Appendix II – NSLJ Extensions

This section describes the features of the NSLJ simulation environment that are not present in the standard system. We expect that these extensions will be incorporated into NSL3_0 in the future.

A.II.1 Additional NslModule Types

The NslOutModules and NslInModules are import within the NSLJ system since they allow for the special processing of display data and are a core part of the user interface. It is important to note that NslInModules and NslOutModules are scheduled by the scheduler while NslInFrames and NslOutFrames are not. NslInFrames and NslOutFrames are just one variable within a NslInModule or a NslOutModule. The NslOutModule is used to control the output going to a NslOutFrame. Every NslOutModule has one and only one NslOutFrame. (The frame's title is generated from the NslOutModule's instance name.) Every NslOutModule has one and only one NslOutFrame. (The frame's title is generated from the NslOutModule's.) Each NslOutModule must specify to which protocols it belongs and all NslOutModule are assumed to belong to the "manual" protocol unless specifically removed. If a NslOutModule is enabled by selection of a protocol, it is executed by the scheduler at the frame's specified DisplayDelta times. The same holds true for NslInModules as well: the module has one and only one NslInFrame, each NslInModule must specify to which protocols it belongs, and it executes at the frame's specified DisplayDelta time.

To define a module of type NslOutModule, type the following:

```
nslOutModule Foo (int size) {
   public void initModule() {
    nslAddProtocolRecursiveUp(“Jumping”);
    nslAddAreaCanvas(outpu,-1,1);
    nslAddTemporalCanvas(energy,-1-,10);
   }
}
```

In the code above, we have declared the nslOutModule "Foo" and have subscribed to the protocol "Jumping". In addtion, we have also declared that two plots shall appear on this NslOutModule's NslOutFrame—namely one Area graph and one temporal graph. All standard output plots can be added in this way.

To define a module of type NslInModule, type the following:

```
nslInModule Moo (int size) {
   public void initModule() {
    nslAddProtocolRecursiveUp(“Learn”);
    nslAddPanel(“controlBar”);
    nslAddButton(“clear”,”Clear image”,”controlBar”);
   //stuff
   }
public void clearPushed() {//the name “clear” came from
    nslAddButton
   //stuff
   }

}
```

In the NslInModule we have added a few of the NSLJ widget types. For a complete list of these, please see our website at http://www-hbp.usc.edu/Projects/nsl.htm. Here are the ones we used above.

NslAddPanel is used to add a blank panel to the canvas

```
public void nslAddPanel(charString name){}
```

where name is the variable name of the panel. The size of the panel will grow with each new button added to the panel.

NslAddButton is used to add a button to the panel.

```
public void nslAddButton(charString name, charString label,
charString panel name){}
```

where name is the name of the button. A corresponding method must be written in the user's code with the name "buttonnamePushed" ("buttonname" concatenated with "Pushed".). The label name is the name that will appear on the button, and the panel name is the name of the panel in which to place the button.

nslAddInputImageCanvas looks like that shown in figure 5.18. NslInputImage-Canvas is placed directly on NslInFrames. Clicking the box of the element desired within the grid will cause that box to become shaded and take on the ymax value. All boxes not selected will have the ymin value.

```
public void nslAddInputImageCanvas(NslNumeric variable, int
ymin, int ymax){}
```

where the NslNumeric can have one or two dimensions; ymin is the lower bound on y; and ymax is the upper bound on y.

NslAddNumericEditorCanvas allows the user to see the values of a zero, one, or two-dimensional array displayed in a grid like fashion.

```
public void nslAddNumericEditorCanvas(NslNumeric variable, int
ymin, int ymax){}
```

where the NslNumeric can have zero, one or two dimensions; ymin is the lower bound on y, and ymax is the upper bound on y. For an example of the NumericEditor widget please see figure 5.17. In that example we add three NumericEditor Widgets to one frame. The NumericEditor widget can be used both as an input widget and an output widget. If used as an output widget, the values are updated every Display Delta increment.

A.II.2 NSLM Extensions

Additional System Methods

Two convenient methods in neural simulation are "nslGetValue" and "nslSetValue." These methods can act as **probes** (system.**nslGetValue**, system.**nslSetValue-**(*foo*, "*modelA.m1.w1*") and as **injectors** (system.**nslSetValue**(*"modelA.m1.w1"*, *foo*)). We can use them in a **NslModule** or a **NslClass** without having to put the "system" in front of the method name. Thus we would just type:

```
foo=(NslFloat1)nslGetValue( "modela.m1.w1");
```

The method **nslGetModelRef** is also convenient for manipulating the model instance by returning a reference to a variable of type **NslModel**. The syntax is:

```
var1=system.nslGetModelRef();
```

The **nslGetRefOfModuleOrClass** method is similar to the **getModelRef** method. It returns a reference to a NslHierarchy class object, thus you will need to cast it to the appropriate type:

```
var2=(NslModule)system.nslGetRefOfModuleOrClass
("modelA.m1",'R');
var3=(NslClass)system.nslGetRefOfModuleOrClass
("modelA.c1",'R');
```

Note that "*modelA.m1*" is m1's *long-name* or *real-name*. *long-names* or *real-names* start with the model instance names and each child module or class instance is appended from there.

Differential Approximation

To add a new approximation method we type:

```
system.addApproximationMethod(NslDiff);
```

To set or get the system approximation method we use:

```
system.setApproximationMethod(NslDiff);
NslDiff diffObj;
diffObj=system.getApproximationMethod();
```

Currently only **NslDiffEuler** and **NslDiffRungeKutta2** are available as parameters to the **setApproximationMethod**. The default approximation method is Euler.

To set or get the current approximation delta we type:

```
system.setApproximationDelta(double);
double var=system.getApproximationDelta();
```

The default delta is 0.1.

To set or get the current approximation time constant we type:

```
system.setApproximationTimeConstant (double);
double var=system.getApproximationTimeConstant();
```

The default time constant is 1.0.

DisplayDelta

To set or get *t5* as a double representing the display delta or update time. The current default display delta is set to every cycle.

```
system.setDisplayDelta(t5); );
double var = system.getDisplayDelta();
```

Additional NslBase Methods

The **NslBase** class is the most primitive class in the NSL class hierarchy tree. Every NSL object inherits this class (except for system) and thus can use these methods. Since these classes could be *subclassed*, we prefix their names with "nsl" so that the model builder does not override our methods accidentally.

To set the parent of an object instance we can use:

```
obj.nslSetParent(NslHierarchy parent);
```

To get the parent objects reference we can type:

```
NslHierarchy objParent;
   objParent = objChild.nslGetParent();
```

To get the parent objects reference of type NslModule we can type:

```
NslModule objParent;
   objParent = objChild.nslGetParentModule();
```

To get the parent objects reference of type NslClass we can type:

```
NslClass objParent;
   objParent = objChild.nslGetParentClass();
```

Additional NslData Methods

The **NslData** class provides the backbone for the classes **NslNumeric**, **NslString** and **NslBoolean**. Half of the methods are abstract or virtual meaning that they must be overridden in all of the subclasses. In all of the following examples we assume that *objX* is of some **NslData** type, such as **NslBoolean0**, **NslString0**, **NslDouble0**, **NslDinDouble0**, or **NslDoutDouble0**.

The first method we will discuss is **duplicateData**. This method is abstract/virtual, and it copies or clones the value of *obj1* and places it in the parameter.

```
NslFloat2 obj1(4,4);
NslFloat2 obj2(4,4);
obj1.duplicateData(obj2);
```

The **duplicateThis** method is abstract/virtual, and it returns a copy of itself.

```
obj2=(NslFloat2)obj1.duplicateThis();
```

where *obj2* is of type and will probably need to be cast to the same type that obj1 is.

The next method is also abstract and called **setReference**. This method sets the reference pointer of this object to the data value of the parameter. (It is similar to two pointers pointing to a same object in C/C++.) Whenever the data value of one side is changed, the other side is changed as well. It is used only in NslPorts.

```
obj3.setReference(obj1);
```

The **isDataSet** method is also used within the NSL system. This method checks whether the object value is null or not. (This method is also abstract/virtual.) A **NslData**'s value can be null if the **NslData** object was created without instantiations, and the user was planning to use **nslMemAlloc** to allocate the space for the value.

```
obj4.isDataSet();
```

The next method is a complement to **isDataSet**. It is called **resetData**. It sets the objects value back to null. (This method is also abstract/virtual.)

```
obj5.resetData();
```

The methods which are not abstract/virtual but which can be overridden are:

```
int sizes[4];
sizes=obj8.getSizes();
```

The last statement returns the sizes for all dimensions. If *obj8* is a scalar, then the *sizes* array will contain all zeros. If *obj8* is a vector (dimension1), then *sizes*[0] will contain the length of the vector, etc.

Additional NslNumeric Methods

The numeric methods involve the following classes: **NslNumeric** (only abstract /virtual), **NslNumeric*X*** (mostly abstract/virtual), **NslDouble*X***, **NslFloat*X***, **NslInt*X***, **NslDinDouble*X***, **NslDinFloat*X***, **NslDinInt*X***, **NslDoutDouble*X***, **NslDoutFloat*X***, **NslDoutInt*X*** where *X* represents 0, 1, 2, 3, or 4. The class hierarchy of the NslDinDouble4 class is shown in figure A.1.1.

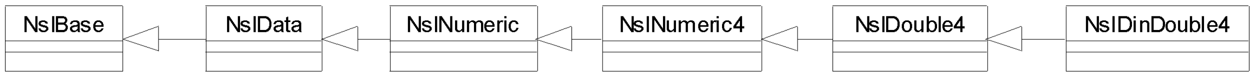


Figure A.II.1
Class Hierarchy of the NslDinDouble4 Class using UML notation.

Since there are so many **NslNumeric** associated classes, we will just mention two of them here: **NslNumeric0**, and **NslDouble2**. We feel that this will give most readers an overview of the methods they are interested in and if the user would like more information, he/she can see our website for the full details on each class.

Additional NslNumeric0 Methods

In this section we will not reiterate the methods that were covered in **NslBase** and **NslData**, we will just assume that the abstract methods mentioned in those classes were implemented correctly in this class or in one of its subclass. In addition to the abstract/virtual type methods, this class also contains pseudo templates for several methods that return a different type based on the type of object the method is associated with. We do this since Java does not let us return different types when a method is declared abstract. In all of the following we assume *obj0* is of some **NslNumeric** type such as **NslDouble0**, **NslDinDouble0**, or **NslDoutDouble0**.

The first of these pseudo abstract methods is **get,** returning a native primitive value or native primitive array reference.

```
var1=obj0.get();
```

The next set of methods are all abstract/virtual, meaning they are all overridden in one of the subclasses NslDouble0, NslFloat0, or NslInt0.

```
doubleVar= obj0.getdouble() ;
floatVar = obj0.getfloat() ;
intVar = obj0.getint() ;
NslDouble0 var = obj0.getNslDouble0() ;
NslFloat0 var = obj0.getNslFloat0() ;
NslInt0 var = obj0.getNslInt0() ;
```

In the next statement value is a either **double**, **float**, **int**, or **NslNumeric0**; notice the set method is *overloaded*

```
obj0.set(value);
```

This last method is not abstract/virtual. It is called **getSize** and is only implemented in **NslNumeric0**, **NslNumeric1**, **NslBoolean0**, and **NslBoolean1**.

```
int someint=obj0.getSize();
```

where *obj0* is either a **NslNumeric0**, **NslNumeric1**, **NslBoolean0**, and **NslBoolean1** type.

Additional NslNumeric2 Methods

Note that in this class we do not declare any abstract/virtual methods since this is a leaf class. However, the methods within the **NslDinDouble2** and **NslDoutDouble2** classes can override these methods. And again we will not repeat the methods covered in **NslBase** or **NslNumeric**. In the following examples, *obj2* is of type **NslDouble2** returning either a reference to the original object's value in the case where no casting is needed, and returning a reference to an object of the appropriate type where casting is indicated.

```
double[][] somedouble2d=obj2.get();
double[] somedouble1d=obj2.get(int);
double somedouble=obj2.get(int,int);
double[][] somedouble2d=obj2.getdouble2();
float[][] somefloat2d=obj2.getfloat2();
int[][] someint2d=obj2.getint2();
double[] somedouble1d=obj2.getdouble1(int);
float[] somefloat1d=obj2.getfloat1(int);
int[] someint1d=obj2.getint1(int);
double somedouble=obj2.getdouble(int,int);
float somefloat=obj2.getfloat(int,int);
int someint=obj2.getint(int,int);
NslDouble2 someNslDouble2(4,4);
NslFloat2 someNslFloat2(4,4);
NslInt2d someNslInt2(4,4);
someNslDouble2=obj2.getNslDouble2();
someNslFloat2=obj2.getNslFloat2();
someNslInt2d=obj2.getNslInt2();
```

Code Segment A.II.5
NslDouble2 Methods Using Get.

Next we have the set methods. All set methods copy the value passed in before assigning to the value of the object. The set methods are overloaded so that they can take a variety of parameters. The first method is:

```
obj2.set(value);
```

where *value* is a native **double**, **float**, **int** array of dimension 2 or **NslNumeric2**; *obj2* is **NslDouble2.** The next method is:

```
obj2.set(int,int,value);
```

where *value* is of type **double**, **float**, **int**, or **NslNumeric0**. This method sets a particular element within the array. The next method is:

```
obj2.set(value);
```

where *value* is of type **double**, **float**, **int** or **NslNumeric0**. This method sets all of the elements of the matrix to the value specified.

Finally, we need to mention the **memAlloc** method. We use this method when we want to dynamically allocate the size of a matrix sometime later on in the simulation. A typical use is to set the dimensions of a variable from a script file or from the NSLS script window. The *Backpropagation* model from chapter 3 set the sizes of some of its NSL objects this way. While a NSL numeric object must be initially specified with an appropriate dimension type, the user may delay specifying the corresponding dimension sizes. For example, a two-dimensional object may have its corresponding sizes specified during object instantiation as follows,

```
NslDouble2 a(size1,size2);
```

or could be specified in two steps using the **memAlloc** function as follows

```
NslDouble2 a();
a.memAlloc(size1,size2);
```

The above memory allocation expression can take place anywhere in the program. Just beware that if the object is used before doing the memory allocation call, errors may result in the program. In addition, NSL port types can only use the **memAlloc** method within the **callFromConstructorBottom** method, or the top of the **makeConn** method. This is due to the fact that **makeConn** wants to make sure the ports are well defined before it connects them to other modules. Remember the dimensions and the sizes of the dimensions on the port types must match to make a connection. If the sizes are not know, then **makeConn** cannot make a connection.

Additional NslBoolean Methods

The **NslBoolean** class inherits from **NslData** and **NslBase**; thus we will not cover the methods from those classes again. However, **NslBoolean** and **NslBoolean*N*** have some methods unique to the **boolean** class. For this example we will look at the **NslBoolean2** class.

In all examples *obj2* is of type **NslBoolean2**. Also the methods that convert from **boolean** to native primitive types, convert true to the value 1 or 1.0, and false to 0 or 0.0.

```
boolean[][] someboolean2d=obj2.get();
boolean[] someboolean1d=obj2.get(int);
boolean someboolean=obj2.get(int,int);
boolean[][] someboolean2d=obj2.getboolean2();
boolean[] someboolean1d=obj2.getboolean1(int);
boolean someboolean=obj2.getboolean(int,int);
NslBoolean2 someNslBoolean2(4,4);
someNslBoolean2=obj2.getNslBoolean2();
```

Next we have the **set** methods. All **set** methods copy the value passed in before assigning to the value of the object. The set methods are overloaded so that they can take a variety of parameters where *value* is a native **double**, **float**, **int** array of dimension 2 or **NslNumeric2**; *obj2* is **NslBoolean2**.

```
obj2.set(value);
```

In the following statement *value* is of type **double**, **float**, **int**, **NslNumeric0** or **NslBoolean0**.

```
obj2.set(int,int,value);
```

In the following statement *value* is of type **double**[], **float**[], **int**[], **NslNumeric1**, or **NslNumeric1**.

```
obj2.set(int, value);
```

Finally we have the **memAlloc** method, and just as in the **NslDouble2** case above, we can dynamically set the sizes of the dimensions of the arrays at run time in any method. However, NSL port types and their dimensions sizes must be defined before the first *nslConnect* statement is made using one of these ports.

```
      NslBoolean2 b();
      b.memAlloc(size1,size2);
```

Additional NslString0 Methods
The **NslString** or the **NslString0** class was covered somewhat in section 6.2. However, we will describe its unique method in more detail here. Again note, that since **NslString** is a subclass of **NslData** and **NslBase** we will not cover those methods here.

In all of the following examples, the *obj0* is of type *NslString0*.

```
   charString somestring=obj0.get();
   charString somestring= obj0.getstring() ;
   NslString0 someNslString0();
   someNslString0 = obj0.getNslString0() ;
```

In the following statement *value* is either a **double**, **float**, **int**, **boolean**, **charString**, **NslNumeric0**, **NslBoolean0**, or **NslString0**; notice the set method is *overloaded*

```
   obj0.set(value);
```

In the following statement *obj0* is of type **NslString0**. **getLength** is only implemented in **NslString0** and it returns the length of the string.

```
   int someint=obj0.getLength();
```

Additional NslHierarchy Methods
The NSLJ class NslHierarchy is the parent class for NslModule and NslClass. Its original name was NslThingsWithChildren but we felt the name was too long. Many of the **NslHierarchy** methods have already been discussed in the **NslSystem** methods earlier in this appendix. We will mention some of them here but will refer you to the **NslSystem** section for a more in depth description of these functions. When the set methods are used in relation to a **NslModule** or **NslClass** object, the setting of a value only change the value of the current module or class and not the entire system. When the get methods are used in relation to a **NslModule** or **NslClass** object, the getting of a value only returns the default for that module or class, and not the system default.

The methods that are also in **NslSystem** are:

```
   value=(cast)mod1.nslGetValue(name);
   mod1.nslSetValue(target,data);
```

where *target* is a **charString** and *data* is of type **NslData**

```
   mod1.nslSetValue(target, num);
```

where *target* is of type **NslData** and *num* is a **charString.**

```
   mod1.nslSetAccessRecursive(char1);
```

where *char1* is either ‘R’, ‘W’, or ‘N’.

The **NslHierarchy** class also contains the following methods (note that all of these methods begin with “nsl” to avoid accidental overrides by subclasses). The following gets the long-name or real-name of the module or class,

```
   somestring=mod1.nslGetRealName();
```

To print the name of the current module/class and the name of its parent module

```
charString=nslGetNameAndParent();
```

To print the name of the current module/class and all ancestors

```
charString=nslGetNameAndParentRecursive();
```

This method gets a reference to the variable with the specified name. This method works as long as the named variable has NSL read access; otherwise it returns null. Note: only NSL types are stored as data variables.

```
var2=nslGetDataVar(name);
```

where name is of type **charString** and *var2* is of type **NslData**. And also

```
var2=nslGetDataVar(name,’R’);
```

where *name* is of type **charString** and *var2* is of type **NslData**. This method gets a reference to the variable with the specified name. This method works as long as the named variable has the specified NSL access; otherwise it returns null. Additional methods, where *name* is **charString** and *status* is boolean are given below:

```
status=nslHasChildClass(name);//true if has instance of
   NslClass
nslPrintChildClasses();// prints all child classes
```

Additional NslModule Methods
All NSL modules inherit from this class. This class contains many methods that we manipulate internally to NSL, and it also contains many classes for the flow of execution, such as the **initSys**, **initModule**, **initRun**, **simRun**, and **endRun** methods that were covered in chapter 6. Since this class is meant to be subclassed we begin all method names with “nsl” and all public attribute variables with underscore. (The exceptions to this rule are the simulation control methods, the setting and getting of delta values, and the getting and setting of the buffering flag.) Also, these methods are typically called from within a NslModule and thus we do not need to put the module instance name in front of the method name. However, if we were to call one of these methods from a different **NslModule** or **NslClass**, then we would need to use the syntax:

```
somemodule.method(param1);.
```

The first set of methods we would like to discuss are the methods that augment the automatic constructor “makeInst”. These methods are meant to be built by the model builder and are described in table A.II.1.

Constructor Methods	Description
callFromConstructorTop	Allows the user to instantiate special objects before the NSL types are instantiated in makeInst. callFromConstructorTop is called immediately upon instantiating a new module; right after any parent attributes are instantiated.
callFromConstructorBottom	Allows the user to instantiate special objects before makeConn called. The callFromConstructorBottom is called immediately after instantiating a new module.
makeInst	makeInst is not overridable and is not callable from the user's code. We use makeInst to instantiate all NSL type parameters and native arrays that were declared in the attribute section of the code. In object-oriented programming terms, makeInst is the heart of the constructor for the module. We could have called it callFromConstrutorMiddle but did not.

Table A.II.1
Module constructor methods.

There are no arguments to **callFromConstructorTop** or **callFromConstructor-Bottom** method. Also, NSL type variables defined in the attribute section of the NslModule, are instantiated after **callFromConstructorTop** and before **callFrom-ConstructorBottom**. Thus, if you need to manipulate one of these attributes, it is best to put the code in **callFromConstructorBottom**. For example, in code segment A.II.1 the **callFromConstructorBottom** method will print the *name* stored for the object as well as the *size* parameter passed to the class during instantiation. This will be done for every new object created of type **MemoryCalc**.

```
public void callFromConstructorBottom()
{
   nslPrint(“MemoryCalc instance name: “, nslName);
   nslPrint(“MemoryCalc size: “, size);
}
```

Code Segment A.II.1
callFromConstructorBottom for NslClasses.

The next method adds a *child* NslModule to the list:

```
nslAddToModuleChildren(child1);
```

The next method gets a reference to the named child module where *name* is of type **charString**:

```
NslModule foo;
foo=nslGetModuleRef(name);
```

To set the access for module and all below it where *char1* is ‘R,’ ‘W,’ or ‘N’ we use:

```
nslSetAccessRecursive(char1);
```

nslHasChildModule will tell you if a module has submodules. Note *status* is of type **boolean** and *name* is of type **charString**.

```
status=nslHasChildModule(name);
```

nslPrintChildModules prints all of the submodules.

```
nslPrintChildModules();
```

nslGetPort will retrieve the reference to the named port.

```
NslDinFloat2 port1(5,5);
port1=nslGetPort(name);
```

The **getDelta** method returns the current simulation delta either Train or Run for this module.

```
double d1=getDelta();
double d1=getTrainDelta();
double d1=getRunDelta();
```

The setTrainDelta method sets the current simulation train delta for this module where *d1* is of type **double.** And the getRunDelta method gets the runDelta value.

```
setTrainDelta(d1);
double var1=getRunDelta(d1);
```

The next methods reset the train delta to the system train delta or the system run delta for all modules :

```
nslResetTrainDelta();
nslResetRunDelta();
```

In the next method, *flag* indicates whether the current module is in the schedule for the training or running phase. We provide this method since sometimes protocols leave out certain modules.

```
boolean status=nslGetTrainEnableFlag();
boolean status=nslGetRunEnableFlag();
```

The next methods sets or gets the currently set approximation delta or methods used in the **nslDiff** methods for this module.

```
double d2=getApproximationDelta();
setApproximationDelta(d2);
NslDiff m2=nslGetApproximationMethod();
nslSetApproximationMethod(m2);
```

Buffering was discussed in chapter 6. However, there are some additional methods. The next method resets the buffering to the system buffering default for all modules below this one.

```
nslResetBuffering ();
```

To add the following protocol name to the system list of protocols and add this name to the module's list of protocols, and add this *name* to all of the protocol lists within the child.

```
nslAddProtocolRecursiveDown(name);
```

To add the following protocol name to the system list of protocols and add this name to the module's list of protocols, and add this name to all of the protocol lists of the ancestors of this module. This is the method typically used by users.

```
nslAddProtocolRecursiveUp(name);
```

To remove the following name from the modules protocol list.

```
nslRemoveFromLocalProtocols(name);
```

To add the name to the system list of protocols and add it also to the NSL Executive list of protocol names.

```
nslDeclareProtocol(name,label);
```

The following methods return the value of the named variable within the parent module. This method is not encouraged since the variable should have been passed to the child.

```
NslData var1;
var1=nslValParent(name);
```

Additional NslClass Methods

NslClass exists because **NslClasses** cannot contain **NslModules**. It inherits from **NslHierarchy** and **NslBase**. Thus, all of the methods available from **NslClass** have already been discussed.

The following method is generated by the preparser and initializes the invisible temporary variables the NSL system uses to in mathematics expressions. It initializes the variables in the specified methods so that it does not have to reinitialized them every cycle.

```
initTempClass();
```

Logical Methods

The following logical methods can be applied pointwise to the variables of either **NslNumeric**, **NslBoolean** or native primitive variables and arrays/matrices.

If *var1* and *var2* are of equal value, the method returns true; else false.

```
nslEqu(var1,var2);
```

If *var1* is greater than or equal to *var2*, the method returns true; else false.

```
nslGeq(var1,var2);
```

If *var1* is greater than to *var2*, the method returns true; else false.

```
nslGtr(var1,var2);
```

If *var1* is less than or equal to *var2*, the method returns true; else false.

```
nslLeq(var1,var2);
```

If *var1* is less than *var2*, the method returns true; else false.

```
nslLes(var1,var2);
```

If *var1* is not equal to *var2*, the method returns true; else false.

```
nslNeq(var1,var2);
```

The following logical methods can be applied to the variables of type **NslBoolean** or native primitive variables and arrays/matrices of type **boolean**. All logical methods are applied pointwise except for **nslAll**, **nslNone** and **nslSome**.

If *var1* and *var2* are both true, the method returns true, else false.

```
nslAnd(var1,var2);
```

If none of the values in *var1* are true, then returns true, else false.

```
nslNone(var1);
```

The following function returns the opposite **boolean** value as that stored in *var1*.

```
nslNot(var1);
```

If *var1* or *var2* is true, the method returns true, else false.

```
nslOr(var1,var2);
```

If all of the values in *var1* are true, then return true, else false.

```
nslAll(var1,);
```

If some of the values in *var1* are true, then return true, else false.

```
nslSome(var1);
```

A.II.3 Displays and Protocols

NSL Protocols

As mentioned in chapter 5 protocols provide an easy way for the model builder to set up predetermined parameters and windows for a particular protocol. We make the distinction between experiment and protocol in that many experiments can be executed for one protocol. For instance if the model builder has a random number generator in the model, then the results of the "run" will be different each time the protocol is executed. The **default protocol** is "**manual**" which means that the model does not have any particular protocols. All modules and the script window subscribe to the manual protocol initially.

Adding Protocols

The user is free to add new protocols via on of the following statements:

```
system.addProtocolToAll("protocolName")
nslAddProtocolRecursiveUp("protocolName")
```

The first statement will subscribe all known modules to the specified protocol; the second will only subscribe the current module and all its ancestors to the protocol. Both statements will add the protocol name to the Executive's menu list of protocols as well as the systems internal list of protocols. The addition of protocols should occur as early as possible in the model creation process; thus we recommend that they be placed in the top module's **initModule** method although they can be placed in any of the initialization methods other than initSys. Also to change the name of protocol in the Executive windows menu, we can use:

```
nslDeclareProtocol("protocolName", "protocolLabel")
```

where only the protocol label will appear in the menu.

Removing Protocols

It is also important to note that a module can remove itself from a particular protocol within an any of the initialization methods (other than initSys) via one of the statements

```
nslRemoveFromLocalProtocols(“protocolName”)
nslRemoveProtocolRecursiveUp(“protocolName”)
```

The difference between the two statements is that the first will unsubscribe only that module from the protocol; the second will unsubscribe the current module and all of it ancestors (even the model module) from the protocol. For instance, in code segments 5.1 and 5.2 we see that these display windows or frames should not appear if the protocol is the default “manual”. If we do not want a particular NslInFrame or NslOutFrame to appear when the NSL system is first started, then we should add a **nslRemoveFrom-LocalProtocols(“manual”)** statement or a **nslRemoveProtocolRecur-siveUp(“name”)** statement to the NslOutModule’s or NslInModule’s initModule method.

Setting the Default Protocol
To set the model up with a particular protocol on start up we can add the statement

```
system.setProtocol(“protocolName”)
```

to any of the initialization methods (other than the initSys) but should be added after all the other protocol statements (if any) have been issued. The statement **system.set-Protocol(**“**protocolName**”**)** first disables any module not subscribed to the protocol, and then enables any module that is subscribed to the protocol. Next it reconnects all of the subscribed modules. Since this is a very expensive operation, we recommend that it be use sparingly and that it only be called from initModule. Also we should note that the setting of the protocol name would only happen after the completion of the initialization cycle or epoch that the statement appears in.

Menu Selection of a Protocol
From the Executive menu we can select a particular protocol that the model builder has provided for us. The new protocol may or may not bring up a new NslInFrame or NslOutFrame (to be discussed below); however, it will almost certainly set different parameter inputs to the model. This is demonstrated in Dominey’s model in chapter 14 and by Jacob Spoelstra’s model in chapter 16.

Getting the Schedule Associated with a Protocol
If curious, the modeler can also query the NSL system to retrieve the schedule of NSL modules that will run under the selected protocol. The call to do this is:

```
nslShowSchedule(“protocolName”);
```

This statement should only be executed only after a protocol has been selected either via the menu system, or via the “system.setProtocol” statement.

Protocol Associated Methods
We can also declare methods associated with the protocol in the same module file that the protocol was declared in. If a protocol is selected, then its associated protocol method will also be called. These methods are not necessary but are a convenient for printing status messages or setting certain variables. Associated protocol methods should be declared in the following way:

```
public void procolnameProtocol() {
      //code
}
```

As can be seen above, the associated protocol method should have a name such as "protocolnameProtocol" where protocol name matches one of the protocols declared concatenated with the word "Prococol".

The following protocol methods are provided for manipulation of the displays and creation and selection of the protocols. We note here that all of the following commands usually are placed in the **initModule** method of the model.

To check if the following protocol exists in the any module use:

```
system.protocolExist(charString);
```

To set or get the current protocol specified by the string name use:

```
system.setProtocol(name);
charString var = system.getProtocol();
```

Another useful method for adding the protocol name to the Executive's menu is:

```
NslDeclareProtocol("name");
```

defined within the NslModule class.

A.II.4 Command Line Parameters

The following methods where designed to get some of the values of the parameters that can be passed into the main model at execution time from the shell window.

Set or get the flag stating whether debug is set or not; default=0. This method can take any integer value and it is up to the user as to its interpretation; debug=0 means no debug.

```
Command line: nslj ModelA –debug int
system.setDebug(int);
int var=system.getDebug();
```

Set or get the flag indicating whether the any graphics should be displayed. If no graphics are to be displayed, the operating system shell window is used as the script window. The default is false. The *noDisplay* option is nice when your are running from a remote machine.

```
Command line: nslj ModelA -noDisplay
system.setNoDisplay(true);
boolean var=system.getNoDisplay();
```

Redirect standard input and output to the console or script window and retrieve whether the *stdio* is to going either the console or script window.

```
Command line: nslj ModelA –stdio script
system.setStdio(charString);
charString var = system.getStdio();
```

Redirect standard error to the console or script window and retrieve whether the *stderr* is going to either the console or script window.

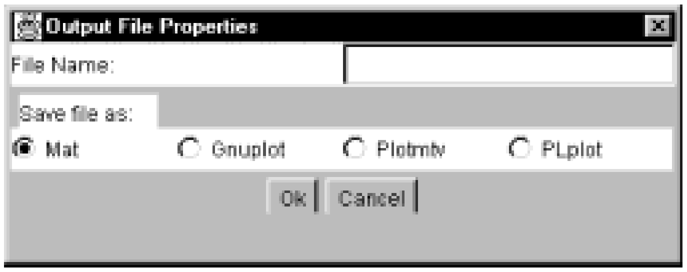
```
Command line: nslj ModelA –stderr console
system.setStderr(charString);
charString var = system.getStderr();
```

Set or get the flag indicating this is a batch job—meaning no graphics and a default script should be provided. The default is false. Batch jobs are convenient for timing a simulation.

```
Command line: nslj ModelA –batch fileName
system.setBatch(boolean);
boolean var=system.getBatch() ;
```

A.II.5 The Interactive System

One of the features left out of chapter 5 was NSLJ's ability to save temporal plot data in the Mathworks Matlab format. To export the data from a Canvas Window first select the canvas and then select "Canvas→Export Data". A pop-up window will appear that looks like that in figure A.II.3. The only plot output currently supported is Matlab from Math-Works. The file specified by the user should end with the ".m" extension. The other files needed to view the NSL data in Matlab are available in NSLJ's "copyme/matlab" directory.



Output File Properties

File Name:

Save file as:

[✓ Mat] Gnuplot Plotmtv PLplot

A.II.2 Figure
The Export Data Popup Window