _label, unsigned int _minLengthHistory=1)
- ClsStateVariable ∗ **addOutputState** (string _name, string _label, unsigned int _minLengthHistory=1)
- virtual StateArray & **getExcitatoryInput** ()
- virtual StateArray & **getInhibitoryInput** ()
- virtual StateArray & **getModulatoryInput** ()

3.2.1 Constructor & Destructor Documentation

iqrcommon::ClsNeuron::ClsNeuron ()

iqrcommon::ClsNeuron::∼ClsNeuron () `[virtual]`

3.2.2 Member Function Documentation

void iqrcommon::ClsNeuron::addExcitatoryInput (StateArray ∗ *_pExcInput*)

void iqrcommon::ClsNeuron::addInhibitoryInput (StateArray ∗ *_pInhInput*)

void iqrcommon::ClsNeuron::addModulatoryInput (StateArray ∗ *_pModInput*)

const iqrcommon::StateVariableList & iqrcommon::ClsNeuron::getListStates () const

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsNeuron::getOutputState ()

Gets the output state, allowing output connections access to the data.

Returns:

> Pointer to the output state variable. WARNING: If addOutputState has not been called, this function returns a NULL pointer.

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsNeuron::getState (string _*name*)

Gets the named state.

Returns:
Pointer to the state variable. WARNING: If the named state variable is not found, this function returns 0.

virtual void iqrcommon::ClsNeuron::update () `[pure virtual]`

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsNeuron::addStateVariable (string _*name*, string _*label*, unsigned int _*minLengthHistory* = 1) `[protected]`

Add a state variable.

This function may be called repeatedly during construction of derived classes to add states with unique names.

Parameters:
_name Name of the state variable.

_label Human-readable label for the state variable.

_iMinLength Minimum length of the state array buffer.

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsNeuron::addOutputState (string _*name*, string _*label*, unsigned int _*minLengthHistory* = 1) `[protected]`

Add the output state, which provides the information transmitted from a group via its output connections.

This function should be called exactly once during the construction of a neuron type. If repeated calls are made, only the last state will be used as the output. WARNING: If this function is not called, the name of the output site remains undefined and a runtime error will occur when getOutputState is called.

Returns:
Pointer to the newly created state variable.

Parameters:
_name Name of the state.

_label Human-readable label of the state.

_minLengthHistory Minimum length of the history.

iqrcommon::StateArray & iqrcommon::ClsNeuron::getExcitatoryInput () `[protected, virtual]`

Calculate the total excitatory input.

This default function sums all excitatory inputs on a neuron-by-neuron basis and returns a reference to the result. The individual inputs are stored in a list of state arrays created by the incoming connections using addExcitatoryInput.

Returns:

Reference to the total excitatory input.

iqrcommon::StateArray & iqrcommon::ClsNeuron::getInhibitoryInput () `[protected, virtual]`

Calculate the total inhibitory input.

This default function sums all inhibitory inputs on a neuron-by-neuron basis and returns a reference to the result. The individual inputs are stored in a list of state arrays created by the incoming connections using addInhibitoryInput.

Returns:

Reference to the total inhibitory input.

iqrcommon::StateArray & iqrcommon::ClsNeuron::getModulatoryInput () `[protected, virtual]`

Calculate the total modulatory input.

This default function sums all modulatory inputs on a neuron-by-neuron basis and returns a reference to the result. The individual inputs are stored in a list of state arrays created by the incoming connections using addModulatoryInput.

Returns:

Reference to the total modulatory input.

3.2.3 Define Documentation

#define MAKE_NEURON_DLL_INTERFACE(NeuronClass, Label)

Value:

```
extern "C" { \
    const char* label() { return Label; } \
    iqrcommon::ClsNeuron* create() { return new NeuronClass; } \
    void destroy(const iqrcommon::ClsNeuron* _pNeuron) { delete _pNeuron; } \
}
```

Macro which builds the shared object interface needed to dynamically load neuron types.

Parameters:

NeuronClass Name of the neuron class definition including the namespace.

Human-readable Name for the neuron class, e.g. for a neuron class ClsNeuronRandomSpike, the name might be "Random spike". Users will use this name to create objects of the neuron type. The name should be supplied inside "".

3.3 iqrcommon::ClsSynapse Class Reference

Public Member Functions

- **ClsSynapse** ()
- virtual ∼**ClsSynapse** ()
- ClsStateVariable ∗ **getInputState** ()
- ClsStateVariable ∗ **getOutputState** ()
- virtual void **update** ()=0
- const StateVariableList & **getListStates** () const
- ClsStateVariable ∗ **getState** (string _name)
- void **setFeedbackInputByName** (string _name, string _input)
- void **setFeedbackInputByLabel** (string _label, string _input)

Protected Member Functions

- ClsStateVariable ∗ **addStateVariable** (string _name, string _label, unsigned int _minLengthHistory=1)
- ClsStateVariable ∗ **addOutputState** (string _name, string _label, unsigned int _minLengthHistory=1)
- ClsStateVariable ∗ **addFeedbackInput** (string _name, string _label, unsigned int _minLengthHistory=1)

3.3.1 Constructor & Destructor Documentation

iqrcommon::ClsSynapse::ClsSynapse ()

iqrcommon::ClsSynapse::∼ClsSynapse () `[virtual]`

3.3.2 Member Function Documentation

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::getInputState ()

const iqrcommon::StateVariableList & iqrcommon::ClsSynapse::getListStates () const

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::getOutputState ()

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::getState (string *_name*)

Gets the named state.

Returns:

Pointer to the state variable. WARNING: If the named state variable is not found, this function returns 0.

virtual void iqrcommon::ClsSynapse::update () `[pure virtual]`

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::addStateVariable (string *_name*, string *_label*, unsigned int *_minLengthHistory* = 1) `[protected]`

Add a state variable.

This function may be called repeatedly during construction of derived classes to add states with unique names.

Parameters:

_name Name of the state variable.

_label Human-readable label for the state variable.

_iMinLength Minimum length of the state array buffer.

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::addOutputState (string *_name*, string *_label*, unsigned int *_minLengthHistory* = 1) `[protected]`

Add the output state.

This function should be called exactly once during the construction. If repeated calls are made, only the last state will be used as the output. WARNING: If this function is not called, the name of the output site remains undefined and a runtime error will occur when getOutputState or getOutput are called.

Returns:

Pointer to the newly created state variable.

Parameters:

_name Name of the state.

_label Human-readable label of the state.

_minLengthHistory Minimum length of the history.

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsSynapse::addFeedbackInput (string *_name*, string *_label*, unsigned int *_minLengthHistory* = 1) `[protected]`

3.3.3 Define Documentation

#define MAKE_SYNAPSE_DLL_INTERFACE(SynapseClass, Label)

Value:

```
extern "C" { \
    const char* label() { return Label; } \
    iqrcommon::ClsSynapse* create() { return new SynapseClass; } \
    void destroy(const iqrcommon::ClsSynapse* _pSynapse) { delete _pSynapse; } \
}
```

Macro which builds the shared object interface needed to dynamically load synapse types.

Parameters:

SynapseClass Name of the synapse class definition including the namespace.

Human-readable Name for the synapse class, e.g. for a synapse class ClsSynapseSimple, the name might be "Simple". Users will use this name to create objects of the synapse type. The name should be supplied inside "".

3.4 iqrcommon::ClsModule Class Reference

Public Member Functions

- **ClsModule** ()
- virtual ∼**ClsModule** ()
- ClsStateVariable ∗ **getState** (string _name)
- **StateVariablePtr** ∗ **addInputFromGroup** (string _name, string _label)
- ClsStateVariable ∗ **addOutputToGroup** (string _name, string _label)
- **StateVariablePtrList** & **getListInputFromGroupPtrs** ()
- virtual void **init** ()
- virtual void **update** ()
- virtual void **cleanup** ()
- virtual moduleIcon **getIcon** ()

3.4.1 Constructor & Destructor Documentation

iqrcommon::ClsModule::ClsModule ()

iqrcommon::ClsModule::∼ClsModule () `[virtual]`

3.4.2 Member Function Documentation

iqrcommon::StateVariablePtr ∗ iqrcommon::ClsModule::addInputFromGroup (string *_name*, string *_label*)

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsModule::addOutputToGroup (string *_name*, string *_label*)

virtual void iqrcommon::ClsModule::cleanup () `[inline, virtual]`

moduleIcon iqrcommon::ClsModule::getIcon () `[virtual]`

StateVariablePtrList& iqrcommon::ClsModule::getListInputFromGroupPtrs () `[inline]`

iqrcommon::ClsStateVariable ∗ iqrcommon::ClsModule::getState (string *_name*)

virtual void iqrcommon::ClsModule::init () `[inline, virtual]`

virtual void iqrcommon::ClsModule::update () `[inline, virtual]`

3.4.3 Define Documentation

#define MAKE_MODULE_DLL_INTERFACE(ModuleClass, Label)

Value:

```
extern "C" { \
    const char* label() { return Label; } \
    iqrcommon::ClsModule* create() { return new ModuleClass; } \
    void destroy(const iqrcommon::ClsModule* _pModule) { delete _pModule; } \
}
```

Macro which builds the shared object interface needed to dynamically load module types.

Parameters:

ModuleClass Name of the module class definition including the namespace.

Human-readable Name for the module class, e.g. for a module class ClsModuleSimple, the name might be "Simple". Users will use this name to create objects of the module type. The name should be supplied inside "".

3.5 iqrcommon::ClsThreadModule Class Reference

Public Member Functions

- ∼**ClsThreadModule** ()

Protected Attributes

- QMutex ∗ **qmutexThread**

3.5.1 Constructor & Destructor Documentation

iqrcommon::ClsThreadModule::∼ClsThreadModule () `[inline]`

3.5.2 Member Data Documentation

QMutex∗ iqrcommon::ClsThreadModule::qmutexThread `[protected]`

3.6 iqrcommon::ModuleError Class Reference

`#include <module.hpp>`

Public Member Functions

- **ModuleError** (const string &_strReason)

3.6.1 Constructor & Destructor Documentation

iqrcommon::ModuleError::ModuleError (const string & *_strReason*) `[inline]`

Parameters:

_strReason Reason for failure.

Index

Bibliography

Josuttis, N. M. (1999). *The C++ standard library: a tutorial and reference.* Addison Wessley.