Neuron Data Elements Environment Intelligent Rules Element

Version 4.1

C++ Programmer's Guide

Contents

Preface

1. C++ API Overview

2. C++ Primer

Preface

Purpose of this Manual

When designing a knowledge-based application that uses the Intelligent Rules Element, you may need some features that a rule- and object-based tool such as the Rules Element may not provide. For example, you may want to use a math library for numerically-intensive computations. Or you may want to write an interface for your knowledge-based application. With this manual, you can write C++ language routines to provide features that the Rules Element doesn't provide, and you can write programs such as an interface to your application. You can write routines and programs that do these tasks by using the routines described in this manual.

Description

Using the Rules Element C++ Application Programming Interface (API), you can write programs that call the Rules Element or routines for the Rules Element to call. The application programming interface is the application programmable interface of the Rules Element. It consists of a set of routines inside the Rules Element that you can call from a program or a routine. Using the routines, you can do tasks such as start the Rules Element's inference engine, find the value of a property slot, and suggest hypotheses. Anything you can do with the graphical user interface of the Rules Element, you can do with the application programming interface. Here are some examples of tasks you can accomplish using this manual and the application programming interface:

- Extend the processing capabilities of the Rules Element. For example, you can embed calls to a math library or to external routines within the Rules Element.
- Write an interface for an application that uses the Rules Element. For example, if you develop an application for the Macintosh and use the Standalone Runtime version of the Rules Element, you can write an interface on top of the Rules Element with the application programming interface.
- Link the Rules Element to databases not supported by the Rules Element. For example, you may have written your own database management system that you want to link to the Rules Element.
- Communicate with and control other processes using the Rules

Element. For example, you can tell the Rules Element to trigger the fire alarm whenever it concludes that there is a fire.

- Monitor real-time processes. For example, you can use the Rules Element in a data-acquisition system.
- Embed the Rules Element's reasoning capabilities in other applications. For example, your CAD / CAM application can call the Rules Element as a subroutine to solve a problem.

Audience

This manual is designed for people who understand programming concepts, the C++ language, and the Rules Element. If you don't understand programming concepts, you may need to review an introductory programming book before you use the API because the interface is used by writing programs and routines. In this manual, the examples are written in C++, so it helps if you are familiar with the C++ language to read through the examples.

Organization

The first time you use this manual, you will probably just read Chapter One, "Overview". The rest of the chapters supply reference information about the individual class methods.

Chapter One, "C++ API Overview" gives you an overview of the application programming interface without going into the details of how to use the Rules Element C++ API.

The API reference chapters describe the following classes:

NxAtom	Provides generic methods that can be applied to all atoms.
NxClass	Provides access to knowledge base classes.
NxCtx	Provides for user-controlled contexts in the Rules Element.
NxEdit	Provides the main knowledge base edit functionality, with fully public members and data.
NxEngine	Provides access to the rule engine.
NxKB	Provides access to knowledge base files.
NxObject	Provides access to knowledge base objects.
NxProp	Provides access to knowledge base properties.
NxRule	Provides access to knowledge base rules.
NxSlot	Provides access to knowledge base slots.

Chapter

1 C++ API Overview

Introduction

This document assumes familiarity with the Intelligent Rules Element (IRE) and C++ programming. It will cover the IRE classes exposed through C++, explaining the hierarchy, the API, techniques used to access the fields and method, and areas to be careful about when using the API.

There are two basic categories of C++ classes for IRE: engine / atom based, and additional functionality based. The first category includes NDNxEngine, NDNxAtom, NDNxRule, etc. The second includes context and edit capabilities, library initialization, etc.

This document briefly describes accessing functionality via class (static) and instance (non-static) methods. The user is encouraged to browse the appropriate header files (noted in the text) for additional information on specific usage. (Note that a few "overhead" methods have been eliminated in order to focus on the specific API of the class.)

Engine / Atom classes

General

The engine and atom classes basically expose what was previously the function oriented C API in a much more object-oriented fashion. The engine class provides basic control over the rules engine (state information, start, etc). The atom classes provide access to methods and data specific to the dynamic atoms of IRE. A generic "Find" method will usually have to be invoked to dynamically lookup one of these atoms, after which instance methods can be invoked. Because of the overlap in atom namespaces, finding the atom will normally require the user to specify the appropriate atom class. For example: "Foo" could be a Rule, Method, Class, Object, Property, Slot (assuming ".Value"), or Knowledge Base (KB).

Note that there is significant overlap between C++ and IRE in the use of object-oriented teminology like object, class, property, method, etc. It will usually be clear from the context whether the keyword in question applies to C++ (the 3GL programmer level) or IRE (the knowledge engineer level). You must not confuse these. In general, a lower case name will apply to C++ (class, object, method), while a capitalized name will apply to IRE (Class, Object, Method), and a name starting with NDNx will be the C++ class of the corresponding IRE entity (NDNxClass, NDNxObject, NDNxMethod).

Also note that in most cases, the class constructor / destructor is "private". This is because these atoms are created and destroyed internally, as needed, by the rule engine. As an end-user, in this release, you may embed an atom reference in any of your structures, but you cannot create or subclass one, because the rule engine would never use it, and it could lead to instability if you passed one to a method. Even when embedding an Atom, you should be aware that the rule engine controls them. It is up to you to be sure that the Atom is still valid at the time you use it (ie: the engine may have deleted it).

An "Indexed" method will return one of N items (when you use the "Count" API to get the number that exist. An index of an Int32 will generally be used as "the index", and will range from 0 ... ("Count" - 1). Parameters and return values, when numeric, are usually set to be "Int32". This is a departure from the previous "NEXPERT" API which used generic Longs and Ints. This is to provide for interoperability across platforms (e.g.: 32 vs 64 bit longs; 16 vs 32 bit ints; etc.).

This API is exception based for error reporting. In general, if you pass an argument that is invalid and causes an error, an exception will be thrown. If you wish to recover from this, you will have to catch it. One of the few cases where an error is not thrown is in looking for an atom by name with a Find method. In this case, a NULL atom will be returned. In the case of an exception being thrown, you will still be able (in most cases) to use the "GetError" method to retrieve additional information about the error, although the text message associated with the exception should be sufficient.

There are still routines that are documented to return status. If a status is returned, it will usually be 1/TRUE. Most returns that would have been 0/FALSE will now trigger exceptions. (In the future, it is likely that these will become "void" functions.)

A VERY IMPORTANT fact to keep in mind is that all "complex primitives" (eg: Str and Variant) returned by this API MUST be disposed by the user when done. Failure to do so could result in significant memory leaks. When an API returns a datatype of "Str" or "NDVarPtr", it represents something that the user is free to modify (having been cloned or constructed from an underlying repsentation). Anything returned of the form "CStr" or "NDVarCPtr" (C for "const") must not be modified or destroyed. If you wish to modify such a return, it is your responsibility to clone it first. Most compilers would not allow you to modify a "const" quantity, anyway. NONE of the IRE Atom- based entites must be destroyed (eg: NDNxAtomPtr, NDNxRulePtr, etc.). Example code will be shown later.

The C++ classes in this section include:

C++ Class	Parent Class	Accesses/Controls	Header File
`NDNxEngine`	`(none)`	`rule engine`	`nxengpub.h`
`NDNxAtom`	`(none)`	`generic IRE Atoms`	`nxatmpub.h`
`NDNxClass`	`(NDNxAtom)`	`IRE Classes`	`nxclspub.h`
`NDNxKB`	`(NDNxAtom)`	`IRE Knowledge bases`	`nxkbpub.h`
`NDNxMethod`	`(NDNxAtom)`	`IRE Methods`	`nxmthpub.h`
`NDNxObject`	`(NDNxAtom)`	`IRE Objects`	`nxobjpub.h`
`NDNxProp`	`(NDNxAtom)`	`IRE Properties`	`nxprppub.h`
`NDNxRule`	`(NDNxAtom)`	`IRE Rules`	`nxrulpub.h`
`NDNxSlot`	`(NDNxAtom)`	`IRE Slots`	`nxsltpub.n`

The Atom class will be used only rarely, for generic access (described later).

Detailed / working examples of this API are found in the Rules C++ examples directory, specifically the "hello" and "nxplinex" examples.

NDNxEngine

There is no NDNxEngine instance / object, as such. All methods relating to control of the inference engine / session are done via static methods. The list of methods and properties exposed are as follows:

```
class NDNxEngine  {

public:
    static NxEngineErrEnumGetError(void);
    static NxEngineCtrlRetEnum  Start(void);
    static NxEngineCtrlRetEnum  Restart(void);
    static NxEngineCtrlRetEnum  Continue(void);
    static NxEngineCtrlRetEnum  Stop(void);
    static NxEngineCtrlRetEnum  Init(void);
    static NxEngineCtrlRetEnum  Exit(void);
    static NxEngineStateEnumGetState(void);
    static Int32 Journal(NxEngineJrnlEnum mode, CStr filename);
    static Int32 GetCurrentStrategy(NxEngineStrategyEnum code);
    static Int32 GetDefaultStrategy(NxEngineStrategyEnum code);
    static void SetCurrentStrategy(NxEngineStrategyEnum code,
                                  Int32 value);
    static void SetDefaultStrategy(NxEngineStrategyEnum code,
                                  Int32 value);
    static NxEngineVolStratEnum GetDefaultResetStrategy(void);
    static void SetDefaultResetStrategy
                                 (NxEngineVolStratEnum strat);
    static NxEngineVolStratEnum GetDefaultVolunteerStrategy(void);
    static void SetDefaultVolunteerStrategy
                                 (NxEngineVolStratEnum strat);
    static NxEngineSugPrioEnum  GetDefaultSuggestStrategy(void);
    static void SetDefaultSuggestStrategy
                                 (NxEngineSugPrioEnum strat);
    static NxpIProc GetHandler(NxEngineProcEnum code);
    static Int32 SetHandler(NxEngineProcEnum code,
                                 NxpIProc proc);
    static NxpIProcGetHandler2(NxEngineProcEnum code,
                                 LongPtr arg);
    static Int32SetHandler2(NxEngineProcEnum code,
                                 NxpIProc proc, Long arg);
    static NxpIProc GetExecuteHandler(CStr name);
    static Int32 SetExecuteHandler(CStr name, NxpIProc proc);
    static NxpIProcGetExecuteHandler2(CStr name, LongPtr arg);
    static Int32SetExecuteHandler2(CStr name, NxpIProc proc,
                                   Long arg);

// use is discouraged for the following methods ...
    static Int32 Compile(CStr str);
    static NxEngineCtrlRetEnum  Control(NxEngineCtrlEnum code);

private:
    NDNxEngine(void) {}
    ~NDNxEngine(void) {}

};
```

The enumerated constants are documented in the header file, and, as enums, help the user select the proper choice of (symbolic) integer values.

The API has been broken down to simplify and clarify its use, and reduce the likelihood of errors. What you will specify in a number of cases will be high level concepts like:

```
NDNxEngine::Start();
NDNxEngine::Restart();
```

rather than low level constructs:

```
NDNxEngine::Control(NXENGINE_CTRL_KNOWCESS);
NDNxEngine::Control(NXENGINE_CTRL_RESTART);
```

These are functionally the same, but the first method is easier to read and understand, and is the mechanism you are encouraged to employ. Note that NXENGINE_CTRL_xxx constants were formerly in the C API as NXP_CTRL_xxx constants. As further encouragement, the low level approach will be phased out over time. Compile also falls into this category (see the NDNxEdit class much later).

Callbacks are performed when the engine needs to resolve issues or notify the user about something. A "SetHandler" mechanism is provided that allows you to register functions in your own program to be called under these circumstances. Be sure to differentiate between default / system handlers and Execute handlers (see the API manuals for details). Execute handlers must be set with the SetExecuteHandler call. There is a variation of both (ending with a "2") that allows additional user-provided information to be passed to your callback. The signature of the function that will be called varies depending on whether you are using the "2" version to register your handler with or not. These signatures are documented in nxengpub.h, as well as in the afore-mentioned API manual. A symmetric "Get" API exists to check for registered callbacks (should you wish to chain them) and / or retrieve additional information.

Some care must be taken registering a callback function under C++. The callback function to be registered (passed as the function pointer) must either not be a member of any class, or, if it is a member of a class, must be a static member. Use of an instance method will cause a compilation error.

NDNxAtom

This class provides generic methods that can be applied to all atoms. The list of methods and properties exposed are as follows:

```
class NDNxAtom  {

protected:
       NXATOM_SUBREC

public:
    static NxAtomPtr  Find(CStr name, NxAtomTypeEnum code);
          NxAtomTypeEnum GetType(void);
          Str          GetName(void);
          Long   GetClientData(void);
          void   SetClientData(Long data);

// use is discouraged for the following methods ...
    static Int32   SetInfo(NxAtomPtr atom1, NxAtomSAInfoEnum code,
                                 NxAtomPtr atom2, Int32 optInt);
    static Int32   GetIntInfo(NxAtomPtr atom1, NxAtomGAInfoEnum code,
                                 NxAtomPtr atom2, Int32 optInt);
    static Long   GetLongInfo(NxAtomPtr atom1, NxAtomGAInfoEnum code,
                                 NxAtomPtr atom2, Int32 optInt);
```

```
    static Double  GetDoubleInfo(NxAtomPtr atom1, NxAtomGAInfoEnum code,
                                  NxAtomPtr atom2, Int32 optInt);
    static Str   GetStrInfo(NxAtomPtr atom1, NxAtomGAInfoEnum code,
                                 NxAtomPtr atom2, Int32 optInt, Int32 len);
    static NxAtomPtr   GetAtomInfo(NxAtomPtr atom1, NxAtomGAInfoEnum code,
                                   NxAtomPtr atom2, Int32 optInt);

};
```

Typical use of this class is when you are passed an atom whose type you wish to determine (eg: in order to invoke additional methods). You could then take whatever actions are appropriate based on:

```
switch (myAtom->GetType()) {
case NXATOM_ATYPE_RULE:
...
}
```

Note that the GetXXXInfo and SetInfo methods are discouraged, for reasons previously discussed in the NDNxEngine section. The functionality previously provided by these calls is now more distributed among the various classes, and you should try to find the functionality you need there first. The "compatibility" functions provided here will be discontinued in future releases.

NDNxClass

The list of methods and properties exposed are as follows:

```
class NDNxClass : public NDNxAtom  {

protected:
    NXCLASS_SUBREC

public:
    static NxClassPtr    Find(CStr name);
    static Int32    GetCount(void);
    static NxClassPtr    GetFirst(void);
    static NxClassPtr    GetLast(void);
          NxClassPtr    GetNext(void);
          NxClassPtr    GetPrevious(void);
          Str           GetName(void);
          Long    GetClientData(void);
          void    SetClientData(Long data);
          NxKBPtr    GetKB(void);
          void    SetKB(NxKBCPtr kb);
              NxObjectPtr   CreateObject(CStr name);
          Int32    DeleteObject(NxObjectPtr object);
          Int32    GetMethodCount(void);
          NxMethodPtr    GetIndexedMethod(Int32 index);
          Int32    GetSlotCount(void);
          NxSlotPtr    GetIndexedSlot(Int32 index);
          Int32    GetParentClassCount(void);
          NxClassPtr    GetIndexedParentClass(Int32 index);
          Int32    GetChildClassCount(void);
          NxClassPtr    GetIndexedChildClass(Int32 index);
          Int32   GetChildObjectCount(void);
          NxObjectPtr   GetIndexedChildObject(Int32 index);
          NxMethodPtr    GetPublicMethod(CStr name);
          NxMethodPtr    GetPrivateMethod(CStr name);
          NxSlotPtr   FindSlot(CStr name);
          NxSlotPtr   FindSlotByProp(NxPropPtr prop);
          NxClassLinkEnum GetLinkType(NxAtomPtr child);
          Int32    MakeLinkPermanent(NxObjectPtr object);
```

```
private:
         NDNxClass(void) {}
         ~NDNxClass(void) {}

};
```

Note that IRE objects have no notion of constructors / destructors (hence the methods CreateObject(...), DeleteObject(...)).

This class allows you to access any information of an IRE Class (name, client data, parents, ...). It allows you to dynamically create and delete Objects associated with this IRE Class. Note: remember that an IRE Class keeps track of its instances (Objects). This is useful, in particular, in pattern matching (please refer to the IRE Reference Manual for more details). An example of this API is:

```
NxClassPtr myClass;
Int32        nChildren;
Str          name;

// find specified Class, if it exists
myClass = NDNxClass::Find("myClass");
if (myClass == NULL) return;

// iterate through child Objects, printing their name
nChildren = myClass->GetChildClassCount();
for (Int32 i=0; i<nChildren, i++) {
      name = myClass->GetIndexedChildClass(i)->GetName();
      printf("%s\n", name);
      NDStr::Dispose(name);
}
```

As mentioned earlier, note that it is important that the returned string be disposed, or memory leaks will occur. Returned Str's must be explicitly destroyed, having been specifically allocated before returning them to the user.

NDNxKB

The list of methods and properties exposed are as follows:

```
class NDNxKB : public NDNxAtom  {

protected:
    NXKB_SUBREC

public:
    static Int32   GetCount(void);
    static NxKBPtr GetFirst(void);
    static NxKBPtr GetLast(void);
    static NxKBPtr Load(CStr name);
    static Int32   ClearAll(void);
    static NxKBPtr GetCurrent(void);
    static void    SetCurrent(NxKBCPtr kb);
    static NxKBPtr Create(CStr name);
    static NxKBPtr Find(CStr name);
    static Int32   Merge(NxKBPtr kb1, NxKBPtr kb2);
          NxKBPtr GetNext(void);
          NxKBPtr GetPrevious(void);
          Str     GetName(void);
          CStr    GetComments(void);
          Int32   MakeLinksPermanent(void);
          Int32   Unload(NxKBUnloadEnum flags);
          Int32   Save(CStr filename, NxKBSaveSet flags);
```

```
private:
    NDNxKB(void) {}
    ~NDNxKB(void) {}

};
```

This class allows you to access or act on Knowledge Base objects as you would in the development environment. Be sure to distinguish between the "Load" method, which will load an existing KB, versus "Create", which creates an empty KB, not associated with any file. An example of this API is:

```
NxKBPtr myKb;

// load an existing KB
myKb = NDNxKB::Load("foo.tkb");
if (myKb == NULL) return;

// if successful, save under a new name with specified options
myKb->Save("bar.tkb", (NXKB_SAVE_TEXT | NXKB_SAVE_COMMENTS));
```

NDNxMethod

The list of methods and properties exposed are as follows:

```
class NDNxMethod : public NDNxAtom  {

protected:
    NXMETHOD_SUBREC

public:
    static Int32 GetCount(void);
    static NxMethodPtr  GetFirst(void);
    static NxMethodPtr  GetLast(void);
    static NxMethodPtr  GetCurrent(void);
          NxMethodPtr  GetNext(void);
          NxMethodPtr  GetPrevious(void);
          Str       GetName(void);
          Str       GetComments(void);
          NxKBPtrGetKB(void);
          void SetKB(NxKBCPtr kb);
          Int32 GetIfConditionCount(void);
          Int32 GetThenActionCount(void);
          Int32 GetElseActionCount(void);
          Str       GetIndexedIfCondition(Int32 index);
          Str       GetIndexedThenAction(Int32 index);
          Str       GetIndexedElseAction(Int32 index);
          Int32 IsPrivate(void);

private:
          NDNxMethod(void) {}
          ~NDNxMethod(void) {}

};
```

This class allows you to access IRE Methods (contents, public or private, etc). See the NDNxProp class example later on for an example of this API.

One thing to note, however, is that it is not quite so straightforward to get an NDNxMethod object, since there is no "Find" method. One very inefficient way would be to use the "GetFirst", "GetNext" methods to iterate the entire list of known methods. The normal way, however, will be to use "Find" methods directly on the NDNxClass, NDNxObject, NDNxSlot, or NDNxProp objects where Methods are attached (note that in this case, the

"Find" methods are named "GetPublicMethod(CStr name)" and "GetPrivateMethod(CStr name)").

NDNxObject

The list of methods and properties exposed are as follows:

```
class NDNxObject : public NDNxAtom  {

protected:
    NXOBJECT_SUBREC

public:
    static NxObjectPtr     Find(CStr name);
    static Int32     GetCount(void);
    static NxObjectPtr     GetFirst(void);
    static NxObjectPtr     GetLast(void);
    static NxObjectPtr     Create(CStr name);
          NxObjectPtr     GetNext(void);
          NxObjectPtr     GetPrevious(void);
          Str            GetName(void);
          Long     GetClientData(void);
          void     SetClientData(Long data);
          NxKBPtr     GetKB(void);
          void     SetKB(NxKBCPtr kb);
    static NxObjectPtr   CreateObject(NxObjectPtr parent, CStr name);
          Int32     Delete(void);
          Int32     DeleteObject(NxObjectPtr childObj);
          Int32     GetMethodCount(void);
          NxMethodPtr     GetIndexedMethod(Int32 index);
          Int32     GetSlotCount(void);
          NxSlotPtr     GetIndexedSlot(Int32 index);
          Int32     GetParentClassCount(void);
          NxClassPtr     GetIndexedParentClass(Int32 index);
          Int32     GetParentObjectCount(void);
          NxObjectPtr     GetIndexedParentObject(Int32 index);
          Int32     GetChildObjectCount(void);
          NxObjectPtr     GetIndexedChildObject(Int32 index);
          NxMethodPtr     GetPublicMethod(CStr name);
          NxMethodPtr     GetPrivateMethod(CStr name);
          NxSlotPtr     FindSlot(CStr name);
          NxSlotPtr     FindSlotByProp(NxPropPtr prop);
          NxObjectLinkEnum GetLinkType(NxObjectPtr child);
          Int32    MakeLinkPermanent(NxAtomPtr atom);

private:
          NDNxObject(void) {}
          ~NDNxObject(void) {}

};
```

This class allows you to access IRE Object attributes (name, client data, methods, children, parents, etc). You can also create and delete IRE Objects from this class (as with NDNxClass). When created dynamically with the Create method of NDNxObject, the Object will not have any Properties and is not attached to any Class. When created with CreateObject(), the Object will be a subobject of the parent, but again not have any properties. This C++ class is very similar to the NDNxClass class.

NDNxProp

The list of methods and properties exposed are as follows:

```
class NDNxProp : public NDNxAtom  {

protected:
    NXPROP_SUBREC

public:
    static NxPropPtr       Find(CStr name);
    static Int32       GetCount(void);
    static NxPropPtr       GetFirst(void);
    static NxPropPtr       GetLast(void);
          NxPropPtr       GetNext(void);
          NxPropPtr       GetPrevious(void);
          NxKBPtr       GetKB(void);
          void       SetKB(NxKBCPtr kb);
          Str             GetFormat(void);
          void       SetFormat(CStr format);
          Int32       GetMethodCount(void);
          NxMethodPtr       GetIndexedMethod(Int32 index);
          Str             GetName(void);
          NxMethodPtr       GetPublicMethod(CStr name);
          NxMethodPtr      GetPrivateMethod(CStr name);
          NxPropDataTypeEnum GetDataType(void);

private:
          NDNxProp(void) {}
          ~NDNxProp(void) {}

};
```

This class allows you to acccess the Property's attributes (name, format, datatype, methods, etc).. An involved example combining use of this class and the NDNxMethod class is:

```
NDNxPropPtr prop;

// iterate through all Properties
for (prop = NDNxProp::GetFirst();
     prop != NULL;
     prop = prop->GetNext()) {
             NDNxMethodPtr method;
             Str           name;
             Int32         n;

             // display name
             name = prop->GetName();
             printf("Property: %s\n", name);
             NDStr::Dispose(name);

             // iterate through all Methods of this Property
             n = prop->GetMethodCount();
             for (Int32 i=0; i<n; i++) {
                    method = prop->GetIndexedMethod(i);
                    name = method->GetName();
                    // print name, and whether public or private
                    if (method->IsPrivate()) {
                          printf("\tPrivate Method: %s\n", name);
                    } else {
                          printf("\tPublic Method: %s\n", name);
                    }
                    NDStr::Dispose(name);
             }
}
```

NDNxRule

The list of methods and properties exposed are as follows:

```
class NDNxRule : public NDNxAtom  {

protected:
    NXRULE_SUBREC

public:
    static NxRulePtr Find(CStr name);
    static Int32     GetCount(void);
    static NxRulePtr GetFirst(void);
    static NxRulePtr GetLast(void);
    static NxRulePtr GetCurrent(void);
          NxRulePtr GetNext(void);
          NxRulePtr GetPrevious(void);
          Str       GetComments(void);
          NxKBPtr   GetKB(void);
          void      SetKB(NxKBCPtr kb);
          Str       GetName(void);
          Int32     IsUnknown(void);
          Int32     IsNotknown(void);
          Int32     IsKnown(void);
          Str       GetWhy(void);
          Int32     GetInferencePriority(void);
          NxSlotPtr GetInferenceSlot(void);
          NxSlotPtr GetHypo(void);
          Int32     GetIfConditionCount(void);
          Int32     GetThenActionCount(void);
          Int32     GetElseActionCount(void);
          Str       GetIndexedIfCondition(Int32 index);
          Str       GetIndexedThenAction(Int32 index);
          Str       GetIndexedElseAction(Int32 index);
          Int32     GetValue(void);
          Int32     SuggestHypo(void);
          Int32     UnsuggestHypo(void);
          Int32     SuggestHypo2(NxRuleSugPrioEnum strategy);
          Int32     IsHypoSuggested(void);

private:
          NDNxRule(void) {}
          ~NDNxRule(void) {}

};
```

This class allows you to access attributes and contents of an IRE Rule (why, comments, name, etc). The NDNxRule class also allows you to suggest or unsuggest a rule (the method will actually cause the hypothesis of this rule to be suggested or unsuggested). You can also check if a rule has been suggested. The standard / default Suggest method uses the default suggest strategy (see nxengpub.h), while Suggest2 allows you to specify a particular strategy to be used.

Other methods of interest are the Find method, which finds a rule by the rule name (to be found in the knowledge base), and GetValue, which returns the state of the rule (true, false, unknown, or notknown).

A simple example of this class is:

```
NDNxRulePtr rule;

// find the Rule (using the Rule name)
rule = NDNxRule::Find("my first rule");
```

```
if (rule == NULL) return;

// if found, suggest the rule, start the engine, get its value
rule->Suggest();
NDNxEngine::Start();
printf("Answer = %d\n", rule->GetValue());
```

NDNxSlot

The list of methods and properties exposed are as follows:

```
class NDNxSlot : public NDNxAtom  {

protected:
    NXSLOT_SUBREC

public:
    static NxSlotPtr      Find(CStr name);
    static Int32      GetCountData(void);
    static NxSlotPtr      GetFirstData(void);
    static NxSlotPtr      GetLastData(void);
    static Int32      GetCountHypo(void);
    static NxSlotPtr      GetFirstHypo(void);
    static NxSlotPtr      GetLastHypo(void);
    static NxSlotPtr      GetCurrent(void);
    static Int32      GetSuggestListCount(void);
    static NxSlotPtr      GetIndexedSuggestList(Int32 index);
    static Int32      GetVolunteerListCount(void);
    static NxSlotPtr      GetIndexedVolunteerList(Int32 index);
          NxSlotPtr      GetNextData(void);
          NxSlotPtr      GetPreviousData(void);
          NxSlotPtr      GetNextHypo(void);
          NxSlotPtr      GetPreviousHypo(void);
          Int32      IsHypo(void);
          Str              GetName(void);
          NxPropPtr      GetProperty(void);
          NxAtomPtr      GetParent(void);
          Long             GetClientData(void);
          void             SetClientData(Long data);
          NxKBPtr      GetKB(void);
          void             SetKB(NxKBCPtr kb);
          Int32      GetMethodCount(void);
          NxMethodPtr      GetIndexedMethod(Int32 index);
          NxMethodPtr      GetPublicMethod(CStr name);
          NxMethodPtr      GetPrivateMethod(CStr name);
          Int32      GetChoiceCount(void);
          Str              GetIndexedChoice(Int32 index);
          Int32      GetContextCount(void);
          NxSlotPtr      GetIndexedContext(Int32 index);
          Str              GetStringValue(void);
          VarPtr      GetValue(void);
          void             SetValue(VarCPtr value);
          Int32      IsUnknown(void);
          Int32      IsNotknown(void);
          Int32      IsKnown(void);
          Int32      Suggest(void);
          Int32      Suggest2(NxSlotSugPrioEnum strategy);
          Int32      IsSuggested(void);
          Int32      Unsuggest(void);
          NxSlotDataTypeEnum GetDataType(void);
          Str              GetPrompt(void);
          Str              GetWhy(void);
          Str              GetComments(void);
          Str              GetFormat(void);
```

```
        void            SetFormat(CStr format);
        Str             GetQuestionWindow(void);
        Str             GetPublicInitValue(void);
        Str             GetPrivateInitValue(void);
        Int32      IsPrivate(void);
        Int32      GetInferencePriority(void);
        NxSlotPtr      GetInferenceSlot(void);
        Int32      GetInheritancePriority(void);
        NxSlotPtr      GetInheritanceSlot(void);
        Str             GetValidationHelp(void);
        Str             GetValidationExecute(void);
        Str             GetValidationFunction(void);
        BoolEnum      GetStrategy(NxSlotStratEnum strategy);
        BoolEnum      IsDefaultStrategy(NxSlotDefStratEnum strategy);
        Int32      Volunteer(VarCPtr value);
        Int32      Volunteer2(VarCPtr value,
                               NxSlotVolStratEnum strategy);

private:
        NDNxSlot(void) {}
        ~NDNxSlot(void) {}

};
```

Many of the methods of this class are similar to methods already seen in other classes.

Note that slots can be either "hypothesis" slots or "data" slots. As a hypo slot, the slot can be suggested (similar to how it was done for the NDNxRule class). As a data slot, however, the slot must be "Volunteered" for the engine to process the information. You can tell the difference, given an arbitrary slot, by using the IsHypo() method. Using the wrong method for the slot will cause an exception to be thrown. While allowed under C++, the assignment operator ("=") has not been overloaded, and you cannot get/set the value of an NDNxSlot using this operator.

Using Volunteer is analogous to Suggest. There is a default volunteer strategy that can be specified in the NDNxEngine class, allowing you to do a simple Volunteer(...). However, you can get precise control over a particular volunteer by using Volunteer2(...), which allows you to specify the strategy to be used at the time you do the volunteer (temporarily overriding the default).

Important:

Use of the "variant". A variant is basically a structure/class that can hold data of any type (for complete information, you should see varpub.h). If you want to volunteer data, you will typically have to construct a variant (either on the heap or the stack), transfer the data, invoke the method, and ensure the variant gets destroyed (or a memory leak will occur). You have a similar situation in invoking the GetValue() method, which returns a variant. Typically, you will get it, copy it somewhere else, followed by destruction of the variant. The variant supports considerable conversion of data from one type to another (eg: "1" --> 1). See the following examples.

To volunteer using a heap allocated variant:

```
NxSlotPtr     slot;
Str           answer = "hello";
NDVarPtr      var;

slot = NDNxSlot::Find("obj.prop");
```

```
if (slot == NULL) return;

var = new NDVar(answer);  // overloaded constructor creates variant
slot->Volunteer(var);
delete var;     // destroy the variant
```

To volunteer using a stack based variant:

```
NxSlotPtr    slot;
Str          answer = "hello";
NDVar        var;

slot = NDNxSlot::Find("obj.prop");
if (slot == NULL) return;

var.SetStr(answer);  // set value for already constructed variant
slot->Volunteer(&var);
// variant destruction is implicit when exit routine
```

Getting the value of a slot as a variant is done as follows (note that the typecast is required because of issues involving C / C++ versions of the variant class):

```
NxSlotPtr slot;
NDVarPtr doubleVar;

slot = NDNxSlot::Find("obj.prop");
if (slot == NULL) return;

// get variant, and print as a double
doubleVar = (NDVarPtr)slot->GetValue();
printf("double value = %.3f\n", doubleVar->CopyToDouble());
// remember to destroy it!
delete doubleVar;
```

Note there are two ways to do a "find" to get a slot. The first way is by its full name:

```
NxSlotPtr slot = NDNxSlot::Find("obj.prop");
```

While the other involves finding the various components individually:

```
NxObjectPtr obj = NDNxObject::Find("obj");
NxSlotPtr slot = obj->FindSlot("prop");
```

The former is usually the more convenient, but sometimes the latter may be required because of how information is passed to you.

Function-based Classes

There are a number of classes outside the above general interface, that are more oriented toward a specific functionality (eg: contexts, editing, libraries, etc). These will be very briefly documented here, as their functionality is more fully documented in the manuals (though their C++ interface is not).

Context API

This NDNxCtx class provides for user-controlled contexts in IRE, and is documented in the nxctxpub.h header file:

```
class NDNxCtx  {

protected:
    NXCTX_SUBREC

public:
    static BoolEnum        IsClientIdValid(NxCtxClientId id);
    static NxCtxClientIdAllocateClientId(void);
    static NxCtxPtr        GetCur(void);
    static void            SetNfyProc(NxCtxNfyProc proc);
    static void            UnsetNfyProc(NxCtxNfyProc proc);
          NxCtxPtr   SetCur(void);
          BoolEnum               IsValid(void);
          void             SetClientData(NxCtxClientId id, ClientPtr ptr);
          ClientPtr  GetClientData(NxCtxClientId id);
    NDNxCtx(void);
    ~NDNxCtx(void);

};
```

A series of examples of its use is provided by the "nxcntx" example, but typical usage includes:

```
NxCtxPtr      cxtx;
NxCtxClientId id;
Str           str;

// allocate an id (should be done only once per application)
id = NDNxCtx::AllocateClientId();
// get current context
cntx->GetCur();
// verify is valid
DBG_CHECK( cntx->IsValid() );
str = NDStr::NewSet("hello world");
// store context-specific client data
cntx->SetClientData(id, (ClientPtr)str);
```

Edit API

The two classes NDNxEdt and NDNxEdtInfo are documented in the nxedtpub.h header file, and a brief example of their use is provided by the "nxedit" example.

The NDNxEdt class provides the main edit functionality, with fully public members and data:

```
class NDNxEdt  {

public:
    NXEDT_SUBREC

public:
    NDNxEdt(void);
     ~NDNxEdt(void);
    void Reset(void);
    Int  Fill(NxAtomPtr atom);
    Int  Delete(NxAtomPtr atom);
    Int  Modify(NxAtomPtr oldAtom, NxAtomPtrPtr newAtom);
    Int  Create(NxAtomPtrPtr newAtom);
    Int  SetAtomType(Int type);
```

```
    Int  SetStr(Int code, CStr value);
    Int  SetNthStr(Int code, CStr value, Int occurrence);
    Int  GetStr(Int code, CStrPtr value);
    Int  GetNthStr(Int code, CStrPtr value, Int occurrence);
    Int  FindIndex(Int code, CStr value, IntPtr occurrence);
    Int  RemoveNthStr(Int code, Int occurrence);
    Int  RemoveStr(Int code);

};
```

The NDNxEdtInfo class is a "helper" class used with some of the public fields of the NDNxEdt class:

```
class NDNxEdtInfo  {

public:
    NXEDTINFO_SUBREC

public:
    NDNxEdtInfo(void);
    ~NDNxEdtInfo(void);
    void            Reset(void);
};
```

Note that these classes are not exception based. This may change in a future release. A complete example of its use is provided by the "nxedit" example, but typical usage includes:

```
NxAtomPtr     atom;
NxAtomPtr     newAtom;
NxEdtPtr      ptr;

ptr = new NDNxEdt();

// create a property...   Note that we should be checking for
// errors, since this is not an assertion-based API
ptr->SetAtomType(NXP_ATYPE_PROP);
ptr->SetStr(NXP_AINFO_NAME, "myProp");
ptr->SetStr(NXP_AINFO_TYPE, "Integer");
ptr->Create(&atom);

// reset the contents for reuse ...
ptr->Reset();
```

Library Control

All the necessary ND libraries have a standard set of APIs used to initialize and terminate their usage. Rather than show all of them, a representative one (for the IRE database / spreadsheet library, NxDb, found in nxdbpub.h) is shown below:

```
class NDNxDb  {

public:
    static Void LibInstall(Void );
    static Void LibLoadInit(Void );
    static Void LibExit(Void );

};
```

These are all static methods. To make a library usable would require:

```
NDNxDb::LibInstall();
NDNxDb::LibLoadInit();
```

and to terminate its usage:

```
NDNxDb::LibExit();
```

The NDNxExe class controls the NxExe library (see nxexepub.h), similar to the above.

The NDNxGfx class is for the NxGfx library (see nxgfxpub.h). It is similar to the standard library classes, except it also has methods to launch the IRE development environment (optionally starting the rules engine):

```
class NDNxGfx  {

public:
    static void LibInstall(void);
    static void LibLoadInit(void);
    static void LibExit(void);
    static void Start(void);
    static void StartRun(void);

};
```

Chapter

2 C++ Primer

This primer teaches you how to use the Intelligent Rules Element application programming interface (API) routines. It starts with very small examples and progressively builds up to more complex examples. For a complete reference to the application programming interface routines, see Chapters Three through Thirteen.

Introduction

This chapter assumes you are familiar with the Rules Element and with programming concepts. It also helps if you are familiar with C++ because the examples are written in C++. This table summarizes the information in each section of this chapter:

About the examples

- What directory do the examples assume that I'm in?
- How do I specify header files so my programs find them?

Starting small with hello.cc

- What is the basic structure of an application programming interface program?
- How do I compile, link, and execute `hello.cc`?

Using the line-mode interpreter

- Why should I use the Rules Element's line-mode interpreter?
- How do I use the line-mode interpreter?

Writing routines that the Rules Element calls

How do I...

- Pass a string from the Rules Element to a routine?
- Pass a list of atoms from the Rules Element to a routine?
- Retrieve atoms by name from the Rules Element?

Writing programs that call the Rules Element

How do I...

- Start the development environment?
- Load a knowledge base and run a Rules Element session?
- Write an interpreter for the Rules Element?
- Use the question handler?

For the advanced programmer

How do I...

- Create objects and assign slot values?
- Investigate the object base?
- Interrupt a session?
- Ask non-modal questions?
- Provide values to the Rules Element during a session?
- Use communication handlers?
- Write to the transcript window?
- Trap transcript messages?
- Compile and edit knowledge bases?
- Monitor a session?

About the Examples

This section provides some information about the examples in this chapter.

Working Directory

If you would like to review the examples in the primer as you read through the primer, you can change your working directory to the Rules Element examples directory, which contains all of the source code for the examples in the primer. The examples directory is created when you install the Rules Element. You can put the examples wherever you want.

The rest of this chapter uses variations of an example called `hello.cc` to illustrate tasks you can do with the application programming interface. The components of an application programming interface program are shown and an example of how to compile, link, and execute `hello.cc` on each platform is given.

You are also introduced to a tool that helps you quickly start interacting with the Rules Element without having to write a lot of code. It is a primitive interface called the Rules Element line-mode interpreter. The rest of the examples build on the Rules Element line-mode interpreter and gradually get more complicated.

Using the Examples

Neuron Data supplies the examples in this primer as files. They are called the Hello examples. For all the examples that follow, two files are provided: `helloN.cc` and `helloN.tkb` where N is a number between 1 and 12. `helloN.cc` contains the source code, and `helloN.tkb` contains the knowledge base to test the source code. (The Macintosh version contains also the resource files `helloN.r`, `helloN.` and the THINK project files `helloN.`)

Specifying Header Files

On Unix and VMS platforms the installation procedure should put all the header files in the correct directory, but you may have to move it or you may need to set up a special definition. On some platforms, you can specify the location in the makefile.

Starting Small - hello1.cc

In this simple example, we introduce the components of a program that uses the application programming interface. The program initializes the application programming interface, loads a knowledge base, and exits.

```
#define  ERR_LIB    HELLO

#include <nxppub.h>
#include <nxengpub.h>
#include "nxpinter.h"

#define ND_GUI          0
#define ND_IR           1
#include <nd.h>

//-----------------------------------------------------------------
//     hello: Execute routine
//-----------------------------------------------------------------

Int    hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *,
                 theAtoms)
{
       C_USE(theStr)
       C_USE(nAtoms)
       C_USE(theAtoms)

       printf("hello world!\n");
       return 1;
}

//------------------------------------------------------------------
//     main
//------------------------------------------------------------------

Int    main L2(Int, argc, char**, argv)
{
       HELLO_Init("hello1")

       ND::Init(argc, argv);
       NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
       NXPLine_Main();
       ND::Exit();

       return EXIT_OK;
}
```

Here is an explanation of the program:

```
#include <nxppub.h>
#include <nxengpub.h>
```

This is the include file supplied by Neuron Data that you need to include in order to use the application programming interface. Additional header files may be needed as you use other classes.

```
HELLO_Init("hello1");
```

This routine performs operating system-specific initializations for console-based I/O.

```
ND::Init(argc, argv);
```

This routine performs Elements Environment initializations for the specified elements such as ND_IR and ND_GUI.

```
NXPLine_Main();
```

This routine invokes a simple command interpreter as described later in this chapter, and eventually gets replaced as the primer gets more advanced.

```
ND::Exit();
```

This routine performs Elements Environment termination and clean-up.

Compiling, Linking, and Executing

Neuron Data supplies you with files to make compiling, linking, and executing easier. On most platforms, a makefile is supplied. See the ReadMe file supplied with the examples.

Using the Line-mode Interpreter

Neuron Data supplies a line-mode interpreter as one of the examples which is invoked by calling NXPLine_Main. Using the line-mode interpreter, you can start interacting with the Rules Element through the application programming interface with only a few lines of code.

For example, in the previous section, we used `hello1.cc` to load a knowledge base but we didn't do anything with it. We could have used more application programming interface routines to do tasks such as:

- Suggest a hypothesis
- Volunteer a slot value
- Start a session

However, we would have had to write more code to do all those things. The line-mode interpreter has all of the application programming interface routines embedded in it that are available as the menu commands in the development system version of the Rules Element. In `hello1.cc`, we could have loaded a knowledge base and then called the line-mode interpreter to test whether we loaded it correctly.

You can use the line-mode interpreter as a learning tool. As you learn more about the application programming interface, you'll write more of your own routines to perform testing. You'll use the line-mode interpreter less and less until you won't need it at all. It is a quick way of learning how to use the application programming interface routines.

Our examples are designed for the runtime library of the Rules Element. The first several examples in this section do not contain the code to load a knowledge base and control a session. Instead, these examples start the line-mode interpreter by calling a procedure called `NXPLine_Main`. The complete source code of the interpreter is in the file `nxpinter.cc` and is partially described in this primer.

The `NXPLine_Main()` statement gives control to the line-mode interpreter. It returns only when you exit the interpreter with the exit command.

You can compile and link this program. For more information, see the examples on compiling, linking, and executing in the previous section.

After you compile, link, and execute `hello.cc`, you'll see the following prompt on your terminal:

```
NXP>
```

This prompt is the prompt of the line-mode interpreter. You can type a question mark (?) to get the list of commands provided by the interpreter. The main commands are:

Command	Purpose
load *filename*	Loads the knowledge base `filename`
suggest *hypo*	Suggests the hypothesis `hypo`
volunteer *slot value*	Volunteers `value` into `slot`
run	Starts the inference engine
restart	Restarts the session
show atom *atomname*	Displays information about the atom `atomname`
show hypo	Displays the list of hypotheses
show data	Displays the list of data
show objects	Displays the list of objects
show classes	Displays the list of classes
?	Displays commands
show?	Displays show subcommands

You can load some of your knowledge bases or the example knowledge bases and run sessions with this interpreter. Here is an example with the satfault.tkb knowledge base:

```
NXP> load satfault.tkb
NXP> suggest possible_leak
NXP> run
Do the two displays (CRT and KDU) agree or disagree?
Enter value: AGREE
During which task did the problem occur?
Enter value: ?
       ATTACHING
       FLUID-TRANSFER
       TESTING
Enter value: TESTING
NXP> show atom possible_leak
Type: Property Slot, Hypothesis
Value Type: Boolean
Value: FALSE
NXP> exit
```

Writing Routines that the Rules Element Calls

For the rest of the examples in this chapter, we use the Unix platform.

Keep the following in mind while working with these examples:

- To compile a file, use the command line declared in the file MAKEFILE.
- The environment variables described in the Installation Guide should be properly set up. Verify them before running any hello examples.
- The examples are console-oriented. On the PC, the examples must be run under Windows and a “pseudo-console” will be started.

Hello5 and Hello11 require the development libraries to run the graphics. NXPGFX routines are not available with the RunTime libraries (default libraries linked within the MAKEFILE).

Makefiles are provided to recompile all files. Refer to the Readme file provided with the examples.

Displaying a Message (hello1)

This example illustrates how to call out from the Rules Element. From a rule, we want to call a C++ procedure that displays the message "hello world" on the screen.

The example knowledge base contains the following rule:

```
(@RULE= R1
       (@LHS=(Execute ("hello"))
       )
       (@HYPO= test_hello)
)
```

When the `Execute ("hello")` condition is evaluated by the inference engine, the Rules Element calls a hello function that you have written and installed as a handler. Of course, if you do not write a hello function but try to run the preceding knowledge base with the standard development system, you receive an error message such as the following:

```
Cannot execute hello, no handler installed
```

Our hello function displays hello world on the screen. The C++ source code is the following:

```
Int  hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
     C_USE(theStr)
     C_USE(nAtoms)
     C_USE(theAtoms)

     printf("hello world!\n");
     return 1;
}
printf displays the message "hello world!".
```

Our hello function returns an integer value of 1. The returned value is only meaningful if the Execute is called from the LHS of a rule. It determines the logical state of the condition. If your function returns 0, the condition is evaluated as FALSE, otherwise the condition is set to TRUE.

C_USE indicates that the variable passed in as an argument (required for this procedure) is, in this case, not actually used by this function. This prevents some compilers from issuing warning messages about unused variables.

Writing the hello function is not sufficient. We must also install this function inside the Rules Element kernel so that the inference engine can call it when needed. This operation is done by calling NDNxEngine::`SetHandler` in our main procedure just after the initialization of the Rules Element kernel. You need to add the following line:

```
NDNxEngine::SetExecuteHandler((NxpIProc)hello, "hello");
```

This routine tells the Rules Element kernel that the hello procedure, the first argument, should be called whenever an Execute "hello" statement is encountered.

Note: The name specified in the second argument may be different from the C++ procedure name. For example, we could pass "HelloWorld" as the third argument in which case our knowledge base must be modified (Execute "hello" becomes Execute "HelloWorld") but we can keep hello as the procedure name in our C++ source file.

You may have noticed that the C++ function is, in this case, actually a C function. This is still a valid C++ function, and will be called by the Rules Element kernel. Another valid alternative is to provide the name of a static C++ class function. For example, you could provide the name

```
MyCppClass::Hello
```

but this must be a static C++ method. The reason is that you must provide an address that is independent of any instance, since when called by the Rule Element, no instance (“this”) is implied. Note that it is possible to use another method, `NDNxEngine::SetExecuteHandler2`, to pass a C++ object (instance) for later use by your code, but you still need a non-instance function for the Rules Element to invoke, and this is beyond the scope of this particular primer.

The complete listing of our hello1.cc program is now:

```
#define ERR_LIB HELLO

#include <nxppub.h>
#include <nxengpub.h>
#include "nxpinter.h"

#define ND_GUI          0
#define ND_IR           1
#include <nd.h>

//------------------------------------------------------------------
//     hello: Execute routine
//------------------------------------------------------------------

Int    hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
       C_USE(theStr)
       C_USE(nAtoms)
       C_USE(theAtoms)

       printf("hello world!\n");
       return 1;
}
```

```
//----------------------------------------------------------------
//     main
//----------------------------------------------------------------

Int    main L2(Int, argc, char**, argv)
{
       HELLO_Init("hello1")

       ND::Init(argc, argv);
       NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
       NXPLine_Main();
       ND::Exit();

       return EXIT_OK;
}
```

This code is contained in the example file `hello1.cc`.

You can compile and link hello1.cc as described previously. Then you can run the modified version. When you get the NXP> prompt, you can test the program with the `hello1.tkb` knowledge base:

```
NXP> load hello1.tkb
NXP> suggest test_hello
NXP> run
hello world!
NXP> show atom test_hello.value
Type: Property Slot, Hypothesis
Value Type: Boolean
Value: TRUE
NXP> exit
```

The hello world! message is printed when we run the session.

Passing a String to an Execute Routine (hello2)

Our first hello routine works but is too specialized. It is impractical to write one routine for every message that we want to output. We can convert our hello routine into a generic routine that displays any string. Instead of being hard-coded in the C++ routine, the "hello world!" message is coded in the knowledge base. The modified `hello2.tkb` contains the following Execute condition:

```
(Execute  ("hello") (@STRING="hello world!";))
```

Note: If you are editing your rules with the development system, the Rules Element prompts you with a special dialog when you click into the second argument of the Execute condition in the Rule editor. In this dialog you must fill the String box with the hello world! message (without quotes). The Rules Element automatically generates the corresponding @STRING statement.

The hello function becomes:

```
//----------------------------------------------------------------
//     hello: Execute routine
//----------------------------------------------------------------

Int    hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
       C_USE(nAtoms)
       C_USE(theAtoms)

       printf("%s\n", theStr);
```

```
        return 1;
}

//-----------------------------------------------------------------
//      main
//-----------------------------------------------------------------

Int     main L2(Int, argc, char**, argv)
{
        HELLO_Init("hello2")

        ND::Init(argc, argv);
        NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
        NXPLine_Main();
        ND::Exit();

        return EXIT_OK;
}
```

The first argument, theStr, receives the string specified with the @STRING statement in the rule.

The second and third arguments allow you to pass a list of atoms, such as objects, classes, and slots, to an external routine. They are ignored in this example but will be useful for our next example. This also required the additional line (not displayed here): `#include <nxatmpub.h>`.

You can compile and link the `hello2.cc` program. Running the program with the `hello2.tkb` knowledge base gives the same results as before.

Passing a List of Atoms to an Execute Routine (hello3)

In our previous example, the hello world! message was hard coded in the rules. In many cases, it would be more interesting to pass an object slot rather than a fixed string to the Execute routine. The value of the slot can be assigned by rules and displayed by the Execute routine. Let us modify our example so that our hello routine displays the contents of the message slot of our knowledge base.

The LHS of our rule becomes:

```
(@LHS=
        (Assign   ("hello world!")(message))
        (Execute  ("hello") (@ATOMID= message.Value;))
)
```

We must modify our hello routine. Instead of receiving the “hello world!” string as first argument, the hello routine will receive a list of atoms. In this case, the list contains only one atom, the Value slot of the message object. The hello routine receives the number of atoms as second argument and a pointer to an array of atoms as third argument. The code of the hello routine becomes:

```
//-----------------------------------------------------------------
//      hello: Execute routine
//-----------------------------------------------------------------

Int     hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
        Str     str;

        // theStr is ignored in this case
        C_USE(theStr)
```

```
        if (nAtoms != 1) {
            printf("Error: hello called with %d atoms\n", nAtoms);
            return 0;
        }
        str = ((NxSlotPtr)theAtoms[0])->GetStringValue();
        if (str == NULL) {
                printf("Error: hello cannot get value\n");
                return 0;
        }
        printf("%s\n", str);
        NDStr::Dispose(str);
        return 1;
}

//------------------------------------------------------------------
//      main
//------------------------------------------------------------------

Int     main L2(Int, argc, char**, argv)
{
        HELLO_Init("hello3")

        ND::Init(argc, argv);
        NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
        NXPLine_Main();
        ND::Exit();

        return EXIT_OK;
}
```

This routine prints an error message and returns 0 if the number of atoms is not 1. Then it calls `GetStringValue` to obtain the value of the first atom in the array theAtoms.

Note: The `GetStringValue` routine should not fail in our example, but it will fail if `theAtoms[0]` is not a slot. For example, if we write `@ATOMID= message` instead of `@ATOMID= message.Value` in our rule, `theAtoms[0]` will be the message object, not the `Value` slot of the message object. Object slots have values, but objects as such do not have values, and thus the `GetStringValue` routine will fail if we pass message instead of `message.Value`.

What is returned from `GetStringValue` is a string. When you are finished strings, you must remember to dispose of any returned by Rules Element calls, or you will get memory leaks. Also, the API is exception-based, so that if there ever is an error during the call, an exception will be thrown. If you think you might get exceptions from some of your calls, you should protect them with the standard Neuron Data error recovery statements.

Once the value of the slot has been obtained by the `GetStringValue`, it is output to the screen with a `printf` statement and a success code is returned.

Retrieving Atoms by Name with Find (hello4);

Instead of passing the atom `message.Value` to the hello routine, we could find the atom `message.Value` from the hello routine. Our Execute condition becomes:

```
(Execute ("hello") (@ATOMID=message.Value;))
```

The hello routine is modified as follows:

```
//---------------------------------------------------------------
//      hello: Execute routine
//---------------------------------------------------------------

Int     hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
        NxSlotPtr       msgSlot;
        Str             locStr;

        C_USE(theStr)
        C_USE(nAtoms)
        C_USE(theAtoms)

        msgSlot = NDNxSlot::Find("message.Value");
        if (msgSlot == NULL) {
                printf("Error: hello cannot get id\n");
                return 0;
        }
        locStr = msgSlot->GetStringValue();
        if (locStr == NULL) {
                printf("Error: hello cannot get value\n");
                return 0;
        }
        printf("%s\n", locStr);
        NDStr::Dispose0(locStr);
        return 1;
}

//---------------------------------------------------------------
//      main
//---------------------------------------------------------------

Int     main L2(Int, argc, char**, argv)
{
        HELLO_Init("hello4")

        ND::Init(argc, argv);
        NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
        NXPLine_Main();
        ND::Exit();

        return EXIT_OK;
}
```

The `Find` routine is described in detail in Chapter Thirteen.

This version is more specific than the previous one because `message.Value` is hard coded in the hello routine. It is also less efficient because `Find` does a search by name in the working memory. Before, the pointer to `message.Value` was determined at compile time and passed directly to the Execute routine.

Writing Programs that Call the Rules Element

The different versions of our hello program use the Rules Element interpreter (they call `NXPLine_Main`). We will now modify the program so that it does not need the line mode interpreter.

Starting the Development Environment (hello5)

If you are working with a development system on PC (with Windows or Presentation Manager), Mac, UNIX or OpenVMS (with Motif), you can launch the development environment instead of starting the line mode interpreter. You must replace the `NXPLine_Main` routine by the following lines:

```
NDNxGfx::LibInstall();
NDNxGfx::LibLoadInit();
NDNxGfx::Start();
NDNxGfx::LibExit();
```

The first two calls initialize the graphics data structures in the development library. The next one triggers the display of the splash screen, opens the Rules Element main window, and then processes all the interface events (clicks and keystrokes) until you select the Quit option from the system menu. The final call does a final cleanup and exit of the development library.

Note: If you are using a runtime system instead of a development system, these calls are unavailable.

Starting the graphics environment allows you to test your Execute routines in the development system. You can load knowledge bases, run sessions, modify rules and objects, and browse the networks as usual. Moreover, when you edit an Execute statement in the rule or meta-slots editor, the first argument popup contains a “Copy Execute” option which allows you to choose among the Execute routines that you have declared with the `NDNxEngine::SetExecuteHandler` routine.

Loading a Knowledge Base and Running a Session (hello6)

Instead of giving control to the line mode interpreter (or to the development environment), this new version of our program will load the `hello6.tkb` knowledge base, suggest the `test_hello` hypothesis, and run the session. Only the main routine needs to be modified:

```
Int    main L2(Int, argc, char**, argv)
{
       NxSlotPtr    testHypo;
       NxKBPtr      testKB;

       HELLO_Init("hello6")

       ND::Init(argc, argv);
       NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);

       printf("loading hello6.tkb\n");
       testKB = NDNxKB::Load("hello6.tkb");
       if (testKB == NULL) {
              printf("Main: error %d while loading KB\n",
                     NDNxEngine::GetError());
              return EXIT_FAIL;
```

```
        }
        testHypo = NDNxSlot::Find("test_hello");
        if (testHypo == NULL) {
                printf("Main: error %d in get hypo id\n",
                        NDNxEngine::GetError());
                return EXIT_FAIL;
        }
        if (!testHypo->Suggest()) {
                printf("Main: error %d in suggest\n",
                        NDNxEngine::GetError());
                return EXIT_FAIL;
        }
        printf("Starting session\n");
        NDNxEngine::Start();
        ND::Exit();

        return EXIT_OK;
}
```

The code should be self explanatory. You can read the `Suggest()` and `Start()` descriptions in Chapter Seven. The `Start()` method will return when the rule session is complete.

This example also illustrates the error handling mechanism. The `NXP_` routines (except `NDNxEngine::GetError`) typically return 1 on success and 0 on failure. If a routine fails, you can call `NDNxEngine::GetError` which will return a code describing the error more precisely. In our example, the error code returned by `NDNxEngine::GetError` is included in the error message (formatted by `printf`). This will not always be the case, however. For many failures, an exception will be thrown, and 0 would never be returned in that case.

This new program is not interactive; it loads the knowledge base, suggests `test_hello`, runs the session (and thus prints the `hello world!` message) and then exits.

Writing the Interpreter (hello7)

At this point, we can write a very simple interpreter which will allow us to control our `hello` example interactively. The main routine becomes:

```
Int     main L2(Int, argc, char**, argv)
{
        int             running= 1;
        NxSlotPtr       testHypo;
        NxKBPtr         testKB;

        HELLO_Init("hello7")

        // startup: same as before without error handling
        ND::Init(argc, argv);
        NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);

        printf("loading hello7.tkb");
        testKB = NDNxKB::Load("hello7.tkb");
        if (testKB == NULL) {
                printf("Main: error %d while loading KB\n",
                        NDNxEngine::GetError());
                ND::Exit();
                return EXIT_FAIL;
        }

        testHypo = NDNxSlot::Find("test_hello");
```

```
	if (testHypo == NULL) {
		printf("Main: error %d in get hypo id\n",
			NDNxEngine::GetError());
		ND::Exit();
		return EXIT_FAIL;
	}

	while (running) {
		/* display prompt */
#if ( defined( MAC) || defined(IBMC2))
		// Must return to line because of MPW shell:
		printf("\nNXP> \n");
#else
		printf("\nNXP> ");
#endif /* MAC */
		// dispatch character
		switch (getfirstchar()) {
		case '\n':
			continue;
		case 's':
			testHypo->Suggest();
			break;
		case 'k':
			NDNxEngine::Start();
			break;
		case 'r':
			NDNxEngine::Restart();
			break;
		case 'q':
			running = 0;
			break;
		case '?':
			printf("\ns: suggest\nk: knowcess");
			printf("\nr: restart\nq: quit");
			printf("\n?: help");
			break;
		default:
			printf("invalid command");
			break;
		}
	}
	ND::Exit();

	return EXIT_OK;
}
```

`getfirstchar()` is a simple C++ routine which returns the first character of the next line you type:

```
char	getfirstchar L0()
{
	char	c;

	c = getchar();
	if (c != '\n') {
		// eat characters till end of line
		while (getchar() != '\n');
	}
	return c;
}
```

With this new version, you can run sessions with a sequence of suggest (`s`), knowcess (`k`), and restart session (`r`). You can quit (`q`) at any time.

Using Question Handlers (hello8)

With some knowledge of the C programming language, you could easily modify our basic interpreter to handle a more complex (but still simple) command language like:

```
load kb_name
suggest hypo_name
...
```

Problems will arise if the inference engine needs to ask a question during the session. If you do not provide a question procedure (or handler), the Rules Element uses its default question handler.

You can try this by modifying the hello7.tkb knowledge base. You can replace the `Assign ("hello world!") (message)` condition by `Assign (message) (message)`. As message is UNKNOWN when you start the session, the Rules Element needs to get the value of message in order to assign it with the Assign operator.

Writing a question handler is fairly simple. The question handler receives two arguments: the slot whose value is needed by the Rules Element and the prompt line associated with this slot. The code of our question handler will be the following:

```
Int     MyQuestion L2(NxSlotPtr, slot, Str, prompt)
{
       Char   answer[255];
       Char   c;
       Int    i;

       // display the prompt line
       printf(prompt);
#if (defined ( MAC) || defined(IBMC2))
       // Must return to line because of MPW shell:
       printf("\nEnter value: \n");
#else
       printf("\nEnter value: ");
#endif

       // get a line of text from the terminal
       for (i = 0; i < 254; i++) {
              c = getchar();
              // exit loop if new line
              if (c == '\n') break;
              answer[i] = c;
       }
       // terminate the string with a NULL character
       answer[i] = '\0';

       // volunteer the answer
       NDVar val;
       val.SetStr(answer);
       slot->Volunteer2(&val, NXSLOT_VOLSTRAT_QFWRD);

       // return 1 - the question has been processed
       return 1;
}
```

Merely writing the question procedure is not a sufficient modification. We must install our question procedure as a handler with a NDNxEngine::SetHandler routine. The following line must be inserted in our main procedure after the initialization of the Rules Element ND::Init(argc, argv):

```
NDNxEngine::SetHandler(NXENGINE_PROC_QUESTION,
(NxpIProc)MyQuestion);
```

Now, our simple interpreter can ask questions, and we can run the modified knowledge base with `Assign (message) (message)`. A sample session will look like:

```
NXP> s
NXP> k
What is the Value of message?
Enter value: hello world!
message.Value = hello world!
NXP>
```

For More Advanced Programmers;

Now that you're comfortable with using application programming interface routines, we can try more advanced tasks.

Accessing the Working Memory

The application programming interface allows a program to retrieve any information about the contents of the working memory. In this section we do not intend to give a complete description of this, but will instead demonstrate with a few examples the mechanisms by which the working memory can be investigated. We will also describe how the working memory can be modified by a program (creation and deletion of objects or links).

Creating Objects and Assigning Slot Values (hello9 - Part 1)

Let us modify our `hello` routine so that it creates objects inside the working memory instead of displaying a message.

The new routine will be:

```
Int     hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
        Int             i;
        Char            name[255];
        NxObjectPtr     myObject;
        NxClassPtr      myClass;
        NxPropPtr       rankProp;
        NxSlotPtr       rankSlot;

        C_USE(theStr)
        C_USE(nAtoms)
        C_USE(theAtoms)

        myClass = NDNxClass::Find("test_class");
        if (myClass == NULL) {
                printf("test_class does not exist\n");
                return 0;
        }

        rankProp = NDNxProp::Find("rank");
        if (rankProp == NULL) {
                printf("rank property does not exist\n");
```

```
                return 0;
        }

        for (i = 1; i <= 10; i++) {

                // generate object name: obj_0, obj_1, ...
                sprintf(name, "obj_%02d", i);

                // create object and link it to test_class
                myObject = myClass->CreateObject(name);

                // get the rank slot of myObject
                rankSlot = myObject->FindSlotByProp(rankProp);
                if (rankSlot == NULL) {
                        printf("rank slot was not created\n");
                        return 0;
                }
                // set rank to i
                NDVar val;
                val.SetInt32(i);
                rankSlot->Volunteer2(&val, NXSLOT_VOLSTRAT_CURFWRD);
        }
        return 1;
}
```

Note that a few new header files were added to accomplish our extra requests: nxobjpub.h, nxclspub.h, and nxprppub.h.

To run this example, you must create a class called `test_class` with one integer slot called `rank`. This new hello routine will create 10 objects called `obj_0`, `obj_1`, ..., `obj_9`. The `CreateObject` method will also attach the newly created objects to the `test_class` class. Therefore, the new objects will inherit a `rank` slot from their parent class (unless you disable the downward inheritability of `test_class.rank`).

This routine also assigns the values of the `rank` slots. For example, `obj_5.rank` receives the value 5. This is achieved by calling the FindSlotByProp method with the property to obtain the `rank` slot of the new object and then calling `Volunteer2` to assign a value to this slot.

Note that the use of `Volunteer2` relies on a data structure known as a "variant". A variant is a structure that can contain almost any datatype, and has methods to convert between various datatypes (e.g.: integer to string).

Note: By default, values assigned with Volunteer or `Volunteer2` are not set immediately. They are queued and set only when the inference engine starts or resumes its processing. As a result, requesting the value just after it has been assigned with `Volunteer2` will return the old value of the slot, not the new one. Nevertheless, you can set the `NXSLOR_VOLSTRAT_SET` bit in the strategy argument of `Volunteer2` if you want the value to be set immediately. You should read the description of the `Volunteer` routine in Chapter Thirteen for more information.

The `hello9.c` example file also contains the code described in the next section so that you can display the objects which have been created by this `hello` Execute routine.

Investigating the Object Base (hello9 - Part 2)

Now let us improve our interpreter and add two commands:

o	To list all the objects in the working memory. The temporary objects will be prefixed by a plus (+) sign. Each object is followed by the list of its slots with their current value.
c	To display the classes and their instances.

We must add two cases in our main switch statement:

```
switch (getfirstchar()) {
case '\n':
      continue;
case 'c':
      ListClasses();
      break;
case 'o':
      ListObjects();
      break;
// continues as before
case 's':
      ...
```

The code for the new functions is the following:

```
//-----------------------------------------------------------------
//     ListSlots - lists slots of one object with their values
//-----------------------------------------------------------------

void   ListSlots L1(NxObjectPtr, obj)
{
       Int32         len;
       Int32         i;
       NxSlotPtr     slot;
       Str           buf;

       // get number of slots
       len = obj->GetSlotCount();

       for (i = 0; i < len; i++) {

              // get ith instance
              slot = obj->GetIndexedSlot(i);

              // get its name and print it
              buf = slot->GetName();
              printf("\n\t%s", buf);
              NDStr::Dispose0(buf);

              // get its value and print it
              buf = slot->GetStringValue();
              printf(" = %s", buf);
              NDStr::Dispose0(buf);
       }
}

//-----------------------------------------------------------------
//     ListObjects - lists objects followed by their slot values
//-----------------------------------------------------------------

void   ListObjects L0()
{
       NxObjectPtr   obj;
```

```
	Int32		type;
	Str		buf;

	obj = NDNxObject::GetFirst();

	while (obj) {

		// get object name
		buf = obj->GetName();

		// is it a temporary object
		type = obj->GetType();

		// print the object
		if (type & NXATOM_ATYPE_TEMP) printf("\n+ %s", buf);
		else printf("\n%s", buf);
		NDStr::Dispose0(buf);

		ListSlots(obj);

		// get next object
		obj = obj->GetNext();
	}
}

//-----------------------------------------------------------------
//	ListInstances - displays list of instances of a class
//	Called by ListClasses
//-----------------------------------------------------------------

void	ListInstances L1(NxClassPtr, clas)
{
	Int32		len;
	Int32		i;
	NxObjectPtrobj;
	Str		buf;

	// get number of instances
	len = clas->GetChildObjectCount();

	for (i = 0; i < len; i++) {
		// get ith instance
		obj = clas->GetIndexedChildObject(i);

		// get its name and print it
		buf = obj->GetName();
		printf("\n\t%s", buf);
		NDStr::Dispose0(buf);
	}
}

//-----------------------------------------------------------------
//	ListClasses - lists classes with their instances
//-----------------------------------------------------------------

void	ListClasses L0()
{
	NxClassPtr	clas;
	Str		buf;

	clas = NDNxClass::GetFirst();

	while (clas) {
```

```
            // get class name and print it
            buf = clas->GetName();
            printf("\n%s", buf);
            NDStr::Dispose0(buf);

            // display list of instances
            ListInstances(clas);

            // get next class
            clas = clas->GetNext();
      }
}
```

To test this version, you can display the list of classes and list of objects before and after having run a session which calls the `hello` Execute routine. You should see the dynamic objects created by the `hello` routine. You can also check that dynamic objects are deleted when the session is restarted.

This example illustrates the two ways to access the elements of a list.

- With the first protocol, `GetFirst` is used to return the first atom of the specified type. Then using the current atom instance, a call is made to `GetNext`, which returns the next atom of the same type or NULL if the end of list has been reached.
- With the second protocol, `GetChildObjectCount` or `GetSlotCount are` called to return the number of atoms in the specified collection. Then, using `GetIndexedChildObject` or `GetIndexedSlot`, and an integer i ranging from 0 to `len-1` (where len is the number of atoms returned by the first method), the `(i+1)th` atom is returned.

Remarks on Accessing Information

You can experiment with other methods of the `NDNxClass`, `NDNxObject`, and `NDNxSlot` classes (as well as the other IRE C++ classes) and increase the power of our interpreter. As exercises, you can write a routine which will delete all the instances of a class (provided that they are dynamic objects), or a routine which recursively displays the subclasses, instances, subobjects, and slots of a given class or object (full right expand in the object network, but displayed as text with different indentation levels). You can also try to display the text of rules, the meta-slot information, the strategy settings, etc.

Confusing the different atom types (classes, objects, properties, slots, hypotheses, data, etc.) causes most of the problems encountered by developers during early stages of their development. The most common sources of confusion are:

- Slots (hypotheses or data are slots) and objects. Slots have values (obtained with `GetValue` or `GetStringValue`), but objects do not have values. For example, the slot `tank1.pressure` has a value, but the object `tank1` does not have one. The risk of confusion is greater with a slot name like `check_tank1` (which may be a hypothesis). The slot is in fact `check_tank1.Value` (even if it is usually displayed without the `.Value` part), not the object `check_tank1` which does not have a value.

- Slots and properties. Slots have values, properties do not have values. For example tank1.pressure is a slot but pressure is a property.

It is also important to remember that retrieving the current information from the working memory, never triggers the inference or inheritance mechanisms.

Advanced Control

With the material described in the previous sections of this primer, you should be able to write external routines and to control simple applications: load a knowledge base, suggest or volunteer, start the inference engine, and then obtain the final results.

In complex applications, you may need to interrupt the inference engine and resume processing afterwards. This section should allow you to understand how you can control the inference engine in the context of an embedded application. The problem of inputting values (i.e. from a data acquisition program) during a session will also be addressed in this section.

Interrupting a Session (hello10 - Part 1)

When you call `NDNxEngine::Start()` to start a session, it starts the inference engine and returns only when the session is finished (the agenda of the inference engine is empty) unless you interrupt the session with a `NDNxEngine::Stop()` during the session.

Since the caller of `NDNxEngine::Start()` doesn't receive control until the rule session is complete, you can only stop the session from a handler that you have set and that will be called by the inference engine.

The execution of a simple application that does not interrupt the session is described by the following flow chart (in this example, the inference engine calls one Execute routine and asks only one question during the whole session):

If you want to interrupt the session, call `NDNxEngine::Stop()` from the execute routine. We can demonstrate this with the following Execute routine and rule:

```
Int     hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
        C_USE(theStr)
        C_USE(nAtoms)
        C_USE(theAtoms)

        while (1) {
#ifdef MAC
                // Must return to line because of MPW shell:
                printf("\nDo you want to interrupt the session (y or
                              n)? : \n");
#else
                printf("\nDo you want to interrupt the session (y or
                              n)? : ");
#endif /* MAC */
                switch (getfirstchar()) {
                case 'y':
                        NDNxEngine::Stop();
                        return 1;
```

```
            case 'n':
                  return 1;
            case '\n':
                  break;
            default:
                  printf("\nInvalid answer");
                  break;
            }
      }
      /* NOT REACHED */
}

(@RULE=test_rule
      (@LHS=
            (Execute      ("hello"))
            (Assign       (message)(message))
      )
      (@HYPO=test_hello)
)
```

If you answer y when you are prompted by the hello routine, the Rules Element will not prompt you for the value of `message`.

Now, we need to modify the main routine so that we can resume the session after the interruption. One way would be to bind a new command character to the `NDNxEngine::Continue()` routine. We can also reuse the `k` character. Typing `k` will start or resume the session, as appropriate. The main routine becomes (the new code is in bold typeface):

```
Int   main L2(Int, argc, char**, argv)
{
      int           running= 1;
      int           restarted = 1;
      NxSlotPtr     testHypo;
      NxKBPtr       testKB;

      HELLO_Init("hello10")

      // startup: same as before without error handling
      ND::Init(argc, argv);

      NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);
      NDNxEngine::SetHandler(NXENGINE_PROC_QUESTION,
                             (NxpIProc)MyQuestion);

      printf("loading hello10.tkb");
      testKB = NDNxKB::Load("hello10.tkb");
      if (testKB == NULL) {
            printf("Main: error %d while loading KB\n",
                  NDNxEngine::GetError());
            ND::Exit();
            return EXIT_FAIL;
      }

      testHypo = NDNxSlot::Find("test_hello");
      if (testHypo == NULL) {
            printf("Main: error %d in get hypo id\n",
                  NDNxEngine::GetError());
            ND::Exit();
            return EXIT_FAIL;
      }

      while (running) {
            // display prompt
```

```
#ifdef MAC
            // Must return to line because of MPW shell:
            printf("\nNXP> \n");
#else
            printf("\nNXP> ");
#endif /* MAC */
            // dispatch character
            switch (getfirstchar()) {
            case '\n':
                  continue;
            case 'c':
                  ListClasses();
                  break;
            case 'o':
                  ListObjects();
                  break;
            case 's':
                  testHypo->Suggest();
                  break;
            case 'k':
                  if (restarted) {
                        restarted = 0;
                        NDNxEngine::Start();
                  } else {
                        NDNxEngine::Continue();
                  }
                  break;
            case 'r':
                  NDNxEngine::Restart();
                  restarted = 1;
                  break;
            case 'q':
                  running = 0;
                  break;
            case '?':
                  printf("\nc: classes\no: objects");
                  printf("\ns: suggest\nk: knowcess");
                  printf("\nr: restart\nq: quit");
                  printf("\n?: help");
                  break;
            default:
                  printf("invalid command");
                  break;
            }
      }
      ND::Exit();

      return EXIT_OK;
}
```

With these modifications, you can interrupt the session when you are prompted by the `hello` routine. At this point, you can list the objects and the classes, and then resume the session by typing `k`. The inference engine will resume its processing and prompt you for the value of `message`.

Non-modal Questions (hello10 - Part 2)

Our current question handler is modal, which means that when the question handler prompts the user, the user must answer the question. The user cannot examine the list of objects and values before answering, nor can he decide to restart the session. A non-modal question handler allows the user to delay answering the question and gives him access to all the commands of the interpreter.

One solution to this problem would be to call a command dispatcher (like our main switch statement) from the question handler. This would make the program behave as expected but introduces a major design flaw in the program. If your question handler dispatcher lets the user restart the session, suggest a hypothesis and start a session, you may end up with a stack of routines like:

Gray lines indicate that the Rules Element kernel procedures are pushed on the stack. The problem is that `MyQuestion` is called recursively. The inner `NDNxEngine::Start()` routine will never receive a meaningful answer from its question handler (if the latter ever returns) because another session has been started in the meantime. The end result is that we have pushed procedures uselessly on the stack and nothing prevents the user from stacking more `NDNxEngine::Start()` routines.

The remedy is to have the question handler interrupt the session and return TRUE without having volunteered an answer. Then the initial `NDNxEngine::Start()` routine will return to its caller (the command dispatcher). The question will be asked again later when the user resumes the session with our `k` command.

The code of the non-modal question handler is the following:

```
Int     MyQuestion L2(NxSlotPtr, slot, Str, prompt)
{
        char   answer[255];
        char   c;
        Int    i;

        // display the prompt line
        printf(prompt);
#if (defined ( MAC) || defined(IBMC2))
        // Must return to line because of MPW shell:
        printf("\nEnter value: \n");
#else
        printf("\nEnter value: ");
#endif /* MAC */

        // get a line of text from the terminal
        for (i = 0; i < 254; i++) {
                c = getchar();
                if (i == 0 && c == '!') {
                        // eat characters till end of line
                        while (getchar() != '\n');
                        NDNxEngine::Stop();
                        return 1;
                }
                // exit loop if new line
                if (c == '\n') break;
                answer[i] = c;
        }
        // terminate the string with a NULL character
        answer[i] = '\0';

        // volunteer the answer
        NDVar val;
        val.SetStr(answer);
        slot->Volunteer2(&val, NXSLOT_VOLSTRAT_QFWRD);

        // return 1 - the question has been processed
        return 1;
}
```

The changes are indicated in bold. The user can escape to the main command dispatcher by typing ! instead of answering the question. In the main command dispatcher, the user can resume his session by typing k.

Entering Values During a Session

In a real time environment, such as process control, your Rules Element application receives data values or notifications (alerts) while a session is running. You must be able to process these incoming events.

The easiest case is when values are entered synchronously. This happens if your application needs to poll a serial port in order to get its data. You can install a polling handler which will be called by the inference engine at each inference cycle.

Sample code would look like:

```
int    MyPolling()
{
       Char          theStr[MAXDATASIZE];
       NxClassPtr    myClass;
       NxObjectPtr   myObject;
       NxSlotPtr     mySlot;

       while (GetStringFromPort(theStr)) {
              // data is present on the input line
              // GetStringFromPort will copy it into theStr

              // eventually create a new object
              myClass = NDNxClass::Find(...);
              myObject = myClass->CreateObject(...);

              // volunteer theStr into a slot (dynamic or not)
              // with appropriate strategy
              mySlot->Volunteer(...);

       }
       return 1;
}
```

The polling procedure must be installed in the initialization part of your program:

```
NDNxEngine::SetHandler(NXENGINE_PROC_POLLING,
(NxpIProc)MyPolling);
```

If the polling procedure returns TRUE, the Rules Element will not call its default polling procedure after `MyPolling`. The default polling procedure is a NO OP (no operation) in the runtime version, but it is used to check the interrupt button of the session control window in the development version of the Rules Element. In this latter case, returning TRUE will disable the interrupt mechanism.

If the values are input by an asynchronous mechanism (interrupts, ASTs on VMS, signals on UNIX), you should not create objects or set values asynchronously (in the interrupt handler or the AST routine) because this may create an inconsistent inference state and corrupt the working memory. Instead, you should set up an internal queue, queue the values asynchronously, and let the inference engine process them synchronously from the polling handler. The code of a typical polling handler will be very similar to the synchronous case described earlier, the

GetStringFromPort routine being replaced by a GetDataFromQueue routine.

Customizing the User Interface

You may also need to customize the user interface of the Rules Element (i.e. to integrate the Rules Element with the existing interface of your application in the case of a fully embedded application). This section will explain how you can use NDNxEngine::SetHandler to control the interaction between the inference engine and its interface.

Using Communication Handlers

The user interface of a Rules Element application can be completely customized with the application programming interface. The communication between the Rules Element kernel and the user interface is controlled by the following handlers:

- NXENGINE_PROC_ALERT
- NXENGINE_PROC_APROPOS
- NXENGINE_PROC_DECRYPT
- NXENGINE_PROC_ENCRYPT
- NXENGINE_PROC_GETDATA
- NXENGINE_PROC_GETSTATUS
- NXENGINE_PROC_NOTIFY
- NXENGINE_PROC_PASSWORD
- NXENGINE_PROC_QUESTION
- NXENGINE_PROC_SETDATA

The Question handler has already been described in this primer.

The Alert handler is called by the Rules Element kernel when an error occurs, or if the user needs to confirm an action (in the development environment an alert dialog appears on the screen).

The Apropos handler is called by the inference engine when a Show statement is executed.

These three handlers (Question, Alert, Apropos) are very specialized. The last four handlers are much more general, and handle all the other communications between the kernel and its interface: sending text to the transcript, getting text from the rule editor window in order to compile a rule, controlling the "select a data type for ..." window during a compilation, . . .). There are, in fact, two bi-directional communication channels between the kernel and the interface:

- A control channel which notifies (NXENGINE_PROC_NOTIFY) the interface when atoms are modified in the working memory. In the other direction, the kernel can query the status of an interface window (NXENGINE_PROC_GETSTATUS).
- A data channel which allows the kernel to send information to a window (NXENGINE_PROC_SETDATA), and to request information from a window (NXENGINE_PROC_GETDATA). These could occur, for

example, when outputting text into the transcript in the first case, and when compiling a rule in the second case.

The role of the communication handlers is summarized in the following diagram:

To customize the user interface, you must install your own communication handlers. The description of `NDNxEngine::SetHandler` in Chapter Seven provides information about the arguments of the different handlers and the valid combinations of arguments which a user program is allowed to process. In this primer, we will illustrate the use of the communication handlers with a couple of examples.

Writing in the Transcript (hello11)

Note: At this time, it is not possible with the C++ API to write a message to the transcript window of the development environment.

In this example, we will use one of the existing communication channels. We will write an execute routine that writes a message to the console window. This example is relevant only if you are programming with a development version of the Rules Element. The source code is the following:

```
#define ERR_LIB  HELLO

#include <nxppub.h>
#include <nxengpub.h>
#include <nxatmpub.h>
#include "nxpinter.h"

#define ND_GUI          1
#define ND_IR           1
#include <nd.h>

#if !defined(MAC) && ND_GUI
       #include <nxgfxpub.h>
#endif

ERR_DECLARE

//-----------------------------------------------------------------
//     hello: Execute routine
//-----------------------------------------------------------------

C_EXTERNC void  STR_Printf(CStr fmt, ... );

Int    hello L3(Str, theStr, Int, nAtoms, NxAtomPtr *, theAtoms)
{
       C_USE(nAtoms)
       C_USE(theAtoms)

       STR_Printf("*** %s\n",theStr);
       return 1;
}

//-----------------------------------------------------------------
//     main
//-----------------------------------------------------------------

Int    main L2(Int, argc, char**, argv)
{
```

```
        HELLO_Init("hello11")

        ND::Init(argc, argv);
        NDNxEngine::SetExecuteHandler("hello", (NxpIProc)hello);

        //
        // MAC VERSION CANNOT LAUNCH GRAPHIC ENVIRONMENT FROM A
        // COMMAND-LINE PROGRAM.  You must relink the entire rules
        // development system in order to use the graphic
        // environment.
        //

#if !defined(MAC) && ND_GUI
        NDNxGfx::LibInstall();
        NDNxGfx::LibLoadInit();
        NDNxGfx::Start();
        NDNxGfx::LibExit();
#endif
        ND::Exit();

        return EXIT_OK;
}
```

This program starts the interactive interface. From the expert menu, you can load the hello11.tkb knowledge base, suggest test_hello and start the session. The hello world message should be logged on your console when we run the session.

Trapping Transcript Messages (hello12)

In the previous example, we did not really customize the user interface of the Rules Element. Instead, we used the existing user interface (transcript window) to display one of our messages.

Now, let us suppose that we run the Rules Element from a character based terminal and that we want to trap the transcript messages in order to display them on the screen. Instead of using one of the communication channels (`SetData` channel), we want to provide our own communication channel which will output the messages on the screen. This is achieved by installing a custom `SetData` handler. The code of our `SetData` handler is the following:

```
Int  MySetData L4(Int, winId, Int32, ctrlId, Int32, index,
                    Str, thePtr)
{
        C_USE(ctrlId)
        C_USE(index)

        if (winId != NXENGINE_WIN_TRAN) return 0;
        if (thePtr == 0) return 0;
        STR_Printf("\n%s", thePtr);
        return 1;
}
```

If your handler returns FALSE, the Rules Element will call its default SetData handler afterwards. You must remember that the Rules Element uses the SetData handler for all its communication with the user interface. It is thus very important to return FALSE if your SetData handler does not process the routine, especially if your program has started the development interface with the `NDNxGfx::Start()` call. In this example, our handler returns TRUE. As a result the Rules Element will not log the messages in the transcript window. If we modify `MySetData` and let it return FALSE in any case, transcript messages will be displayed onto the screen by our `SetData` handler and logged into transcript by the default `SetData` handler which is called afterwards by the Rules Element kernel.

We must install this handler with a `NDNxEngine::SetHandler` routine in the initialization of our program:

```
NDNxEngine::SetHandler(NXENGINE_PROC_SETDATA,
(NxpIProc)MySetData);
```

This code could seem sufficient to trap the transcript messages. In fact, it will only work if we are running from the development interface with the transcript enabled. The reason is that before running a session, the Rules Element queries the interface to know if the transcript window is enabled or not. This refinement has been introduced to avoid formatting useless messages and thus speed up the inference engine when the trace information is not requested.

The interface is queried with the `GetStatus` handler. In order to make our example work, we must also provide our own version of the `GetStatus` handler:

```
Int     MyGetStatus L3(Int, winId, Int32, code, Str, thePtr)
{
   if (winId != NXENGINE_WIN_TRAN || code !=
        NXENGINE_GS_ENABLED)
       return 0;
   *(IntPtr)thePtr = 1;
   return 1;
}
```

We must also install this handler in the initialization of our program:

```
NDNxEngine::SetHandler(NXENGINE_PROC_GETSTATUS,
(NxpIProc)MyGetStatus);
```

Compiling and Editing Knowledge Bases

With the application programming interface, you could also rewrite the development environment of the Rules Element and, for example, provide rule or object editors which run on character based terminals. You can also use the compilation function to compile rules which have been generated automatically by a program.

The `NXEdt` routines are described in Chapter Six. With the `NxKB::Save` routine, you can save knowledge bases which have been created or modified by your program.

Chapter

3 *NxAtom Class*

Error Return Codes

NxAtomErrEnum

Enumerated type that defines the error types of the atom class.

Identifier	Description
NXATOM_ERR_NOERR	Call was successful
NXATOM_ERR_INVARG1	First argument of call is invalid
NXATOM_ERR_INVARG2	Second argument of call is invalid
NXATOM_ERR_INVARG3	Third argument of call is invalid
NXATOM_ERR_INVARG4	Fourth argument of call is invalid
NXATOM_ERR_INVARG5	Fifth argument of call is invalid
NXATOM_ERR_INVARG6	Sixth argument of call is invalid
NXATOM_ERR_NOTFOUND	No atom of the right type was found
NXATOM_ERR_INVATOM	Some atoms saved in the file are invalid
NXATOM_ERR_UNKNOWN	The value of the atom is UNKNOWN
NXATOM_ERR_NOTKNOWN	The value of the atom is NOTKNOWN
NXATOM_ERR_INVSTATE	Argument is in an invalid state
NXATOM_ERR_MATHERROR	A floating point error occurred
NXATOM_ERR_NOTAVAIL	This information is not available
NXATOM_ERR_COMPILEPB	Compilation of the new atom did not succeed and the error was not reported correctly
NXATOM_ERR_PROTPB	A program security error has occurred
NXATOM_ERR_FILEOPEN	File could not be opened
NXATOM_ERR_FILEEOF	End of file encountered unexpectedly
NXATOM_ERR_FILEREAD	Error reading the file
NXATOM_ERR_FILEWRITE	Error writing the file
NXATOM_ERR_FILESEEK	Error seeking the file
NXATOM_ERR_SYNCERROR	The parser has lost its synchronization. The contents of the file may be corrupted.
NXATOM_ERR_FORMATERROR	The file header is invalid
NXATOM_ERR_NOMEMORY	Memory allocation failed
NXATOM_ERR_ABORT	Compilation was aborted by user, or the description was incomplete and no interface was provided to prompt the user
NXATOM_ERR_SYNTAX	The text file contains a syntax error
NXATOM_ERR_INTERNAL	Some internal consistency check failed
NXATOM_ERR_VOLINVAL	VolunteerValidate handler returned FALSE
NXATOM_ERR_SAVEDISABLED	Saving of KBs has been disabled
NXATOM_ERR_VALIDATEERROR	Data validation error ... data failed validation
NXATOM_ERR_VALIDATEMISSING	Data validation error ... data is missing
NXATOM_ERR_VALIDATEUSER	Data validation error ... user rejected data
NXATOM_ERR_INVARG7	Seventh argument of call

Find Atom by Name and Type

NxAtomTypeEnum

Enumerated type that defines atom varieties.

Identifier	Description
NXATOM_ATYPE_DATA	slot used as data
NXATOM_ATYPE_HYPO	slot used as hypo
NXATOM_ATYPE_PERM	permanent object
NXATOM_ATYPE_TEMP	temporary object
NXATOM_ATYPE_MASK	mask
NXATOM_ATYPE_NONE	type unspecified
NXATOM_ATYPE_CLASS	class
NXATOM_ATYPE_OBJECT	object (input type)
NXATOM_ATYPE_PERMOBJECT	permanent object (return / output type only)
NXATOM_ATYPE_TEMPOBJECT	temporary object (return / output type only)
NXATOM_ATYPE_PROP	property
NXATOM_ATYPE_SLOT	slot (input type)
NXATOM_ATYPE_DATASLOT	data slot (return / output type only)
NXATOM_ATYPE_HYPOSLOT	hypo slot (return / output type only)
NXATOM_ATYPE_RULE	rule
NXATOM_ATYPE_LHS	condition of a rule or method (Left-Hand-Side)
NXATOM_ATYPE_RHS	DO action of rule or method (Right-Hand-Side)
NXATOM_ATYPE_CACTIONS	If Change method
NXATOM_ATYPE_SOURCES	Order of Sources method
NXATOM_ATYPE_KB	Knowledge base
NXATOM_ATYPE_EHS	ELSE action of rule or method (Else-Hand-Side)
NXATOM_ATYPE_METHOD	Method
NXATOM_ATYPE_CONTEXT	Context link

Find

static NxAtomPtr NDNxAtom::Find(CStr *name*, **NxAtomTypeEnum** *code***);**

Method to get an atom pointer given its name. `code' is an optional hint to the search process indicating the type of atom being searched for. This can greatly speed up the search, but is not required. Returns the atom pointer, if found, otherwise NULL.

SetInfo API

Use of this API is discouraged. All the various enumerated options should be available via other means.

NxAtomSAInfoEnum

Enumerated type that defines SetInfo options.

Identifier	Description
`NXATOM_SAINFO_CURRENTKB`	set the current or default knowledge base
`NXATOM_SAINFO_PERMLINK`	changes the links of an atom to permanent
`NXATOM_SAINFO_MERGEKB`	merges two knowledge bases into one
`NXATOM_SAINFO_INKB`	sets the knowledge base that an atom belongs to
`NXATOM_SAINFO_PERMLINKKB`	changes all links in a knowledge base to permanent
`NXATOM_SAINFO_AGDVBREAK`	sets / unsets agenda breakpoints on hypotheses
`NXATOM_SAINFO_INFBREAK`	sets / unsets inference breakpoints on atoms
`NXATOM_SAINFO_DISABLESAVEKB`	disables the saving of knowledge bases from the API

SetInfo

static Int32 NDNxAtom::SetInfo(NxAtomPtr *atom1*, **NxAtomSAInfoEnum** *code*, **NxAtomPtr** *atom2*, **Int32** *optInt***);**

Sets various types of information about this atom. See the API reference manual for much more extensive information on allowed codes, etc. Returns an integer status.

GetInfo API

Use of this API is discouraged. All the various enumerated options should be available via other means.

NxAtomGAInfoEnum

Enumerated type that defines GetInfo options.

Identifier	Description
`NXATOM_GAINFO_PUBLIC`	mask / flag for public information
`NXATOM_GAINFO_PRIVATE`	mask / flag for private information
`NXATOM_GAINFO_CURSTRAT`	mask / flag for current strategy information
`NXATOM_GAINFO_MLHS`	mask for Left-Hand-Side information
`NXATOM_GAINFO_MRHS`	mask for Right-Hand-Side information
`NXATOM_GAINFO_MEHS`	mask for Else-Hand-Side information
`NXATOM_GAINFO_MASK`	mask
`NXATOM_GAINFO_NAME`	returns the string name of the atom
`NXATOM_GAINFO_TYPE`	returns the type of the atom (not the datatype)
`NXATOM_GAINFO_VALUETYPE`	returns the datatype of the atom
`NXATOM_GAINFO_VALUE`	returns the value of the atom
`NXATOM_GAINFO_NEXT`	returns information about the next atom
`NXATOM_GAINFO_PREV`	returns information about the previous atom
`NXATOM_GAINFO_PARENTOBJECT`	returns information about the parent object(s) of an object

Identifier	Description
NXATOM_GAINFO_CHILDOBJECT	returns information about the child object(s) of an object or class
NXATOM_GAINFO_PARENTCLASS	returns information about the parent class(es) of an object or class
NXATOM_GAINFO_CHILDCLASS	returns information about the child class(es) of a class
NXATOM_GAINFO_SLOT	returns information about the properties of an object or class
NXATOM_GAINFO_LINKED	returns information about the type of link between a class or an object and another class or object
NXATOM_GAINFO_CHOICE	returns the choice of values (as displayed in the session control window) for a given slot
NXATOM_GAINFO_PARENT	returns information about the parent of an atom
NXATOM_GAINFO_SUGGEST	returns whether a hypothesis is suggested or not
NXATOM_GAINFO_CURRENT	returns information about the current atoms in the inference engine
NXATOM_GAINFO_HYPO	returns the hypothesis of a rule
NXATOM_GAINFO_LHS	returns information about the conditions of a rule or method
NXATOM_GAINFO_RHS	returns information about the DO actions (Right-Hand-Side of a rule or method
NXATOM_GAINFO_CACTIONS	returns the text of the If Change method conditions or actions
NXATOM_GAINFO_SOURCES	returns the text of the Order of Sources methods attached to an atom
NXATOM_GAINFO_PROP	returns the property of a specified slot
NXATOM_GAINFO_HASMETA	returns whether or not a slot has meta-information defined for it
NXATOM_GAINFO_INFCAT	returns the inference priority number attached to an atom
NXATOM_GAINFO_INHCAT	returns the inheritance priority number attached to an atom
NXATOM_GAINFO_INHDEFAULT	returns whether or not the slot inheritability of the atom follows the default (global strategy)
NXATOM_GAINFO_INHUP	returns whether or not the slot is upward inheritable
NXATOM_GAINFO_INHDOWN	returns whether or not the slot is downward inheritable
NXATOM_GAINFO_INHVALDEFAULT	returns whether or not the inheritability of the value of the atom follows the default (global strategy)
NXATOM_GAINFO_INHVALUP	returns whether or not the value of the atom is upward inheritable
NXATOM_GAINFO_INHVALDOWN	returns whether or not the value of the atom is downward inheritable
NXATOM_GAINFO_DEFAULTFIRST	returns whether or not the inheritance strategy for the atom follows the default (global strategy)
NXATOM_GAINFO_PARENTFIRST	returns whether the inheritance search for the atom should begin by searching the parent objects of the atom or the classes to which the atom belongs
NXATOM_GAINFO_BREADTHFIRST	returns whether the inheritance search for the atom is done in a breadth first or depth first manner
NXATOM_GAINFO_PROMPTLINE	returns the prompt line information attached to the atom

Identifier	Description
NXATOM_GAINFO_DEFVAL	returns the initvalue for the slot (if any) as a string
NXATOM_GAINFO_INHOBJUP	returns whether or not object slots are inheritable upwards
NXATOM_GAINFO_INHOBJDOWN	returns whether or not object slots are inheritable downwards
NXATOM_GAINFO_INHCLASSUP	returns whether or not class slots are inheritable upwards
NXATOM_GAINFO_INHCLASSDOWN	returns whether or not class slots are inheritable downwards
NXATOM_GAINFO_PWTRUE	returns whether or not the context propagation is enabled on TRUE hypotheses
NXATOM_GAINFO_PWFALSE	returns whether or not the context propagation is enabled on FALSE hypotheses
NXATOM_GAINFO_PWNOTKNOWN	returns whether or not the context propagation is enabled on NOTKNOWN hypotheses
NXATOM_GAINFO_EXHBWRD	returns whether or not exhaustive backward chaining is enabled
NXATOM_GAINFO_PFACTIONS	returns whether or not the assignments done in the RHS of rules or in methods are forwarded
NXATOM_GAINFO_SUGLIST	returns the list of hypotheses kept in the suggest selection
NXATOM_GAINFO_VOLLIST	returns the list of slots kept in the volunteer selection
NXATOM_GAINFO_INFATOM	returns the inference priority atom attached to the atom
NXATOM_GAINFO_INHATOM	returns the inheritance priority atom attached to the atom
NXATOM_GAINFO_CONTEXT	returns the hypotheses that are inthe context of a given hypothesis
NXATOM_GAINFO_KBID	returns the knowledge base to which the atom belongs
NXATOM_GAINFO_SOURCESON	returns whether or not Order of Sources are enabled
NXATOM_GAINFO_CACTIONSON	returns whether of not If Change methods are enabled
NXATOM_GAINFO_COMMENTS	returns the comments attached to the atom
NXATOM_GAINFO_FORMAT	returns the format information attached to the atom
NXATOM_GAINFO_WHY	returns the why information attach to the atom
NXATOM_GAINFO_MOTSTATE	returns information about the current state of the inference engine
NXATOM_GAINFO_PTGATES	returns whether or not forward chaining through gate is enabled
NXATOM_GAINFO_CLIENTDATA	returns the client or user information attached to an atom
NXATOM_GAINFO_VERSION	returns the names and version number of the software componenets included in the package used
NXATOM_GAINFO_SELF	returns the name or atom of the current SELF atom
NXATOM_GAINFO_PROCEXECUTE	returns the number of execute routines installed or the name of the execute handler
NXATOM_GAINFO_FOCUSPRIO	returns the priority of the hypotheses on the agenda
NXATOM_GAINFO_AGDVBREAK	returns whether a specific hypothesis has an agenda break point set for it
NXATOM_GAINFO_INFBREAK	returns whether a specific rule, condition, method, slot, object, class, or property has an inference breakpoint set on it

Identifier	Description
NXATOM_GAINFO_BWRDLINKS	returns the backward links from a hypothesis to its rules
NXATOM_GAINFO_FWRDLINKS	returns the forward links from a slot to the conditions
NXATOM_GAINFO_KBNAME	retuns the name of a knowledge base, given its atom
NXATOM_GAINFO_CURRENTKB	returns the current knowledge base containing the atom
NXATOM_GAINFO_VALUELENGTH	returns the length of a string slot value
NXATOM_GAINFO_SOURCESCONTINUE	returns whether or not Order of Sources methods will be fully executed even after a value is determined
NXATOM_GAINFO_CACTIONSUNKNOWN	returns whether or not If Change methods will also be execute when the slot if set to UNKNOWN
NXATOM_GAINFO_VALIDUSER_OFF	returns whether or not the validation of values entered by the end user is disabled
NXATOM_GAINFO_VALIDUSER_ACCEPT	returns whether or not the validation of values entered by the end user is enabled and the value accepted automatically if the validation expression is incomplete
NXATOM_GAINFO_VALIDUSER_REJECT	returns whether or not the validation of values entered by the end user is enabled and the value rejected automatically if the validation expression is incomplete
NXATOM_GAINFO_VALIDENGINE_OFF	returns whether or not the validation of values set by the engine is enabled
NXATOM_GAINFO_VALIDENGINE_ACCEPT	returns whether or not the validation of values set by the engine is enabled and the value accepted automatically if the validation expression is incomplete
NXATOM_GAINFO_VALIDENGINE_REJECT	returns whether or not the validation of values set by the engine is enabled and the value rejected automatically if the validation expression is incomplete
NXATOM_GAINFO_VALIDUSER_ON	returns whether or not the validation of values entered by the end user is enabled
NXATOM_GAINFO_VALIDENGINE_ON	returns whether or not the validation of values set by the engine is enabled
NXATOM_GAINFO_EHS	returns information about the Else actins (Right-Hand-Side) of a rule or method
NXATOM_GAINFO_PFELSEACTIONS	returns the value to which the forward Else actions strategy is set
NXATOM_GAINFO_PFMETHODACTIONS	returns the value to which the forward LHS/RHS actions from methods strategy is set
NXATOM_GAINFO_PFMETHODELSEACTIONS	returns the value to which the forward Else actions from methods strategy is set
NXATOM_GAINFO_VALIDFUNC	returns the validation expression string attached to the atom
NXATOM_GAINFO_VALIDEXEC	returns the validation external routine name attached to the atom
NXATOM_GAINFO_VALIDHELP	returns the validation error string attached to the atom
NXATOM_GAINFO_QUESTWIN	returns the question window name attached to the atom this may be used by the user to trigger different question windows)
NXATOM_GAINFO_METHODS	returns the list of methods attached to the atom
NXATOM_GAINFO_PROPPRIVATE	returns whether or not a slot is private
NXATOM_GAINFO_PROPPUBLIC	returns whether or not a slot is public

GetIntInfo

static Int32 NDNxAtom::GetIntInfo(NxAtomPtr *atom1*, **NxAtomGAInfoEnum** *code*,
NxAtomPtr *atom2*, **Int32** *optInt***);**

Method to get various integer datatype information about `atom1'. See the API reference manual for more extensive information on allowed codes, etc. Returns the value directly

GetLongInfo

static Long NDNxAtom::GetLongInfo(NxAtomPtr *atom1*, **NxAtomGAInfoEnum** *code*,
NxAtomPtr *atom2*, **Int32** *optInt***);**

Method to get various long datatype information about `atom1'. See the API reference manual for more extensive information on allowed codes, etc. Returns the value directly

GetDoubleInfo

static Double NDNxAtom::GetDoubleInfo(NxAtomPtr *atom1*,
NxAtomGAInfoEnum *code*, **NxAtomPtr** *atom2*, **Int32** *optInt***);**

Method to get various double datatype information about `atom1'. See the API reference manual for more extensive information on allowed codes, etc. Returns the value directly

GetStrInfo

static Str NDNxAtom::GetStrInfo(NxAtomPtr *atom1*, **NxAtomGAInfoEnum** *code*,
NxAtomPtr *atom2*, **Int32** *optInt*, **Int32** *len***);**

Method to get various string datatype information about `atom1'. See the API reference manual for more extensive information on allowed codes, etc. Returns the value directly

GetAtomInfo

static NxAtomPtr NDNxAtom::GetAtomInfo(NxAtomPtr *atom1*,
NxAtomGAInfoEnum *code*, **NxAtomPtr** *atom2*, **Int32** *optInt***);**

Method to get various atom datatype information about `atom1'. See the API reference manual for more extensive information on allowed codes, etc. Returns the value directly

Atom Type

GetType

NxAtomTypeEnum NDNxAtom::GetType(void);

Returns the atom type of `atom' (Class, Object, Rule, etc)

Miscellaneous Methods

GetName

Str NDNxAtom::GetName(void);

Returns the string name of `atom'.

GetClientData

Long NDNxAtom::GetClientData(void);

Gets the user-defined client data value (stored as a long) associated with atom'.

SetClientData

void NDNxAtom::SetClientData(Long *data*);

Sets a user-defined client data value (stored as a long) to be associated with `atom'.

Atom and Datatype Descriptors

NxAtomDescEnum

Enumerated type that defines the atom data type options.

Identifier	Description
NXATOM_DESC_UNKNOWN	Unknown
NXATOM_DESC_NOTKNOWN	Notknown
NXATOM_DESC_INT	Integer
NXATOM_DESC_FLOAT	Float
NXATOM_DESC_DOUBLE	Double
NXATOM_DESC_STR	String
NXATOM_DESC_ATOM	Atom
NXATOM_DESC_VALUE	Value
NXATOM_DESC_LONG	Long
NXATOM_DESC_DATE	Data
NXATOM_DESC_TIME	Time

Chapter

4 NxClass Class

Static Methods

GetCount

static Int32 NDNxClass::GetCount(void);

Returns the number of IRE Classes currently loaded.

GetFirst

static NxClassPtr NDNxClass::GetFirst(void);

Returns the first IRE Class of the currently loaded set.

GetLast

static NxClassPtr NDNxClass::GetLast(void);

Returns the last IRE Class of the currently loaded set.

Find

static NxClassPtr NDNxClass::Find(CStr *name*);

Returns the IRE Class specified by `name'. Returns NULL if not found.

Non-Static Methods

GetNext

NxClassPtr NDNxClass::GetNext(void);

Returns the next IRE Class of the currently loaded set. This requires a valid Class (eg: from first / next).

GetPrevious

NxClassPtr NDNxClass::GetPrevious(void);

Returns the previous IRE Class of the currently loaded set. This requires a valid Class (eg: from first / next).

GetName

Str NDNxClass::GetName(void);

Returns the string name of the specified class.

GetClientData

Long NDNxClass::GetClientData(void);

Gets a user-defined client data value associated with this class.

SetClientData

void NDNxClass::SetClientData(Long *data***);**

Sets a user-defined client data value to be associated with this class.

GetKB

NxKBPtr NDNxClass::GetKB(void);

Returns the Knowledge Base associated with this class.

SetKB

void NDNxClass::SetKB(NxKBCPtr *kb***);**

Changes the Knowledge Base that contains the definition of this class.

CreateObject

NxObjectPtr NDNxClass::CreateObject(CStr *name***);**

Creates an object named `name' with the specified parent class.

DeleteObject

Int32 NDNxClass::DeleteObject(NxObjectPtr *object***);**

Removes the object `object' from the specified parent class. Returns an integer status.

GetMethodCount

Int32 NDNxClass::GetMethodCount(void);

Returns the number of user-defined methods attached directly to this class.

GetIndexedMethod

NxMethodPtr NDNxClass::GetIndexedMethod(Int32 *index***);**

Returns the Nth method attached directly to this class.

GetSlotCount

Int32 NDNxClass::GetSlotCount(void);

Returns the number of slots attached directly to this class.

GetIndexedSlot

NxSlotPtr NDNxClass::GetIndexedSlot(Int32 *index***);**

Returns the Nth slot attached directly to this class.

GetParentClassCount

Int32 NDNxClass::GetParentClassCount(void);

Returns the number of parent classes attached directly to this class.

GetIndexedParentClass

NxClassPtr NDNxClass::GetIndexedParentClass(Int32 *index***);**

Returns the Nth parent class attached directly to this class.

GetChildClassCount

Int32 NDNxClass::GetChildClassCount(void);

Returns the number of child classes attached directly to this class.

GetIndexedChildClass

NxClassPtr NDNxClass::GetIndexedChildClass(Int32 *index***);**

Returns the Nth child class attached directly to this class.

GetChildObjectCount

Int32 NDNxClass::GetChildObjectCount(void);

Returns the number of child objects attached directly to this class.

GetIndexedChildObject

NxObjectPtr NDNxClass::GetIndexedChildObject(Int32 *index***);**

Returns the Nth child object attached directly to this class.

GetPublicMethod

NxMethodPtr NDNxClass::GetPublicMethod(CStr *name***);**

Returns the method pointer/object of the public method named `name' attached directly to this class.

GetPrivateMethod

NxMethodPtr NDNxClass::GetPrivateMethod(CStr *name***);**

Returns the method pointer/object of the private method named `name' attached directly to this class.

FindSlot

NxSlotPtr NDNxClass::FindSlot(CStr *name***);**

Returns the IRE Slot specified with a property name of `name' for this class. Returns NULL if not found.

FindSlotByProp

NxSlotPtr NDNxClass::FindSlotByProp(NxPropPtr *prop***);**

Returns the IRE Slot specified with a property of `prop' for this class. Returns NULL if not found.

Class and Object Link Control

GetLinkType

NxClassLinkEnum NDNxClass::GetLinkType(NxAtomPtr *child***);**

Returns link information: none, permanent, temporary, etc.

MakeLinkPermanent

Int32 NDNxClass::MakeLinkPermanent(NxObjectPtr *object***);**

Changes an object's temporary link(s) to permanent link(s). Only links between `object' and this class are affected. Status is returned.

NxClassLinkEnum

Enumerated type that defines class link options.

Identifier	Description
NXCLASS_LINK_NOLINK	No link
NXCLASS_LINK_TEMPLINK	Temporarily linked (created in rules, methods, or by external calls
NXCLASS_LINK_PERMLINK	Permanent link (ie: kept in the knowledge base)
NXCLASS_LINK_TEMPUNLINK	Temporarily deleted link (deleted in rules, methods, or by external calls)

Chapter

5 NxCtx Class

Enumerated Types

NxCtxEnum

Enumerated type that defines context options.

Identifier	Description
NXCTX_NFYSWITCH,	indicates a context switch is occurring
NXCTX_NFYDISPOSE	indicates that a context is being disposed

Constructor and Destructor

Constructor

NDNxCtx::NDNxCtx(void);

Destructor

NDNxCtx::~NDNxCtx(void);

Member Functions

GetCur

static NxCtxPtr NDNxCtx::GetCur(void);

Returns the current context.

SetCur

NxCtxPtr NDNxCtx::SetCur(void);

Sets the current context. The previous current context pointer is returned, and may be ignored if desired. A NULL return means there was no previous established context.

IsValid

BoolEnum NDNxCtx::IsValid(void);

A debugging aid to ensure that the specified context is valid.

IsClientIdValid

static BoolEnum NDNxCtx::IsClientIdValid(NxCtxClientId *id***);**

A debugging aid to ensure that the specified client id is valid.

AllocateClientId

static NxCtxClientId NDNxCtx::AllocateClientId(void);

Allocates and returns a client id so that multiple clients can associate multiple pieces of data with the same context block.

SetClientData

void NDNxCtx::SetClientData(NxCtxClientId *id***, ClientPtr** *ptr***);**

Allows a customer to store information for their purposes along with the context block. The client id `id' must be specified (a unique value having been obtained from AllocateClientId). This data will not be examined by any part of IRE, and is available for user customization purposes (eg: it could be a pointer to a structure or string; it could be a numeric value; etc).

GetClientData

ClientPtr NDNxCtx::GetClientData(NxCtxClientId *id***);**

Allows a customer to retrieve the information previously set with NxCtx::SetClientData(). As in that case, the information depends on the context block, as well as the client id (`id').

SetNfyProc

static void NDNxCtx::SetNfyProc(NxCtxNfyProc *proc***);**

Associates a notification procedure with the context mechanism. This is setup on a global basis, and not on a per-context basis. It is possible to have more than one notification procedure installed. The procedures will be called (in reverse order of registration) for events such as the occurrence of a context switch or a context being destroyed. The user may take advantage of this to update, cleanup,initialize, or whatever as needed.

UnsetNfyProc

static void NDNxCtx::UnsetNfyProc(NxCtxNfyProc *proc***);**

This will remove the specified procedure from the list of procedures that will be notified when a context event occurs.

Chapter

6 NxEdt Class

NxEdt and NxEdtInfo Data Structures

NxEdt Structure

NXEDTINFO_SUBREC

Identifier	Description
`Codes`	numeric codes
`Strs`	formatted strings
`Atoms`	NxAtomPtrs

NXEDT_SUBREC

Identifier	Description
`AtomType;`	Atom Type
`Id;`	Field IDs
`Text;`	Filed Text Strings
`Errors;`	Errors, if desired
`Dependencies;`	Dependencies, if desired

Error Explanation

Explanation of errors related to compiling with the nxedt API. These error codes are defined in nxppub.h Errors related to the NxEdtPtr (edPtr) passed to NxEdt::Create and NxEdt::Modify:

Identifier	Description
`NXP_ERR_NULLEDPTR`	The edPtr is NULL.
`NXP_ERR_NULLDATA`	The Text or Id array in the edPtr are NULL.
`NXP_ERR_DATANSYNC`	The lengths of the Text and Id array are not equal.
`NXP_ERR_INVALIDID`	There is an ID in the Id array which is not valid for the the AtomType of the edPtr.
`NXP_ERR_INVALIDVSTR`	There is a NULL entry in the Text array of the edPtr. All of the VStrPtr's in the Text array must be not NULL.
`NXP_ERR_MISSINGREQD`	A required piece of information for the AtomType in the edPtr is missing - For instance all atom types require an entry in the Id array of NXP_AINFO_NAME. If one did not exist in the Id array of the edPtr passed, this error would result.
`NXP_ERR_NOATOMTYPE`	The AtomType associated with the edPtr is invalid.

Constructor and Destructor

Constructor

NDNxEdt::NDNxEdt(void);

Destructor

NDNxEdt::~NDNxEdt(void);

Member Functions

Reset

void NDNxEdt::Reset(void);

Resets the fields of the edit record for reuse. Arrays will be reset to zero length; any strings will be disposed.

Fill

Int NDNxEdt::Fill(NxAtomPtr *atom***);**

Fills the edit structure with the current definition of an existing Atom. This can then be used for display or modification purposes.

Delete

Int NDNxEdt::Delete(NxAtomPtr *atom***);**

Attempts to delete the specified Atom from the Knowledge Base. If there are cross-dependency issues and no dependency structure has been provided with the EditInfo, then the Atom will be deleted regardless of dependencies, and the engine will fix things up in its standard fashion. If there are cross-dependency issues and a dependency structure has been provided, the Atom will not be deleted, and the dependency information will be returned. If there are no cross-dependency issues, the Atom will be deleted regardless of whether the dependency structure is present. Status is returned.

Modify

Int NDNxEdt::Modify(NxAtomPtr *oldAtom***, NxAtomPtrPtr** *newAtom***);**

Modifies the information of an existing Atom, depending on whether or not dependency information is present. (See NxEdt::Delete() for additional information on how the dependency information / modify mechanism works. Status is returned.

Create

Int NDNxEdt::Create(NxAtomPtrPtr *newAtom***);**

Creates a new Atom in the Knowledge Base using the specified information. ’newAtom' is the newly created Atom. Status is returned.

SetAtomType

Int NDNxEdt::SetAtomType(Int *type***);**

Sets the AtomType field in the edit record to the type specified by `type'.

SetStr

Int NDNxEdt::SetStr(Int *code***, CStr** *value***);**

Sets the value of the field specified by `code' to the string specified in `value'. The edit record will either have an entry modified or added, as necessary. Status is returned.

SetNthStr

Int NDNxEdt::SetNthStr(Int *code***, CStr** *value***, Int** *occurrence***);**

Sets the `occurrence'-th instance of the field specified by `code' to the string passed in `value'. The string is cloned, so the user is free to do whatever they want with it afterward. The index (`occurrence') is 0-based. If it previously exists, it will be replaced with the new value. If the index is higher than any previously existing, it will simply be added to the end. Status is returned.

GetStr

Int NDNxEdt::GetStr(Int *code***, CStrPtr** *value***);**

Retrieves into `value' the string corresponding to field `code' for the atom. Status is returned. The user must NOT modify these strings directly. Status is returned.

GetNthStr

Int NDNxEdt::GetNthStr(Int *code***, CStrPtr** *value***, Int** *occurrence***);**

Retrieves the string into `value' for the specified Atom which matches the `occurrence'-th instance for the field code `code'. This is used for field codes that frequently have more than one value for a particular field code (eg: multiple conditions / actions in a rule, child classes in a class, etc). Status is returned.

FindIndex

Int NDNxEdt::FindIndex(Int *code***, CStr** *value***, IntPtr** *occurrence***);**

Retrieves the index number in `occurrence' of the specified Atom, with a field code of `code' and a string value of `value'. Status is returned.

RemoveNthStr

Int NDNxEdt::RemoveNthStr(Int *code***, Int** *occurrence***);**

Removes the `occurrence'-th instance of the field specified by `code' from the edit record of this Atom. This will not affect the Atom until a Modify / Create operation is performed. Status is returned.

RemoveStr

Int NDNxEdt::RemoveStr(Int *code***);**

Removes the entry in the edit record specified with the field code `code'.

Chapter

7 NxEngine Class

Engine Error Codes

NxEngineErrEnum

Enumerated type that defines inference engine errors.

Identifiers	Description
NXENGINE_ERR_NOERR	Call was successful
NXENGINE_ERR_INVARG1	First argument of call is invalid
NXENGINE_ERR_INVARG2	Second argument of cal lis invalid
NXENGINE_ERR_INVARG3	Third argument of call is invalid
NXENGINE_ERR_INVARG4	Fourth argument of call is invalid
NXENGINE_ERR_INVARG5	Fifth argument of call is invalid
NXENGINE_ERR_INVARG6	Sixth argument of call is invalid
NXENGINE_ERR_NOTFOUND	No atom of the right type was found
NXENGINE_ERR_INVATOM	Some atoms saved in the file are invalid
NXENGINE_ERR_UNKNOWN	The value of the atom is UNKNOWN
NXENGINE_ERR_NOTKNOWN	The value of the atom is NOTKNOWN
NXENGINE_ERR_INVSTATE	Argument is in an invalid state
NXENGINE_ERR_MATHERROR	A floating point error occurred
NXENGINE_ERR_NOTAVAIL	This information is not available
NXENGINE_ERR_COMPILEPB	Compilation of the new atom did not succeed and the error was not reported correctly
NXENGINE_ERR_PROTPB	A program security error has occurred
NXENGINE_ERR_FILEOPEN	File could not be opened
NXENGINE_ERR_FILEEOF	End of file encountered unexpectedly
NXENGINE_ERR_FILEREAD	Error reading the file
NXENGINE_ERR_FILEWRITE	Error writing the file
NXENGINE_ERR_FILESEEK	Error seeking the file
NXENGINE_ERR_SYNCERROR	The parser has lost its synchronization. The contents of the file may be corrupted.
NXENGINE_ERR_FORMATERROR	The file header is invalid
NXENGINE_ERR_NOMEMORY	Memory allocation failed
NXENGINE_ERR_ABORT	Compilation was aborted by user, or the description was incomplete and no interface was provided to prompt the user
NXENGINE_ERR_SYNTAX	The text file contains a syntax error
NXENGINE_ERR_INTERNAL	Some internal consistency check failed
NXENGINE_ERR_VOLINVAL	VolunteerValidate handler returned FALSE
NXENGINE_ERR_SAVEDISABLED	Saving of KBs has been disabled
NXENGINE_ERR_VALIDATEERROR	Data validation error ... data failed validation
NXENGINE_ERR_VALIDATEMISSING	Data validation error ... data is missing
NXENGINE_ERR_VALIDATEUSER	Data validation error ... user rejected data
NXENGINE_ERR_INVARG7	Seventh argument of call is invalid.

GetError

static NxEngineErrEnum NDNxEngine::GetError(void);

In the case of an error accessing any API, this will retrieve a more detailed error code. (unless an exception is thrown, instead).

Engine Control

Start

static NxEngineCtrlRetEnum NDNxEngine::Start(void);

Start the rule engine. Returns status.

Restart

static NxEngineCtrlRetEnum NDNxEngine::Restart(void);

Perform a "restart" operation to reset the values in the rule engine. Returns status.

Continue

static NxEngineCtrlRetEnum NDNxEngine::Continue(void);

Resume a (temporarily) stopped session. Returns status.

Stop

static NxEngineCtrlRetEnum NDNxEngine::Stop(void);

Stop the rule engine. Returns status.

Init

static NxEngineCtrlRetEnum NDNxEngine::Init(void);

Initialize the rule engine library. Returns status.

Exit

static NxEngineCtrlRetEnum NDNxEngine::Exit(void);

Terminate the rule engine library. Returns status.

Control Access

The following is for advanced use. You should try to stick with the above, first, or, alternatively, check some of the other class APIs eg: NxKB).

NxEngineCtrlRetEnum

Enumerated type that defines inference engine status types.

Identifiers	Description
NXENGINE_CTRLRET_ERROR	An error occurred during execution
NXENGINE_CTRLRET_NOERROR	No error occurred during execution

Identifiers	Description
NXENGINE_CTRLRET_REMOTE	Mask indicating a remote handler was successfully requested
NXENGINE_CTRLRET_REMOTEEXECUTE	A remote execute handler was triggered
NXENGINE_CTRLRET_REMOTEPOLLING	A remote polling handler fired
NXENGINE_CTRLRET_REMOTEQUESTION	A remote question handler was triggered

Control

static NxEngineCtrlRetEnum NDNxEngine::Control(NxEngineCtrlEnum *code*);

Advanced access control of the rule engine. Normally the other, more direct methods (eg: Start, Stop, ...) should be used. Status is returned.

NxEngineCtrlEnum

Enumerated type that defines inference engine controls.

Identifiers	Description
NXENGINE_CTRL_MASK	Mask
NXENGINE_CTRL_INIT	Initialize working memory
NXENGINE_CTRL_KNOWCESS	Start inference engine
NXENGINE_CTRL_STOPSESSION	Suspends the current engine execution
NXENGINE_CTRL_CONTINUE	Restarts a session stopped with a STOPSESSION
NXENGINE_CTRL_RESTART	Restarts the sessions by resetting to UNKNOWN all the slots in the knowledge base
NXENGINE_CTRL_CLEARKB	Clears all knowledge bases from memory
NXENGINE_CTRL_EXIT	Unloads the library and cleans up memory
NXENGINE_CTRL_SETSTOP	Places a special stop context in the engine queue, which will trigger a STOPSESSION when hit
NXENGINE_CTRL_SAVESTRAT	Causes a special save strategies context to be saved in the engine queue, which will trigger a reload of the current strategies when re-encountered
NXENGINE_CTRL_ATTOP	Special control for SETSTOP and SAVESTRAT. This is the default, and makes it the next item to be processed, unless more items get queued in front of it.
NXENGINE_CTRL_ATBOTTOM	Special control for SETSTOP and SAVESTRAT. This is the default, and makes it the next item to be processed, unless more items get queued in front of it.

Engine State

GetState

static NxEngineStateEnum NDNxEngine::GetState(void);

Return the current "state" of the rule engine.

NxEngineStateEnum

Enumerated type that defines inference engine state.

Identifier	Description
NXENGINE_STATE_NOTINIT	Not initialized
NXENGINE_STATE_DONE	The state after the end of a session
NXENGINE_STATE_RUNNING	The inference engine is running (eg: when executing an Execute routine.
NXENGINE_STATE_STOPPED	The session has been interrupted (either by clicking on on Interrupt in the development environment, or calling Stop
NXENGINE_STATE_QUESTION	A question is pending. The engine is stopped, waiting for a value

Journaling

Journal

static Int32 ND NxEngine::Journal(NxEngineJrnlEnum *mode***, CStr** *filename***);**

Save the working state of the rule engine to a specified file for later reuse. This method class provides the replay capability. Returns an integer status.

NxEngineJrnlEnum

Enumerated type that defines inference engine journal status.

Identifiers	Description
NXENGINE_JRNL_RECORDSTART	Starts recording
NXENGINE_JRNL_RECORDSTOP	Stop the current recording
NXENGINE_JRNL_PLAYSTART	Starts a replay ... (see also PLAYSTEP, PLAYSKIPSHOW, PLAYNOSCAN)
NXENGINE_JRNL_PLAYSTOP	Stops the replaying of the current file
NXENGINE_JRNL_VALUESSAVE	Saves all the current working memory values in the order they were set in an NXP format file
NXENGINE_JRNL_STATESAVE	Saves the current state in a machine dependent file
NXENGINE_JRNL_STATERESTORE	Restores the state from a file. The state should have been saved previously with a STATESAVE call or from the journaling interface in the development environment
NXENGINE_JRNL_PLAYNOSCAN	Starts a replay, but disables the scanning of the file for each value (to be OR-ed in with PLAYSTART)
NXENGINE_JRNL_PLAYSKIPSHOW	Starts a replay, but indicates that Show conditions should be skipped (to be OR-ed in with PLAYSTART)
NXENGINE_JRNL_PLAYSTEP	Starts a replay, but indicates a step by step replay. to be OR-ed in with PLAYSTART)

Strategy Control

GetCurrentStrategy

static Int32 ND NxEngine::GetCurrentStrategy(NxEngineStrategyEnum *code***);**

Retrieves the current setting for the strategy option specified.

GetDefaultStrategy

static Int32 NDNxEngine::GetDefaultStrategy(NxEngineStrategyEnum *code***);**

Retrieves the current setting for the strategy option specified.

SetCurrentStrategy

static void NDNxEngine::SetCurrentStrategy(NxEngineStrategyEnum *code***,**
Int32 *value***);**

Sets the strategy option `code' to the specified value. Returns an integer status.

SetDefaultStrategy

static void NDNxEngine::SetDefaultStrategy(NxEngineStrategyEnum *code***,**
Int32 *value***);**

Sets the strategy option `code' to the specified value. Returns an integer status.

NxEngineStrategyEnum

Enumerated type that defines inference engine strategy options.

Typically, these strategies are either "on" (1) or "off" (0), except as noted in the NxEngineFwrdStratEnum section following this one.

Identifiers	Description
NXENGINE_STRATEGY_BREADTHFIRST	Returns whether the inheritance search for the atom is done in a breadth first or depth first manner
NXENGINE_STRATEGY_CACTIONSON	Returns whether of not If Change methods are enabled
NXENGINE_STRATEGY_CACTIONSUNKNOWN	Returns whether or not If Change methods will also be execute when the slot if set to UNKNOWN
NXENGINE_STRATEGY_EXHBWRD	Returns whether or not exhaustive backward chaining is enabled
NXENGINE_STRATEGY_INHCLASSDOWN	Returns whether or not class slots are inheritable downwards
NXENGINE_STRATEGY_INHCLASSUP	Returns whether or not class slots are inheritable upwards
NXENGINE_STRATEGY_INHOBJDOWN	Returns whether or not object slots are inheritable downwards
NXENGINE_STRATEGY_INHOBJUP	Returns whether or not object slots are inheritable upwards
NXENGINE_STRATEGY_INHVALDOWN	Returns whether or not the value of the atom is downward inheritable
NXENGINE_STRATEGY_INHVALUP	Returns whether or not the value of the atom is upward inheritable
NXENGINE_STRATEGY_PARENTFIRST	Returns whether the inheritance search for the atom should begin by searching the parent objects of the atom or the classes to which the atom belongs
NXENGINE_STRATEGY_PFACTIONS	Returns whether or not the assignments done in the RHS of rules or in methods are forwarded
NXENGINE_STRATEGY_PFELSEACTIONS	Returns the value to which the forward Else actions strategy is set

Identifiers	Description
NXENGINE_STRATEGY_PTGATES	Returns whether or not forward chaining through gate is enabled
NXENGINE_STRATEGY_PFMETHODACTIONS	Returns the value to which the forward LHS/RHS actions from methods strategy is set
NXENGINE_STRATEGY_PFMETHODELSEACTIONS	Returns the value to which the forward Else actions from methods strategy is set
NXENGINE_STRATEGY_PWFALSE	Returns whether or not the context propagation is enabled on FALSE hypotheses
NXENGINE_STRATEGY_PWNOTKNOWN	Returns whether or not the context propagation is enabled on NOTKNOWN hypotheses
NXENGINE_STRATEGY_PWTRUE	Returns whether or not the context propagation is enabled on TRUE hypotheses
NXENGINE_STRATEGY_SOURCESCONTINUE	Returns whether or not Order of Sources methods will be fully executed even after a value is determined
NXENGINE_STRATEGY_SOURCESON	Returns whether or not Order of Sources are enabled
NXENGINE_STRATEGY_VALIDENGINE_ACCEPT	Returns whether or not the validation of values set by the engine is enabled and the value accepted automatically if the validation expression is incomplete
NXENGINE_STRATEGY_VALIDENGINE_OFF	Returns whether or not the validation of values set by the engine is enabled
NXENGINE_STRATEGY_VALIDENGINE_ON	Returns whether or not the validation of values set by the engine is enabled
NXENGINE_STRATEGY_VALIDENGINE_REJECT	Returns whether or not the validation of values set by the engine is enabled and the value rejected automatically if the validation expression is incomplete
NXENGINE_STRATEGY_VALIDUSER_ACCEPT	Returns whether or not the validation of values entered by the end user is enabled and the value accepted automatically if the validation expression is incomplete
NXENGINE_STRATEGY_VALIDUSER_OFF	Returns whether or not the validation of values set by the engine is enabled
NXENGINE_STRATEGY_VALIDUSER_ON	Returns whether or not the validation of values entered by the end user is enabled
NXENGINE_STRATEGY_VALIDUSER_REJECT	Returns whether or not the validation of values entered by the end user is enabled and the value rejected automatically if the validation expression is incomplete

NxEngineFwrdStratEnum

Enumerated type that defines inference engine forward action effects strategy options.

Identifier	Description
NXENGINE_FWRDSTRAT_OFF	Action effects are not forwarded. The strategies this applies to are: PFELSEACTIONS, PFMETHODACTIONS, and PFMETHODELSEACTIONS.
NXENGINE_FWRDSTRAT_ON	Action effects are forwarded. The strategies this applie to are: PFELSEACTIONS, PFMETHODACTIONS, and PFMETHODELSEACTIONS.

Identifier	Description
`NXENGINE_FWRDSTRAT_GLOBAL`	Action effects are set to the global forward actions strategy. The strategies this applies to are: PFELSEACTIONS, PFMETHODACTIONS, and PFMETHODELSEACTIONS.

GetDefaultResetStrategy

static NxEngineVolStratEnum NDNxEngine::GetDefaultResetStrategy(void);

Returns the default strategy to be used when resetting a slot/hypo to UNKNOWN. If unset, the default is NXENGINE_VOLSTRAT_VOLFWRD.

SetDefaultResetStrategy

static void NDNxEngine::SetDefaultResetStrategy(NxEngineVolStratEnum *strat***);**

Sets the default strategy to be used when resetting a slot/hypo to UNKNOWN. If unset, the default is NXENGINE_VOLSTRAT_VOLFWRD.

GetDefaultVolunteerStrategy

static NxEngineVolStratEnum NDNxEngine::GetDefaultVolunteerStrategy(void);

Returns the default strategy to be used when volunteering data. If unset, the default is NXENGINE_VOLSTRAT_VOLFWRD.

SetDefaultVolunteerStrategy

static void NDNxEngine::SetDefaultVolunteerStrategy(NxEngineVolStratEnum *strat***);**

Sets the default strategy to be used when volunteering data. If unset, the default is NXENGINE_VOLSTRAT_VOLFWRD.

NxEngineVolStratEnum

Enumerated type that defines inference engine volunteer strategy options.

Identifier	Description
`NXENGINE_VOLSTRAT_QUEUE`	Queue the value with the forwarding priority, but set the value when the inference engine evaluates it.
`NXENGINE_VOLSTRAT_SET`	Force the new value in the slot immediately.
`NXENGINE_VOLSTRAT_SETQUEUE`	Same as QUEUE OR-ed with SET.
`NXENGINE_VOLSTRAT_NOCHECK`	Disable data type checking for performance needs.
`NXENGINE_VOLSTRAT_NOFWRD`	The new value will not be forwarded in the rule network. It will just be pasted in the value slot and will not influence the inference process.
`NXENGINE_VOLSTRAT_VOLFWRD`	The new value will be forwarded in the rule network as if it was volunteered manually from the interface with a global or local menu. This options is recommended when trying to propagate all the consequences of a new value. It is better to use this option at the beginning of a session.
`NXENGINE_VOLSTRAT_RHSFWRD`	The new value will be forwarded in the rule network as if it was set from inside an RHS. The engine will not examine all the possible pattern matching rules (selective forward) but will investigate the strong links
`NXENGINE_VOLSTRAT_CURFWRD`	Same as VOLFWRD except that the global strategy setting Forward-Action-Effects will be checked first. If it is off, the value will not be forwarded

Identifier	Description
NXENGINE_VOLSTRAT_QFWRD	This priority should be used when sending the answer to the current question. A continue session message would be needed anyway if the question handler had called stop session (in case one wants non-modal questions)
NXENGINE_VOLSTRAT_RESET	Used for resetting the backward chaining on a hypothesis. The value will be set back to UNKNOWN with its backward chaining

GetDefaultSuggestStrategy

static NxEngineSugPrioEnum NDNxEngine::GetDefaultSuggestStrategy(void);

Returns the default strategy to be used when suggesting a hypo. If unset, the default is NXENGINE_SUGPRIO_SUG.

SetDefaultSuggestStrategy

static void NDNxEngine::SetDefaultSuggestStrategy(NxEngineSugPrioEnum *strat***);**

Sets the default strategy to be used when suggesting a hypo. If unset, the default is NXENGINE_SUGPRIO_SUG.

NxEngineSugPrioEnum

Enumerated type that defines inference engine suggest options.

Identifier	Description
NXENGINE_SUGPRIO_UNSUG	The atom will be removed from the agenda.
NXENGINE_SUGPRIO_SUG	The atom is queued for evaluation with the same priority as if it was suggested from the interface through a popup menu or through the Suggest global menu. The current atom being investigated is evaluated and then control switches back to the atom.
NXENGINE_SUGPRIO_HYPISL	The atom is queued in the current knowledge island.
NXENGINE_SUGPRIO_DATAISL	The atom is queued in the curent knowledge island but with a priority less than HYPISL. All the hypotheses queued with HYPISL will be investigated before any of those queued with DATAISL.
NXENGINE_SUGPRIO_CNTX	The atom will compete with the contexts.

Compilation

Compile

static Int32 NDNxEngine::Compile(CStr *str***);**

Compiles a user-supplied string with the TKB parser/compiler into the current KB. Returns an integer status.

Handlers

GetHandler

static NxpIProc NDNxEngine::GetHandler(NxEngineProcEnum *code***);**

Returns any previously installed 3GL handler specified by `code'. A NULL return indicates no handler had been installed. Note, this is NOT for use with Execute routines, which should use NXENGINE_GetExecuteHandler instead.

SetHandler

static Int32 NDNxEngine::SetHandler(NxEngineProcEnum *code***, NxpIProc** *proc***);**

Allows a user to establish a 3GL handler specified by `code', that should call `proc' when required. Status is returned. Note, this is NOT for use with Execute routines, which should use NXENGINE_SetExecuteHandler instead.

GetHandler2

static NxpIProc NDNxEngine::GetHandler2(NxEngineProcEnum *code***, LongPtr** *arg***);**

Returns any previously installed 3GL handler specified by `code'. A NULL return indicates no handler had been installed. `arg' will be filled with the custom information the user registered during the call to NXENGINE_SetHandler2(). Note, this is NOT for use with Execute routines, which should use NXENGINE_GetExecuteHandler2 instead.

SetHandler2

static Int32 NDNxEngine::SetHandler2(NxEngineProcEnum *code***, NxpIProc** *proc***,
Long** *arg***);**

Allows a user to establish a 3GL handler specified by `code', that should call `proc' when required. `arg' is user-supplied custom information (eg: a value or pointer) that will not be interpreted by the rule engine. arg' will be passed to the registered callback procedure. Status is returned. Note, this is NOT for use with Execute routines, which should use NxEngine::SetExecuteHandler instead.

NxEngineProcEnum

Enumerated types that defines inference engine callback procedures.

Identifier	Description
`NXENGINE_PROC_EXECUTE`	User defined Execute routine.
`NXENGINE_PROC_POLLING`	Polling procedure called at each inference engine cycle.
`NXENGINE_PROC_QUESTION`	Question asked by the engine.
`NXENGINE_PROC_ALERT`	Alert information brought on the screen.
`NXENGINE_PROC_APROPOS`	Show action.
`NXENGINE_PROC_NOTIFY`	Notifies the interface when something changes in the working memory.
`NXENGINE_PROC_GETSTATUS`	Checks the availability of an interface.
`NXENGINE_PROC_SETDATA`	Sends data to the interface.
`NXENGINE_PROC_GETDATA`	Gets data from the interface.

Identifier	Description
NXENGINE_PROC_FORMINPUT	Get control before a form is open.
NXENGINE_PROC_CANCEL	Interrupt handler.
NXENGINE_PROC_ENCRYPT	Knowledge base encryption handler.
NXENGINE_PROC_DECRYPT	Knowledge base decryption handler.
NXENGINE_PROC_PASSWORD	Prompts for an encrypted knowedge base password.
NXENGINE_PROC_MEMEXIT	Exits when no more memory is available.
NXENGINE_PROC_ENDOFSESSION	Called upon the end of a session.
NXENGINE_PROC_VOLVALIDATE	Supplies your data valiation function.
NXENGINE_PROC_QUIT	Called the the Rule Element is going to exit.
NXENGINE_PROC_VALIDATE	User-supplied data validation procedure.
NXENGINE_PROC_TRANSCRIPT	Called with the string being put in the transcript.

GetExecuteHandler

static NxpIProc ND NxEngine::GetExecuteHandler(CStr *name***);**

Returns any previously installed 3GL execute handler specified by `name'. A NULL return indicates no handler had been installed.

SetExecuteHandler

static Int32 ND NxEngine::SetExecuteHandler(CStr *name***, NxpIProc** *proc***);**

Allows a user to establish a 3GL execute handler specified by `name', that should call `proc' when required. Status is returned.

GetExecuteHandler2

static NxpIProc ND NxEngine::GetExecuteHandler2(CStr *name***, LongPtr** *arg***);**

Returns any previously installed 3GL execute handler specified by `name'. `arg' will be filled with the custom information the user registered during the call to NXENGINE_SetExecuteHandler2(). A NULL return indicates no handler had been installed.

SetExecuteHandler2

static Int32 ND NxEngine::SetExecuteHandler2(CStr *name***, NxpIProc** *proc***, Long** *arg***);**

Allows a user to establish a 3GL execute handler specified by `name', that should call `proc' when required. `arg' is user-supplied custom information eg: a value or pointer) that will not be interpreted by the rule engine. `arg' will be passed to the registered callback procedure. Status is returned.

Alert Types passed in Alert Handler

NxEngineAlrtEnum

Enumerated type that defines inference engine alert window options.

Identifier	Description
NXENGINE_ALERT_OK	OK alert window.
NXENGINE_ALERT_OKCANCEL	OK / Cancel alert window.
NXENGINE_ALERT_YESNOCANCEL	Yes / No / Cancel alert window.

Alert Return Codes from an Alert Handler

NxEngineAlrtRetEnum

Enumerated type that defines inference engine alert window button selection.

Identifier	Description
NXENGINE_ALERTRET_OK	OK button selected.
NXENGINE_ALERTRET_CANCEL	Cancel button selected.
NXENGINE_ALERTRET_YES	Yes button selected.
NXENGINE_ALERTRET_NO	No button selected.

Engine Notify Handler Codes

NxEngineNfyEnum

Enumerated type that defines inference engine notification codes.

Identifier	Description
NXENGINE_NFY_DELETE	If the atom is NULL, you are notified that the knowledge base was cleared. Otherwise you are notified that the atom will be deleted and should no longer be referenced it becomes invalid). The atom can be an object, class, property, slot, or rule.
NXENGINE_NFY_CREATE	If the atom is NULL, you are notified that a knowledge base was loaded. Otherwise you are notified that the atom has just been created. The atom can be an object, class, property, slot or rule.
NXENGINE_NFY_MODIFY	This notification informs you that a structural change occurred in the object base. The atom is that of the parent atom involved in the change. You will be notified every time a link is created or deleted in the object base. For example, when an object is attached to a class or removed from a class, the notification handler is called with the class. It is also called when subobjects are created or deleted.
NXENGINE_NFY_UPDATE	This notification informs you that the value of a slot has changed. The atom is the slot whose value has changed. This notification differs from an IfChange method because it is done even if the session is not running (when you volunteer values interactively) or if the value is reset to UNKNOWN.
NXENGINE_NFY_RESTART	The session has been restarted.
NXENGINE_NFY_REDRAW	This notification is sent every time the system is interrupted or pauses for a question.

GetStatus Codes

NxEngineGSEnum

Enumerated type that defines inference engine GetStatus handler code.

Identifier	Description
NXENGINE_GS_ENABLED	Used with the GetStatus handler to indicate the window is enabled.

Window Codes with Notify Handler

NxEngineWinEnum

Enumerated type that defines inference engine window options.

Identifier	Description
NXENGINE_WIN_TRAN	Transcript window.
NXENGINE_WIN_RULE	Current Rule window.
NXENGINE_WIN_CONC	Conclusions window.
NXENGINE_WIN_HYPO	Current Hypothesis window.

Chapter

8 NxKB Class

Enumerated Types

NxKBUnloadEnum

Enumerated type that defines knowledge base options.

Item	Description
NXKB_UNLOAD_ENABLE	The knowledge base is enabled.
NXKB_UNLOAD_DISABLEWEAK	The knowledge base is disabled, but the agenda of the inference is not modified.
NXKB_UNLOAD_DISABLESTRONG	The knowledge base is disabled and its rules, hypotheses, methods are removed from the agenda of the inference engine.
NXKB_UNLOAD_DELETE	The knowledge base is disabled, its rules, hypotheses, methods are removed from the agenda of the inference engine and the data structures associated with rules and methods are released in memory so that the memory space that they occupied can be reused for other rules, objects, or for other applications. The objects and classes belonging to the knowledge base remain in memory.
NXKB_UNLOAD_WIPEOUT	Same as NXKB_UNLOAD_DELETE, but the data structures associated with the objects and classes are also released.

NxKBSaveSetEnum

Enumerated type that defines knowledge base save options.

Item	Description
NXKB_SAVE_TEXT	Knowledge base saved in text form.
NXKB_SAVE_COMPILED	Knowledge base saved in compiled form.
NXKB_SAVE_COMMENTS	Knowledge base saved with comments.

Static Methods

GetCount

static Int32 NDNxKB::GetCount(void);

Returns the number of Knowledge Bases currently loaded.

GetFirst

static NxKBPtr NDNxKB::GetFirst(void);

Returns the first Knowledge Base of the currently loaded set.

GetLast

static NxKBPtr NDNxKB::GetLast(void);

Returns the last Knowledge Base of the currently loaded set.

Load

static NxKBPtr NDNxKB::Load(CStr *name***);**

Loads the Knowledge Base from the file specified by `name'. Returns the KB object.

ClearAll

static Int32 NDNxKB::ClearAll(void);

Clears all the Knowledge Bases from memory. Returns an integer status.

GetCurrent

static NxKBPtr NDNxKB::GetCurrent(void);

Retrieves the currently active / set Knowledge Base, or NULL if an error.

SetCurrent

static void NDNxKB::SetCurrent(NxKBCPtr *kb***);**

Sets the currently active Knowledge Base to `kb'.

Create

static NxKBPtr NDNxKB::Create(CStr *name***);**

Creates a knowledge base specified by `name'.

Find

static NxKBPtr NDNxKB::Find(CStr *name***);**

Given `name', find the corresponding KB object.

Merge

static Int32 NDNxKB::Merge(NxKBPtr *kb1***, NxKBPtr** *kb2***);**

Merges all components from knowledge base `kb2' into knowledge base `kb1'. Returns an integer status.

Non-Static Methods

GetNext

NxKBPtr NDNxKB::GetNext(void);

Returns the next Knowledge Base of the currently loaded set, based from `kb'. This requires a valid KB (eg: from first / next).

GetPrevious

NxKBPtr NDNxKB::GetPrevious(void);

Returns the previous Knowledge Base of the currently loaded set, based from `kb', This requires a valid KB (eg: from first / next).

GetName

Str NDNxKB::GetName(void);

For a given Knowledge Base `kb', returns the string name associated with it.

GetComments

CStr NDNxKB::GetComments(void);

For a given Knowledge Base `kb', returns any user comments associated with it.

MakeLinksPermanent

Int32 NDNxKB::MakeLinksPermanent(void);

This changes all temporary links of objects / classes of the `kb' to permanent. Returns an integer status.

KB Unload

Unload

Int32 NDNxKB::Unload(NxKBUnloadEnum *flags***);**

Unloads the Knowledge Base from memory, under control specified by `flags'. Returns an integer status.

KB Save

Save

Int32 NDNxKB::Save(CStr *filename***, NxKBSaveSet** *flags***);**

Saves the specified Knowledge Base to a file, with `flags' to control saving of comments, compiled, text forms, etc. Returns an integer status.

Chapter

9 NxMethod Class

Static Methods

GetCount

static Int32 NDNxMethod::GetCount(void);

Returns the number of IRE Methods currently loaded.

GetFirst

static NxMethodPtr NDNxMethod::GetFirst(void);

Returns the first IRE Method of the currently loaded set.

GetLast

static NxMethodPtr NDNxMethod::GetLast(void);

Returns the last IRE Method of the currently loaded set.

GetCurrent

static NxMethodPtr NDNxMethod::GetCurrent(void);

Retrieves the current method, which may be NULL if there is none.

Non-Static Methods

GetNext

NxMethodPtr NDNxMethod::GetNext(void);

Returns the next IRE Method of the currently loaded set. This requires a valid Method (eg: from first / next).

GetPrevious

NxMethodPtr NDNxMethod::GetPrevious(void);

Returns the previous IRE Method of the currently loaded set. This requires a valid Method (eg: from first / next).

GetName

Str NDNxMethod::GetName(void);

Returns the string name of the specified method.

GetComments

Str NDNxMethod::GetComments(void);

Returns the user comments string associated with the method.

GetKB

NxKBPtr NDNxMethod::GetKB(void);

Returns the Knowledge Base associated with this method.

SetKB

void NDNxMethod::SetKB(NxKBCPtr *kb***);**

Changes the Knowledge Base that contains the definition of this method.

GetIfConditionCount

Int32 NDNxMethod::GetIfConditionCount(void);

Returns the number of Conditions found in this method (from the If or LHS part).

GetIndexedIfCondition

Str NDNxMethod::GetIndexedIfCondition(Int32 *index***);**

Returns the string representation of the Nth LHS of this method.

GetThenActionCount

Int32 NDNxMethod::GetThenActionCount(void);

Returns the number of Then-Actions found in this method (from the Then or RHS part).

GetIndexedThenAction

Str NDNxMethod::GetIndexedThenAction(Int32 *index***);**

Returns the string representation of the Nth RHS of this method.

GetElseActionCount

Int32 NDNxMethod::GetElseActionCount(void);

Returns the number of Else-Actions found in this method (from the Else or EHS part).

GetIndexedElseAction

Str NDNxMethod::GetIndexedElseAction(Int32 *index***);**

Returns the string representation of the Nth EHS of this method.

IsPrivate

Int32 NDNxMethod::IsPrivate(void);

Returns a boolean value indicating if the method is private.

Chapter

10 NxObject Class

Enumerated Types

NxObjectLinkEnum

Enumerated type that defines object link options.

Item	Description
NXOBJECT_LINK_NOLINK	No link.
NXOBJECT_LINK_TEMPLINK	Temporarily linked (created in rules, methods, or by external calls.
NXOBJECT_LINK_PERMLINK	Permanent link (ie: kept in the knowledge base).
NXOBJECT_LINK_TEMPUNLINK	Temporarily deleted link (deleted in rules, methods, or by external calls).

Static Methods

GetCount

static Int32 NDNxObject::GetCount(void);

Returns the number of IRE Objects currently loaded.

GetFirst

static NxObjectPtr NDNxObject::GetFirst(void);

Returns the first IRE Object of the currently loaded set.

GetLast

static NxObjectPtr NDNxObject::GetLast(void);

Returns the last IRE Object of the currently loaded set.

Create

static NxObjectPtr NDNxObject::Create(CStr *name***);**

Creates an object named `name' with no specific parent. Returns the object, or NULL if an error.

Find

static NxObjectPtr NDNxObject::Find(CStr *name***);**

Returns the IRE Object specified by `name'. Returns NULL if not found.

Non-Static Methods

GetNext

NxObjectPtr NDNxObject::GetNext(void);

Returns the next IRE Object of the currently loaded set. This requires a valid Object (eg: from first / next).

GetPrevious

NxObjectPtr NDNxObject::GetPrevious(void);

Returns the previous IRE Object of the currently loaded set. This requires a valid Object (eg: from first / next).

GetName

Str NDNxObject::GetName(void);

Returns the string name of the specified object.

GetClientData

Long NDNxObject::GetClientData(void);

Gets a user-defined client data value associated with this object.

SetClientData

SetClientDatavoid NDNxObject::SetClientData(Long *data*);

Sets a user-defined client data value to be associated with this object.

GetKB

NxKBPtr NDNxObject::GetKB(void);

Returns the Knowledge Base associated with this object.

SetKB

void NDNxObject::SetKB(NxKBCPtr *kb*);

Changes the Knowledge Base that contains the definition of this object. NxObjectPtr.

CreateObject

NDNxObject::CreateObject(CStr *name*);

Creates an object named `name' with the specified parent object. Returns the object, or NULL if an error.

Delete

Int32 NDNxObject::Delete(void);

Deletes the object named `object'. Returns an integer status.

DeleteObject

Int32 NDNxObject::DeleteObject(NxObjectPtr *childObj***);**

Removes an object named `childObj' from the specified parent object. Returns an integer status.

GetMethodCount

Int32 NDNxObject::GetMethodCount(void);

Returns the number of user-defined methods attached directly to this object.

GetIndexedMethod

NxMethodPtr NDNxObject::GetIndexedMethod(Int32 *index***);**

Returns the Nth method attached directly to this object.

GetSlotCount

Int32 NDNxObject::GetSlotCount(void);

Returns the number of slots attached directly to this object.

GetIndexedSlot

NxSlotPtr NDNxObject::GetIndexedSlot(Int32 *index***);**

Returns the Nth slot attached directly to this object.

GetParentClassCount

Int32 NDNxObject::GetParentClassCount(void);

Returns the number of parent classes attached directly to this object.

GetIndexedParentClass

NxClassPtr NDNxObject::GetIndexedParentClass(Int32 *index***);**

Returns the Nth parent class attached directly to this object.

GetParentObjectCount

Int32 NDNxObject::GetParentObjectCount(void);

Returns the number of parent objects attached directly to this object.

GetIndexedParentObject

NxObjectPtr NDNxObject::GetIndexedParentObject(Int32 *index***);**

Returns the Nth child object attached directly to this object.

GetChildObjectCount

Int32 NDNxObject::GetChildObjectCount(void);

Returns the number of child objects attached directly to this object.

GetIndexedChildObject

NxObjectPtr NDNxObject::GetIndexedChildObject(Int32 *index***);**

Returns the Nth child object attached directly to this object.

GetPublicMethod

NxMethodPtr NDNxObject::GetPublicMethod(CStr *name***);**

Returns the method pointer/object of the public method named `name' attached directly to this object.

GetPrivateMethod

NxMethodPtr NDNxObject::GetPrivateMethod(CStr *name***);**

Returns the method pointer/object of the private method named `name' attached directly to this object.

FindSlot

NxSlotPtr NDNxObject::FindSlot(CStr *name***);**

Returns the IRE Slot specified with a property name of `name' and a parent object `parent'. Returns NULL if not found.

FindSlotByProp

NxSlotPtr NDNxObject::FindSlotByProp(NxPropPtr *prop***);**

Returns the IRE Slot for this object, given a property of `prop'. Returns NULL if not found.

Class/Object Link Control

GetLinkType

NxObjectLinkEnum NDNxObject::GetLinkType(NxObjectPtr *child***);**

Returns link information: none, permanent, temporary, etc.

MakeLinkPermanent

Int32 NDNxObject::MakeLinkPermanent(NxAtomPtr *atom***);**

Changes an object's temporary link(s) to permanent link(s). If `atom' is specified, only links between this object and `atom' (a class or object) are affected. If `atom' is NULL, all links from this object are affected. Status is returned.

Chapter

11 NxProperty Class

Enumerated Types

NxPropDataTypeEnum

Enumerated type that defines property data type varieties.

Item	Description
NXPROP_DATATYPE_BOOL	Boolean.
NXPROP_DATATYPE_DOUBLE	Double (floating point).
NXPROP_DATATYPE_STR	String.
NXPROP_DATATYPE_SPECIAL	Special (returned only if the slot is the Value property).
NXPROP_DATATYPE_DATE	Date.
NXPROP_DATATYPE_LONG	Long integer.
NXPROP_DATATYPE_TIME	Time.

Static Methods

GetCount

static Int32 NDNxProp::GetCount(void);

Returns the number of IRE Properties currently loaded.

GetFirst

static NxPropPtr NDNxProp::GetFirst(void);

Returns the first IRE Property of the currently loaded set.

GetLast

static NxPropPtr NDNxProp::GetLast(void);

Returns the last IRE Property of the currently loaded set.

Find

static NxPropPtr NDNxProp::Find(CStr *name***);**

Returns the IRE Property specified by `name'. Returns NULL if not found.

Non-Static Methods

GetNext

NxPropPtr NDNxProp::GetNext(void);

Returns the next IRE Property of the currently loaded set. This requires a valid Property (eg: from first / next).

GetPrevious

NxPropPtr NDNxProp::GetPrevious(void);

Returns the previous IRE Property of the currently loaded set. This requires a valid Property (eg: from first / next).

GetKB

NxKBPtr NDNxProp::GetKB(void);

Returns the Knowledge Base associated with this property.

SetKB

void NDNxProp::SetKB(NxKBCPtr *kb***);**

Changes the Knowledge Base that contains the definition of this property.

GetFormat

Str NDNxProp::GetFormat(void);

Returns the format string used with the property.

SetFormat

void NDNxProp::SetFormat(CStr *format***);**

Sets the format string used with the property.

GetMethodCount

Int32 NDNxProp::GetMethodCount(void);

Returns the number of user-defined method attached directly to this property.

GetIndexedMethod

NxMethodPtr NDNxProp::GetIndexedMethod(Int32 *index***);**

Returns the Nth method attached directly to this property.

GetName

Str NDNxProp::GetName(void);

Returns the string name of the specified property.

GetPublicMethod

NxMethodPtr NDNxProp::GetPublicMethod(CStr *name***);**

Returns the method pointer / object of the public method named `name' attached directly this property.

GetPrivateMethod

NxMethodPtr NDNxProp::GetPrivateMethod(CStr *name***);**

Returns the method pointer / object of the private method named `name' attached directly this property.

Property Datatype Codes

GetDataType

NxPropDataTypeEnum NDNxProp::GetDataType(void);

Returns the datatype of the property.

Chapter

12 NxRule Class

Enumerated Types

NxRuleSugPrioEnum

Enumerated type that defines suggest priority options.

Item	Description
NXRULE_SUGPRIO_UNSUG	The atom will be removed from the agenda.
NXRULE_SUGPRIO_SUG	The atom is queued for evaluation with the same priority as if it was suggested from the interface through a popup menu or through the Suggest global menu. The current atom being investigated is evaluated and then control switches back to the atom.
NXRULE_SUGPRIO_HYPISL	The atom is queued in the current knowledge island.
NXRULE_SUGPRIO_DATAISL	The atom is queued in the curent knowledge island but with a priority less than HYPISL. All the hypotheses queued with HYPISL will be investigated before any of those queued with DATAISL.
NXRULE_SUGPRIO_CNTX	The atom will compete with the contexts.

Static Methods

GetCount

static Int32 ND NxRule::GetCount(void);

Returns the number of IRE Rules currently loaded.

GetFirst

static NxRulePtr ND NxRule::GetFirst(void);

Returns the first IRE Rule of the currently loaded set.

GetLast

static NxRulePtr ND NxRule::GetLast(void);

Returns the last IRE Rule of the currently loaded set.

Find

static NxRulePtr ND NxRule::Find(CStr *name***);**

Returns the IRE Rule specified by `name'. Returns NULL if not found.

GetCurrent

static NxRulePtr NDNxRule::GetCurrent(void);

Retrieves the current rule, which may be NULL if there is none.

Non-Static Methods

GetNext

NxRulePtr NDNxRule::GetNext(void);

Returns the next IRE Rule of the currently loaded set. This requires a valid Rule (eg: from first / next).

GetPrevious

NxRulePtr NDNxRule::GetPrevious(void);

Returns the previous IRE Rule of the currently loaded set. This requires a valid Rule (eg: from first / next).

GetComments

Str NDNxRule::GetComments(void);

Returns the user comments string associated with the rule.

GetKB

NxKBPtr NDNxRule::GetKB(void);

Returns the Knowledge Base associated with this rule.

SetKB

void NDNxRule::SetKB(NxKBCPtr *kb***);**

Changes the Knowledge Base that contains the definition of this rule.

GetName

Str NDNxRule::GetName(void);

Returns the string name of this rule.

IsUnknown

Int32 NDNxRule::IsUnknown(void);

Returns information on whether the rule is UNKNOWN.

IsNotknown

Int32 NDNxRule::IsNotknown(void);

Returns information on whether the rule is NOTKNOWN.

IsKnown

Int32 NDNxRule::IsKnown(void);

Returns information on whether the rule is KNOWN.

GetWhy

Str NDNxRule::GetWhy(void);

Returns the Why information associated with this rule.

GetInferencePriority

Int32 NDNxRule::GetInferencePriority(void);

Returns the inference priority number associated with this rule.

GetInferenceSlot

NxSlotPtr NDNxRule::GetInferenceSlot(void);

Returns the inference priority slot associated with this rule.

GetHypo

NxSlotPtr NDNxRule::GetHypo(void);

Return the hypothesis of this rule.

GetIfConditionCount

Int32 NDNxRule::GetIfConditionCount(void);

Returns the number of Conditions found in this rule (from the If or LHS part).

GetIndexedIfCondition

Str NDNxRule::GetIndexedIfCondition(Int32 *index***);**

Returns the string representation of the Nth LHS of this rule.

GetThenActionCount

Int32 NDNxRule::GetThenActionCount(void);

Returns the number of Then-Actions found in this rule (from the Then or RHS part).

GetIndexedThenAction

Str NDNxRule::GetIndexedThenAction(Int32 *index***);**

Returns the string representation of the Nth RHS of this rule.

GetElseActionCount

Int32 NDNxRule::GetElseActionCount(void);

Returns the number of Else-Actions found in this rule (from the Else or EHS part).

GetIndexedElseAction

Str NDNxRule::GetIndexedElseAction(Int32 *index***);**

Returns the string representation of the Nth EHS of this rule.

GetValue

Int32 NDNxRule::GetValue(void);

Retrieves the boolean value of this rule (true / false / notknown / unknown).

Rule Suggest Operations

SuggestHypo

Int32 NDNxRule::SuggestHypo(void);

Suggest the hypothesis of this rule to be evaluated by the rule engine. Uses the DefaultSuggestStrategy. Returns an integer status.

UnsuggestHypo

Int32 NDNxRule::UnsuggestHypo(void);

Removes the hypothesis of this rule from the agenda of the rule engine. Returns an integer status.

SuggestHypo2

Int32 NDNxRule::SuggestHypo2(NxRuleSugPrioEnum *strategy***);**

Suggest the hypothesis of this rule to be evaluated by the rule engine using the Suggest strategy provided. Returns an integer status.

IsHypoSuggested

Int32 NDNxRule::IsHypoSuggested(void);

Return a boolean indicating whether the hypothesis of this rule is to be evaluated by the rule engine.

Chapter

13 NxSlot Class

Enumerated Types

NxSlotBoolEnum

Enumerated type that defines slot state options.

Item	Description
NXSLOT_BOOL_UNKNOWN	Slot is UNKNOWN.
NXSLOT_BOOL_NOTKNOWN	Slot is NOTKNOWN.
NXSLOT_BOOL_FALSE	Slot is FALSE.
NXSLOT_BOOL_TRUE	Slot is TRUE.

NxSlotSugPrioEnum

Enumerated type that defines slot suggest priority options.

Item	Description
NXSLOT_SUGPRIO_UNSUG	The atom will be removed from the agenda.
NXSLOT_SUGPRIO_SUG	The atom is queued for evaluation with the same priority as if it was suggested from the interface through a popup menu or through the Suggest global menu. The current atom being investigated is evaluated and then control switches back to the atom.
NXSLOT_SUGPRIO_HYPISL	The atom is queued in the current knowledge island.
NXSLOT_SUGPRIO_DATAISL	The atom is queued in the curent knowledge island but with a priority less than HYPISL. All the hypotheses queued with HYPISL will be investigated before any of those queued with DATAISL.
NXSLOT_SUGPRIO_CNTX	The atom will compete with the contexts.

NxSlotDataTypeEnum

Enumerated type that defines slot data types.

Item	Description
NXSLOT_DATATYPE_BOOL	Boolean.
NXSLOT_DATATYPE_DOUBLE	Double (floating point).
NXSLOT_DATATYPE_STR	String.
NXSLOT_DATATYPE_SPECIAL	Special (returned only if the slot is the Value property).
NXSLOT_DATATYPE_DATE	Date.
NXSLOT_DATATYPE_LONG	Long integer.
NXSLOT_DATATYPE_TIME	Time.

NxSlotStratEnum

Enumerated type that defines slot inheritance strategy options.

Item	Description
NXSLOT_STRAT_INHDOWN	Returns whether or not the slot is downward inheritable.
NXSLOT_STRAT_INHUP	Returns whether or not the slot is upward inheritable.
NXSLOT_STRAT_INHVALDOWN	Returns whether or not the value of the atom is downward inheritable.
NXSLOT_STRAT_INHVALUP	Returns whether or not the value of the atom is upward inheritable.

NxSlotStratEnum

Enumerated type that defines slot inheritance searchoptions.

Item	Description
NXSLOT_STRAT_BREADTHFIRST	Returns whether the inheritance search for the atom is done in a breadth first or depth first manner.
NXSLOT_STRAT_PARENTFIRST	Returns whether the inheritance search for the atom should begin by searching the parent objects of the atom or the classes to which the atom belongs.

NxSlotDefStratEnum

Enumerated type that defines slot default inheritability options.

Item	Description
NXSLOT_DEFSTRAT_INHDEFAULT	Returns whether or not the slot inheritability of the atom follows the default (global strategy).
NXSLOT_DEFSTRAT_INHVALDEFAULT	Returns whether or not the inheritability of the value of the atom follows the default (global strategy).

NxSlotDefStratEnum

Enumerated type that defines slot default inheritance strategy options.

Item	Description
NXSLOT_DEFSTRAT_DEFAULTFIRST	Returns whether or not the inheritance strategy for the atom follows the default (global strategy).

NxSlotVolStratEnum

Enumerated type that defines slot volunteer options.

Item	Description
NXSLOT_VOLSTRAT_QUEUE	Queue the value with the forwarding priority, but set the value when the inference engine evaluates it.
NXSLOT_VOLSTRAT_SET	Force the new value in the slot immediately.
NXSLOT_VOLSTRAT_SETQUEUE	Same as QUEUE or'ed with SET.
NXSLOT_VOLSTRAT_NOCHECK	Disable data type checking for performance needs.
NXSLOT_VOLSTRAT_NOFWRD	The new value will not be forwarded in the rule network. It will just be pasted in the value slot and will not influence the inference process.

Item	Description
NXSLOT_VOLSTRAT_VOLFWRD	The new value will be forwarded in the rule network as if it was volunteered manually from the interface with a global or local menu. This options is recommended when trying to propagate all the consequences of a new value. It is better to use this option at the beginning of a session.
NXSLOT_VOLSTRAT_RHSFWRD	The new value will be forwarded in the rule network as if it was set from inside an RHS. The engine will not examine all the possible pattern matching rules selective forward) but will investigate the strong links.
NXSLOT_VOLSTRAT_CURFWRD	Same as VOLFWRD except that the global strategy setting. Forward-Action-Effects will be checked first. If it is off, the value will not be forwarded.
NXSLOT_VOLSTRAT_QFWRD	This priority should be used when sending the answer to the current question. A continue session message would be needed anyway if the question handler had called stop session (in case one wants non-modal questions).
NXSLOT_VOLSTRAT_RESET	Used for resetting the backward chaining on a hypothesis. The value will be set back to UNKNOWN with its backward chaining.

Static Methods

GetCountData

static Int32 NDNxSlot::GetCountData(void);

Returns the number of IRE Data Slots currently loaded.

GetFirstData

static NxSlotPtr NDNxSlot::GetFirstData(void);

Returns the first IRE Data Slot of the currently loaded set.

GetLastData

static NxSlotPtr NDNxSlot::GetLastData(void);

Returns the last IRE Data Slot of the currently loaded set.

GetCountHypo

static Int32 NDNxSlot::GetCountHypo(void);

Returns the number of IRE Hypothesis Slots currently loaded.

GetFirstHypo

static NxSlotPtr NDNxSlot::GetFirstHypo(void);

Returns the first IRE Hypothesis Slot of the currently loaded set.

GetLastHypo

static NxSlotPtr NDNxSlot::GetLastHypo(void);

Returns the last IRE Hypothesis Slot of the currently loaded set.

Find

static NxSlotPtr NDNxSlot::Find(CStr *name*);

Returns the IRE Slot specified by `name'. Returns NULL if not found.

GetCurrent

static NxSlotPtr NDNxSlot::GetCurrent(void);

Retrieves the current slot, which may be NULL if there is none.

GetSuggestListCount

static Int32 NDNxSlot::GetSuggestListCount(void);

Returns the number of slots in the suggest list.

GetIndexedSuggestList

static NxSlotPtr NDNxSlot::GetIndexedSuggestList(Int32 *index*);

Returns the Nth slot in the suggest list.

GetVolunteerListCount

static Int32 NDNxSlot::GetVolunteerListCount(void);

Returns the number of slots in the volunteer list.

GetIndexedVolunteerList

static NxSlotPtr NDNxSlot::GetIndexedVolunteerList(Int32 *index*);

Returns the Nth slot in the volunteer list.

Non-Static Methods

GetNextData

NxSlotPtr NDNxSlot::GetNextData(void);

Returns the next IRE Data Slot of the currently loaded set. This requires a valid data slot (eg: from first / next).

GetPreviousData

NxSlotPtr NDNxSlot::GetPreviousData(void);

Returns the previous IRE Data Slot of the currently loaded set. This requires a valid data slot (eg: from first / next).

GetNextHypo

NxSlotPtr NDNxSlot::GetNextHypo(void);

Returns the next IRE Hypothesis Slot of the currently loaded set. This requires a valid hypo slot (eg: from first / next).

GetPreviousHypo

NxSlotPtr NDNxSlot::GetPreviousHypo(void);

Returns the previous IRE Hypo Slot of the currently loaded set. This requires a valid Hypo slot (eg: from first / next).

IsHypo

Int32 NDNxSlot::IsHypo(void);

Returns a boolean indicating whether the slot is used as the hypothesis of a rule.

GetName

Str NDNxSlot::GetName(void);

Returns the string name of this slot.

GetProperty

NxPropPtr NDNxSlot::GetProperty(void);

Returns the property referenced by this slot.

GetParent

NxAtomPtr NDNxSlot::GetParent(void);

Returns the parent (object or class) referenced by this slot.

GetClientData

Long NDNxSlot::GetClientData(void);

Gets a user-defined client data value associated with this slot.

SetClientData

void NDNxSlot::SetClientData(Long ***data*****);**

Sets a user-defined client data value to be associated with this slot.

GetKB

NxKBPtr NDNxSlot::GetKB(void);

Returns the Knowledge Base associated with this slot.

SetKB

void NDNxSlot::SetKB(NxKBCPtr ***kb*****);**

Changes the Knowledge Base that contains the definition of this slot.

GetMethodCount

Int32 NDNxSlot::GetMethodCount(void);

Returns the number of user-defined methods attached directly to this slot.

GetIndexedMethod

NxMethodPtr NDNxSlot::GetIndexedMethod(Int32 ***index*****);**

Returns the Nth method attached directly to this slot.

GetPublicMethod

NxMethodPtr NDNxSlot::GetPublicMethod(CStr *name***);**

Returns the method pointer / object of the public method named `name' attached directly to this slot.

GetPrivateMethod

NxMethodPtr NDNxSlot::GetPrivateMethod(CStr *name***);**

Returns the method pointer / object of the private method named `name' attached directly to this slot.

GetChoiceCount

Int32 NDNxSlot::GetChoiceCount(void);

Returns the number of choices which the rules engine will present when querying the user for the value of the slot. The choices are based on possible values found within the loaded Knowledge Bases for slots which are of type string.

GetIndexedChoice

Str NDNxSlot::GetIndexedChoice(Int32 *index***);**

Returns the Nth choice which the rules engine will present when querying the user for the value of the slot. The choices are based on possible values found within the loaded Knowledge Bases for slots which are of type string.

GetContextCount

Int32 NDNxSlot::GetContextCount(void);

Returns the number of hypotheses that are in the context of this hypothesis.

GetIndexedContext

NxSlotPtr NDNxSlot::GetIndexedContext(Int32 *index***);**

Returns the Nth context hypothesis.

Slot Values

GetStringValue

Str NDNxSlot::GetStringValue(void);

Retrieves the value of this slot as a formatted string (using any format specified with this slot.

GetValue

VarPtr NDNxSlot::GetValue(void);

Retrieves the value of this slot, ignoring formats, in a variant. Boolean Data requested as a numeric value will have one of the values specified in NxSlotBoolEnum.

SetValue

void NDNxSlot::SetValue(VarCPtr *value***);**

Volunteers the specified value into this slot, using the Default Volunteer Strategy

IsUnknown

Int32 NDNxSlot::IsUnknown(void);

Returns information on whether the slot is UNKNOWN.

IsNotknown

Int32 NDNxSlot::IsNotknown(void);

Returns information on whether the slot is NOTKNOWN.

IsKnown

Int32 NDNxSlot::IsKnown(void);

Returns information on whether the slot is KNOWN.

Slot Suggest Priority Codes

Suggest

Int32 NDNxSlot::Suggest(void);

Enters a slot to be evaluated by the rule engine when it starts / resumes. Uses the DefaultSuggestStrategy. Returns an integer status.

Suggest2

Int32 NDNxSlot::Suggest2(NxSlotSugPrioEnum *strategy***);**

Enters a slot to be evaluated by the rule engine when it starts / resumes. Uses the Suggest strategy provided. Returns an integer status.

IsSuggested

Int32 NDNxSlot::IsSuggested(void);

Returns a boolean indicating whether the slot has been suggested to be evaluated by the rule engine.

Unsuggest

Int32 NDNxSlot::Unsuggest(void);

Removes a slot from the agenda of the rule engine. Returns an integer status.

Slot Datatype Codes

GetDataType

NxSlotDataTypeEnum NDNxSlot::GetDataType(void);

Returns the datatype of the slot. The values returned are specified by the NxSlotDataTypeEnum values.

Slot Meta-Information

GetPrompt

Str NDNxSlot::GetPrompt(void);

Returns the prompt string to be used when asking a question about this slot.

GetWhy

Str NDNxSlot::GetWhy(void);

Returns the Why information associated with this slot.

GetComments

Str NDNxSlot::GetComments(void);

Returns the user comments string associated with the slot.

GetFormat

Str NDNxSlot::GetFormat(void);

Returns the format string used with the slot.

SetFormat

void NDNxSlot::SetFormat(CStr *format***);**

Sets the format string used with the slot.

GetQuestionWindow

Str NDNxSlot::GetQuestionWindow(void);

Returns the name of an Open Interface question window to be used when asking a question.

GetPublicInitValue

Str NDNxSlot::GetPublicInitValue(void);

Returns a string containg the public (inheritable) initial value for this slot.

GetPrivateInitValue

Str NDNxSlot::GetPrivateInitValue(void);

Returns a string containg the private (not-inheritable) initial value for this slot.

IsPrivate

Int32 NDNxSlot::IsPrivate(void);

Returns a boolean indicating whether this slot is private or not.

GetInferencePriority

Int32 NDNxSlot::GetInferencePriority(void);

Returns the inference priority number associated with this slot.

GetInferenceSlot

NxSlotPtr NDNxSlot::GetInferenceSlot(void);

Returns the inference priority slot associated with this slot.

GetInheritancePriority

Int32 NDNxSlot::GetInheritancePriority(void);

Returns the inheritance priority number associated with this slot.

GetInheritanceSlot

NxSlotPtr NDNxSlot::GetInheritanceSlot(void);

Returns the inheritance priority slot associated with this slot.

GetValidationHelp

Str NDNxSlot::GetValidationHelp(void);

Returns the string that will be used to provide additional help when a validation violation is discovered on this slot.

GetValidationExecute

Str NDNxSlot::GetValidationExecute(void);

Returns the name of the Execute to be invoked to provide additional user validation for this slot.

GetValidationFunction

Str NDNxSlot::GetValidationFunction(void);

Returns a string representation of a validation function to be evaluated to determine whether this slot represents a valid response.

Slot Inheritability and Inheritance Strategy Codes

GetStrategy

BoolEnum NDNxSlot::GetStrategy(NxSlotStratEnum *strategy***);**

Returns a boolean indicating the setting for the specified strategy. Slot Default Inheritability / Inheritance Strategy codes.

Inheritance

IsDefaultStrategy

BoolEnum NDNxSlot::IsDefaultStrategy(NxSlotDefStratEnum *strategy***);**

Returns a boolean indicating whether the specified strategy is the default strategy.

Slot Volunteer Strategy Codes

Volunteer

Int32 NDNxSlot::Volunteer(VarCPtr *value***);**

Volunteers a value to the slot using the default volunteer strategy. Returns integer status.

Volunteer2

Int32 NDNxSlot::Volunteer2(VarCPtr *value***, NxSlotVolStratEnum** *strategy***);**

Volunteers a value to the slot using the specified volunteer strategy. Returns integer status.

Index

U

W