Neuron Data Elements Environment
Intelligent Rules Element

Version 4.1

C Programmer's Guide

Contents

3. The C Library

4. NXP_GetAtomInfo Routine

5. NXP_SetAtomInfo Routine

6. NXP_Edit Functions

7. NXP_Context Functions

A. Retrieving Rules Element Information

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. If you don't understand the Rules Element, you may need to review the Getting Started manual.

Organization

The first several times you use this manual, you will probably just read Chapter One, "Overview" and Chapter Two, "Primer." They help you understand how to use the API. After you are familiar with the API, use Chapter Three, "C Library," Chapter Four, "NXP_GetAtomInfo," and Chapter Five, "NXP_SetAtomInfo" as a reference to remind you of syntax. Chapter Six, "Edit Functions" teaches you how to use API routines to modify object/class relationships in a knowledge base. Chapter Seven, "Context Functions" shows how to use the context switching API to invoke an independent session of the Rules Element.

Here are more details on the contents of each chapter:

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

- Introduces the icons for each platform.
- Explains the system requirements to run the Rules Element.
- Describes the interfaces that are available and how they affect the Rules Element API.
- Describes the format for the Rules Element's interface on each platform.
- Introduces the contents of the required #include file.
- Reviews Rule Element atoms and discusses how they relate to the Rules Element API.

Chapter Two, "C Primer" uses examples to teach you the fundamentals of using the application programming interface. It starts with a program, `hello.c`, to call the Rules Element. `hello.c` is used to illustrate the components required of every program. It is also used to describe how to compile and link on each platform. Then, the chapter uses variations of `hello.c` to illustrate various tasks using the application programming interface. For more advanced programmers, the last section describes more advanced features of the application programming interface.

Chapter Three, "C Library" is a reference for all the C routines in the application programming interface. They are organized alphabetically within the chapter. The beginning of the chapter gives you an overview of the purpose of each routine.

Chapter Four, "NXP_GetAtomInfo" is a reference on the most commonly-used routine, NXP_GetAtomInfo, which can obtain any

information about any atom. Macros and constants are provided to help you use this routine, and they are organized alphabetically within the chapter. The beginning of the chapter gives you an overview of the macros and constants.

Chapter Five, "NXP_SetAtomInfo" describes the features available for the application development effort that let the developer implement knowledge structures in the Rules Element environment.

Chapter Six, "Edit Functions" teaches you how to use NXP_CreateObject and NXP_DeleteObject to modify object / class relationships in a knowledge base. It also describes other routines you can use to access the editors in the Rules Element.

Chapter Seven, "Context Functions" teaches you how to use the context switching API to invoke an independent session of the Rules Element while already in a session, and not have the knowledge bases / name spaces collide. For example, an application may be in a question handler, and in order to answer the question, it may be necessary to run another KB to get the answer.

Documentation Conventions

Throughout the manual, we use the following formatting conventions.

`% user action`	Text preceded by a system prompt and in this typeface indicates a command that you must enter exactly as shown.
`% run filename`	Text in this typeface and in italics indicates names that you supply, such as file names.
`source code`	Text in this monospaced typeface indicates program examples.
`filenames`	Text in this typeface are file names or directories.

In the rest of this manual, the Intelligent Rules Element is called the Rules Element.

Related Manuals

The library of manuals for the Intelligents Rules Element is designed to help you do different tasks. This table helps you understand which manual you need:

If you want to do this	Then read this manual
Learn about the Rules Element.	Getting Started
Learn about the Rules Element agenda, knowledge representation, and inference engine control.	Language Programmer's Guide
Learn to use the Rules Element through the graphical interface.	User's Guide
Look up encyclopedic information about the Rules Element.	Language Reference
Exchange data between your database and a the Rules Element knowledge base.	Language Reference

Learn to use Open Editor's main windows to create GUI libraries and modules.	Open Interface User's Guide
Look up widget editor information.	Open Interface User's Guide

Users who receive the Intelligent Rules Element packaged with other Neuron Data products, including the Open Interface Element and the Data Access Element, will have other documents in addition to the Rules Element documents described above.

Chapter

1 API Overview

This chapter describes the hardware platforms supported by the Intelligent Rules Element Application Programming Interface (API), the supplied #include file called nxpdef.h, and the concepts of calling in and calling out.

Introduction

After reading this chapter, you will understand how the Rules Element API and your hardware platform are related. You will also understand the format of the API on your platform and how to access the API using the supplied #include file. The following list identifies questions that this chapter provides answers to.

About nxpdef.h

- What is the nxpdef file?
- What does it contain?
- How do I use nxpdef macros?
- How does the nxpdef file declare API routines?

Calling In and Calling Out

- How do I call the Rules Element?
- How can the Rules Element call routines?

About the Format

Generally, the API consists of a library of C-language routines. With one exception, the version of the Rules Element doesn't affect how much of the API you can use. The exception is NXPGFX_Control. NXPGFX_Control allows you to initialize, start, and end the graphical interface. For example, if you are using the API and your program initializes the graphical interface with this routine:

```
NXPGFX_Control(NXPGFX_CTRL_INIT);
```

you are put into the development interface - what you see when you start the development system version of the Rules Element, complete with windows and icons. To return to your program, use the Quit command in the development interface.

About nxpdef.h

To use the API, you need to use an include file supplied by Neuron Data called `nxpdef.h`. Include this file in a C or C++ program like this:

```
#include <stdio.h>
#include <nxpdef.h>
```

```
main()
{
  .
  .
  .
```

nxpdef.h contains definitions that you need in order to use the API. For example, it declares all the API routines as external functions that return an integer. This allows you to compile your file, using API routines in the code, without receiving undeclared function errors. The references to the API routines are resolved when you link your program to the object library of the Rules Element.

What nxpdef.h Contains

nxpdef.h contains type definitions, constant definitions, macros, and function declarations.

The type definitions include definitions for types such as AtomId, which is a value that identifies an atom.

The constant definitions include definitions such as NXP_ERR_INVATOM, which is an error code that indicates an invalid atom.

The macros define an easier way to use the API because they make it easier to use NXP_GetAtomInfo. The macros in nxpdef.h are described in the next section.

The function declarations define the C routines of the API. They are described later in this chapter in the section, "How nxpdef.h Declares Functions"

Using nxpdef.h Macros

NXP_GetAtomInfo is a routine that retrieves any information about an atom. To simplify retrieving the most commonly requested information, macros are defined in nxpdef.h. For example, NXP_GETNAME is a macro that uses NXP_GetAtomInfo to retrieve the name of an atom, and you only need to specify three arguments instead of the seven that NXP_GetAtomInfo requires. For more information on the macros in nxpdef.h, see the section on NXP_GetAtomInfo in Chapter Four, "NXP_GetAtomInfo." Every time you can substitute a macro for a NXP_GetAtomInfo routine, that section tells you.

How nxpdef.h Declares Functions

The function declarations in nxpdef.h declare the C routines of the API. nxpdef.h also provides a way for you to perform type checking on the arguments of the routines, if your compiler supports function prototypes. Function prototypes allow functions declarations to type, and optionally name, their arguments.

Macintosh Note: In special cases, Macintosh programmers need to add the following extra argument NXP_ExtTable *TablePtr; For more information, see the Macintosh API manual.

Calling In and Calling Out

Calling in and calling out refers to what has control when using the API: the Rules Element or your program. Calling in and calling out describes the two basic ways that the Rules Element and a program interact through the API.

This section gives you an overview of calling in and calling out, describes how to call in, and describes how to call out.

Calling in: your program controls the sequence of events, and uses API routines to interact with the Rules Element.

Calling out: your routines are embedded inside the Rules Element.

Calling in means your program has control and it interacts with the Rules Element according to the algorithms you've defined in your program.

Calling out means the Rules Element has control and it interacts with your program whenever it finds an EXECUTE statement or an event occurs and you've installed a routine to handle that event. For example, an event occurs when the Rules Element asks a question or sends you an alert message.

You can call in and call out in the same program. For example, your program can initialize the Rules Element, load a knowledge base, suggest a hypothesis, and instruct the Rules Element to knowcess. Up until now, your program has control and you've called in to the Rules Element. With the command to knowcess, control passes to the Rules Element. It processes information until it reaches a question, when it checks to see if you've defined a routine to respond to questions. If so, it calls the routine. The Rules Element is calling out. When the Rules Element finishes knowcessing, control returns to your program.

Calling Into the Rules Element

Call into the Rules Element with the Rules Element's C library routines. For a complete description of C library routines, see Chapter Three, "The C Library."

You can call in to the Rules Element for several reasons:

- To investigate working memory.
- To modify working memory.
- To control the inference engine of the Rules Element.

For example, this routine investigates working memory by obtaining information about an atom's value:

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_VALUE, 0, 0, NXP_DESC_STR,
                    string_value, 255);
```

This routine modifies working memory by changing the value of an atom's slot:

```
NXP_Volunteer(theAtom, NXP_DESC_STR, thePtr, priority);
```

This routine controls the inference engine by instructing the Rules Element to knowcess:

```
NXP_Control(NXP_CTRL_KNOWCESS);
```

How the Rules Element Calls Routines

You can tell the Rules Element to call out by embedding a routine name in an EXECUTE statement in the Rules Element and installing that routine as a handler. Handlers are routines that respond to predefined events. The Rules Element has predefined events for which you can install your own handlers:

Event	When does the handler for this event get called?
Alert	Called when the Rules Element needs to send an alert message to the user or ask the user to confirm something.
Apropos	Called when the Rules Element encounters a Show operator.
Decrypt	Called when the Rules Element needs to decrypt an encrypted knowledge base.
Encrypt	Called when the Rules Element needs to encrypt a knowledge base.
Execute	Called when the Rules Element encounters an Execute statement in a rule or a method. The Execute statement specifies the name of the routine to call, so you can install many routines as Execute handlers.

A window in the Rules Element can be thought of as a box that sends and receives information. Some graphical interfaces represent windows as rectangles with scrollable text.

GetData	Called when the Rules Element receives data from a window.
GetStatus	Called when the Rules Element checks the availability of an interface. For example, you can check if the Transcript is on or off.
Notify	Called when the state of the knowledge base changes. For example, a value changes when an atom is created and therefore the state of the knowledge base changes.
Password	Called when the Rules Element needs the password of an encrypted knowledge base.
Polling	Called after each inference cycle of the Rules Element.
Question	Called when the Rules Element needs to ask a question to get the value of a slot.
SetData	Called when the Rules Element needs to display a message or send data to a window.

For more information, see the description of `NXP_SetHandler` and `NXP_SetHandler2` in Chapter Three, “The C/C++ Library.”

When one of these events occurs, the Rules Element checks to see if you installed your own handler to respond to it. If not, the Rules Element calls its own default handler for that event.

To install one of your routines as a handler, use the `NXP_SetHandler` or `NXP_SetHandler2` routines. For example, this routine installs the routine `MyQuestions` to be called whenever the Rules Element needs to ask a question:

```
NXP_SetHandler(NXP_PROC_QUESTION, MyQuestion, 0);
```

A Note about Atoms

When using the API, most programming errors are caused by confusing atom types. Therefore, we include this section as a programmer's review of atoms.

What is an Atom?

An atom in the Rules Element can be any of the following:

- Class
- Object (for example, an instance of a class)
- Property slot of an object (for example, object.property)
- Knowledge Base

A right-hand side (RHS) action is the action in an "If x, then perform action y else perform action z" statement. RHS stands for right-hand side. EHS stands for the else right-hand side.

- Hypothesis
- Data
- Property
- Rule
- Condition
- RHS action
- EHS action
- Method (such as Order of Sources, or If Change)

This illustration helps show the relationship of the above items:

For example, a data is a property slot, but a property slot is not necessarily a data.

Properties and Property Slots

A property holds the characteristics of the property. Characteristics are items such as boolean, floating point, integer, date, time, or string. A property itself does not have a value.

A property slot is attached to an object or a class and does have a value. For example, `pressure` and `value` are properties, while `tank1.pressure` and `alert.value` are property slots.

Property Slots and Data

A data is a type of property slot. Data are property slots that are used explicity in LHS or RHS of a rule or meta-slot. A property slot is any slot attached to an object.

For example, `tank1.pressure` is a data if it appears in a condition such as:

```
tank1.pressure > 100
```

If pressures are always tested and set, and not used in the left-hand side (LHS) or right-hand side (RHS or EHS) of a rule or method,

`tank1.pressure` is a property slot. This is an example of tank1.pressure as a property slot:

```
<tanks>.pressure > 100
```

Hypotheses and data are always property slots, even if the default property name is omitted in the notebooks, the rule editor, and the network. For example, if you write a rule with the word `alert` as the hypothesis, an object named `alert` is created with property slot `value`. The hypothesis is actually the propert slot `alert.value`, not the object `alert`, but it is displayed by the Rules Element as `alert`.

Using Atoms with Atom ids

Every atom is identified by a value called the atom id. You can get the id of an atom with NXP_GetAtomId by specifying the atom name.

If you have the id of an atom, then you can get information about the atom with NXP_GetAtomInfo. The following table summarizes how to get information about an atom. In the table, the id of an atom is specified with theAtom:

type, such as class, object or property slot

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_TYPE, optAtom, optInt, desc,
                thePtr, len);
```

value type, such as boolean, floating point, or integer (property slots only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_VALUETYPE, optAtom, optInt, desc,
                thePtr, len);
```

value (property slots only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_VALUE, optAtom, optInt, desc,
                thePtr, len);
```

range of possible values (string slots only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_CHOICE, optAtom, optInt, desc,
                thePtr, len);
```

parent classes (objects and classes only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_PARENTCLASS, optAtom, optInt, desc,
                thePtr, len);
```

subclasses (classes only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_CHILDCLASS, optAtom, optInt, desc,
                thePtr, len);
```

subobjects (for objects) or instances (for classes)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_CHILDOBJECT, optAtom, optInt, desc,
                thePtr, len);
```

property slots (objects and classes only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_SLOT, optAtom, optInt, desc,
                thePtr, len);
```

links to classes or objects (objects and classes only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_LINKED, optAtom, optInt, desc,
                thePtr, len);
```

hypotheses (property slots only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_HYPO, optAtom, optInt, desc,
                thePtr, len);
```

LHS or RHS (rules only)

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_LHS, optAtom, optInt, desc, thePtr, len);
```

(or substitute NXP_AINFO_RHS for NXP_AINFO_LHS)

methods (property slots only)

for Order of Sources methods attached to theAtom, use:

```
NXP_GetAtomInfo(theAtom, NXP_AINFO_SOURCES, optAtom, optInt, desc,
                thePtr, len);
```

or for If Change actions methods attached to theAtom, substitute NXP_AINFO_CACTIONS for NXP_AINFO_SOURCES

You can also use NXP_GetAtomInfo to retrieve the list of all the hypotheses, rules, data, methods, objects, classes, and properties in the knowledge base (with NXP_AINFO_NEXT) or information about the current focus of the inference engine (with NXP_AINFO_CURRENT).

Atom ids Aren't Memory Pointers

Atom ids have a data type of `AtomId`, which is defined in `nxpdef.h`. An AtomId is a value that identifies an atom, but it is not a memory pointer. To use atom ids, use the API routines. The routines check if the AtomIds are valid and therefore help to maintain the integrity of the knowledge base. It also ensures that your programs will be compatible with different versions of the Rules Element.

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, Four, Five, Six and Seven.

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 it?

Starting small with hello.c

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

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.c` 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.c` 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.c` and `helloN.tkb` where N is a number between 1 and 12. `helloN.c` 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.π.rsrc` and the THINK project files `helloN.π`)

Specifying Header Files

On Unix and VAX 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.c

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

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

/*************************************************************/

Int    hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
       printf("hello world!\n");
       return 1;
}

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

       ND_Init(argc, argv);
       NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");
       NXPLine_Main();
       ND_Exit();

       return EXIT_OK;
}
```

Here is an explanation of the program:

`#include <nxppub.h>`

This is the include file supplied by Neuron Data that you need to include in order to use the application programming interface.

`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.

`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.c` 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.c`, 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.c` 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.c`, 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()
{
       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.

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 `NXP_SetHandler` in our main procedure just after the initialization of the Rules Element kernel. You need to add the following line:

```
NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");
```

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

Note: The name specified in the third 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.

The complete listing of our `hello1.c` program is now:

```
#define ERR_LIB NEXPERT
#include <nxppub.h>
#include "nxpinter.h"

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

/*************************************************************/

Int    hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
       printf("hello world!\n");
       return 1;
}

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

       ND_Init(argc, argv);
       NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");
       NXPLine_Main();
       ND_Exit();

       return EXIT_OK;
}
```

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

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

```
Int    hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
       printf("%s\n", theStr);
       return 1;
}

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

       ND_Init(argc, argv);
       NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "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.

You can compile and link the `hello2.c` 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:

```
Int hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
       Char   locStr[255];
```

```
        /* theStr is ignored in that case */
        if (nAtoms != 1) {
                printf("Error: hello called with %d atoms\n", nAtoms);
                return 0;
        }
        if (!NXP_GetAtomInfo(theAtoms[0], NXP_AINFO_VALUE, (AtomId)0, 0,
                                        NXP_DESC_STR, locStr, 255)) {
                printf("Error: hello cannot get value\n");
                return 0;
        }
        printf("%s\n", locStr);
        return 1;
}

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

                ND_Init(argc, argv);
                NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "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 `NXP_GetAtomInfo` to obtain the value of the first atom in the array theAtoms.

Note: The `NXP_GetAtomInfo` routine should not fail in our example, but it will fail if `theAtoms[0]` is not the id of a slot. For example, if we write `@ATOMID= message` instead of `@ATOMID= message.Value` in our rule, `theAtoms[0]` will be the id of the object `message`, not the id of the `Value` slot of the object `message`. Object slots have values (see discussion on atoms in Chapter One, "Overview"), but objects as such do not have values, and thus the `NXP_GetAtomInfo` routine will fail if we pass message instead of `message.Value`.

The fifth argument passed to `NXP_GetAtomInfo` is `NXP_DESC_STR`, because we pass a string buffer as a sixth argument. In the last argument, we indicate how many bytes have been allocated for the buffer.

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

Retrieving Atoms by Name with NXP_GetAtomId (hello4)

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

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

The hello routine is modified as follows:

```
Int     hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
        AtomId msgSlot;
        Char   locStr[255];
```

```
        if (!NXP_GetAtomId("message.Value", &msgSlot,
                                        NXP_ATYPE_SLOT)) {
                printf("Error: hello cannot get id\n");
                return 0;
        }
        if (!NXP_GetAtomInfo(msgSlot, NXP_AINFO_VALUE, (AtomId)0,
                0, NXP_DESC_STR, locStr, 255)) {
                printf("Error: hello cannot get value\n");
                return 0;
        }
        printf("%s\n", locStr);
        return 1;
}

Int     main_L2(Int, argc, Str*, argv)
{
        HELLO_Init("hello4")

        ND_Init(argc, argv);
        NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");
        NXPLine_Main();
        ND_Exit();

        return EXIT_OK;
}
```

The NXP_GetAtomId routine is described in detail in Chapter Three, “The C Library.”

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 NXP_GetAtomId does a search by name in the working memory. Before, the id of 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 X Windows graphics), you can launch the development environment instead of starting the line mode interpreter. You must replace the NXPLine_Main routine by the two following lines:

```
NXPGFX_Control(NXPGFX_CTRL_INIT);
NXPGFX_Control(NXPGFX_CTRL_START);
NXPGFX_Control(NXPGFX_CTRL_EXIT);
```

The first call to NXPGFX_Control initializes the graphics data structures. The second one displays 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.

Note: If you are using a runtime system instead of a development system, NXPGFX_Control is 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 `NXP_SetHandler` 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 `hello.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, Str*, argv)
{
       AtomId testHypo;
       KBId   testKB;

       HELLO_Init("hello6")

       ND_Init(argc, argv);
       NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");

       printf("loading hello6.tkb\n");
       if (!NXP_LoadKB("hello6.tkb",&testKB)) {
              printf("Main: error %d while loading KB\n",
              NXP_Error());
              return EXIT_FAIL;
       }
       if (!NXP_GetAtomId("test_hello", &testHypo, NXP_ATYPE_SLOT)) {
              printf("Main: error %d in get hypo id\n",
              NXP_Error());
              return EXIT_FAIL;
       }
       if (!NXP_Suggest(testHypo, NXP_SPRIO_SUG)) {
              printf("Main: error %d in suggest\n",
              NXP_Error());
              return EXIT_FAIL;
       }
       printf("Starting session\n");
       NXP_Control(NXP_CTRL_KNOWCESS);
       ND_Exit();

       return EXIT_OK;
}
```

The code should be self explanatory. You can read the `NXP_Suggest` and `NXP_Control` descriptions in Chapter Three, "The C Library." The last `NXP_Control` routine will return when the knowcess is complete.

This example also illustrates the error handling mechanism. The `NXP_` routines (except `NXP_Error`) return 1 on success and 0 on failure. If a routine fails, you can call `NXP_Error` which will return a code describing the error more precisely. In our example, the error code returned by `NXP_Error` is included in the error message (formatted by `printf`).

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, Str*, argv)
{
       int    running= 1;
       AtomId testHypo;
       KBId   testKB;

       HELLO_Init("hello7")

       /* startup: same as before without error handling */
       ND_Init(argc, argv);
       NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)hello, "hello");

       printf("loading hello7.tkb");
       if (!NXP_LoadKB("hello7.tkb",&testKB)) {
              printf("Main: error %d while loading KB\n", NXP_Error());
              ND_Exit();
              return EXIT_FAIL;
       }

       if (!NXP_GetAtomId("test_hello", &testHypo, NXP_ATYPE_SLOT)) {
              printf("Main: error %d in get hypo id\n", NXP_Error());
              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':
                     NXP_Suggest(testHypo, NXP_SPRIO_SUG);
                     break;
              case 'k':
                     NXP_Control(NXP_CTRL_KNOWCESS);
                     break;
              case 'r':
                     NXP_Control(NXP_CTRL_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 until 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 id of 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(AtomId, 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 */
        NXP_Volunteer(slot, NXP_DESC_STR, answer, NXP_VSTRAT_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 `NXP_SetHandler` routine. The following line must be inserted in our main procedure after the initialization of the Rules Element `ND_Init(argc, argv)`:

```
NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)MyQuestion,
(Str)0);
```

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 `NXP_GetAtomInfo` function of 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 the `NXP_GetAtomInfo` routine. We 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, AtomId*, theAtoms)
{
       Int    i;
```

```
        Char   name[255];
        AtomId myObject;
        AtomId myClass;
        AtomId rankProp;
        AtomId rankSlot;

        if (!NXP_GetAtomId("test_class", &myClass, NXP_ATYPE_CLASS)) {
               printf("test_class does not exist\n");
               return 0;
        }

        if (!NXP_GetAtomId("rank", &rankProp, NXP_ATYPE_PROP)) {
               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 */
               NXP_CreateObject((AtomId)0, name, myClass, &myObject, 0);

               /* get the id of the rank slot of myObject */
               if (!NXP_GetAtomInfo(myObject, NXP_AINFO_SLOT,
                      rankProp, 0, NXP_DESC_ATOM, (char*)&rankSlot,0)) {
                      printf("rank slot was not created\n");
                      return 0;
               }
               /* set rank to i */
               NXP_Volunteer(rankSlot, NXP_DESC_INT, (Str)&i,
                                          NXP_VSTRAT_CURFWRD);
        }
        return 1;
}
```

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 `NXP_CreateObject` routine 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 `NXP_GetAtomInfo` with the appropriate parameters to obtain the id of the `rank` slot of the new object and then calling `NXP_Volunteer` to assign a value to this slot.

Note: By default, values assigned with `NXP_Volunteer` are not set immediately. They are queued and set only when the inference engine starts or resumes its processing. As a result, calling `NXP_GetAtomInfo` with the `NXP_AINFO_VALUE` code to get the value just after it has been assigned with `NXP_Volunteer` will return the old value of the slot, not the new one. Nevertheless, you can set the `NXP_VSTRAT_SET` bit in the fourth argument of `NXP_Volunteer` if you want the value to be set immediately. You should read the description of the `NXP_Volunteer` routine in the C library chapter 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(AtomId, obj)
{
       Int    len;
       Int    i;
       AtomId slot;
       Char   buf[255];

       /* get number of slots */
       NXP_GetAtomInfo(obj, NXP_AINFO_SLOT, (AtomId)0, -1,
                                     NXP_DESC_INT, (Str)&len, 0);

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

              /* get ith instance */
              NXP_GetAtomInfo(obj, NXP_AINFO_SLOT, (AtomId)0, i,
                                   NXP_DESC_ATOM, (Str)&slot, 0);

              /* get its name and print it */
              NXP_GetAtomInfo(slot, NXP_AINFO_NAME, (AtomId)0, 0,
                                     NXP_DESC_STR, buf, 255);
              printf("\n\t%s", buf);

              /* get its value and print it */
              NXP_GetAtomInfo(slot, NXP_AINFO_VALUE, (AtomId)0, 0,
                                     NXP_DESC_STR, buf, 255);
              printf(" = %s", buf);
       }
}
```

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

void   ListObjects L0()
{
       AtomId obj;
       Int    type;
       Char   buf[255];

       NXP_GetAtomInfo((AtomId)0, NXP_AINFO_NEXT, (AtomId)0,
                     NXP_ATYPE_OBJECT, NXP_DESC_ATOM, (Str)&obj, 0);

       while (obj) {

              /* get object name */
              NXP_GetAtomInfo(obj, NXP_AINFO_NAME, (AtomId)0, 0,
                                          NXP_DESC_STR, buf, 255);

              /* is it a temporary object */
              NXP_GetAtomInfo(obj, NXP_AINFO_TYPE, (AtomId)0, 0,
                                   NXP_DESC_INT, (Str)&type, 0);

              /* print the object */
              if (type & NXP_ATYPE_TEMP) printf("\n+ %s", buf);
              else printf("\n%s", buf);

              ListSlots(obj);

              /* get next object */
              NXP_GetAtomInfo(obj, NXP_AINFO_NEXT, (AtomId)0,
                     NXP_ATYPE_OBJECT, NXP_DESC_ATOM, (Str)&obj, 0);
       }
}

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

void   ListInstances L1(AtomId, class)
{
       Int    len;
       Int    i;
       AtomId obj;
       Char   buf[255];

       /* get number of instances */
       NXP_GetAtomInfo(class, NXP_AINFO_CHILDOBJECT, (AtomId)0, -1,
                                   NXP_DESC_INT, (Str)&len, 0);

       for (i = 0; i < len; i++) {
              /* get ith instance */
              NXP_GetAtomInfo(class, NXP_AINFO_CHILDOBJECT, (AtomId)0, i,
                                          NXP_DESC_ATOM, (Str)&obj, 0);

              /* get its name and print it */
              NXP_GetAtomInfo(obj, NXP_AINFO_NAME, (AtomId)0, 0,
                                          NXP_DESC_STR, buf, 255);
              printf("\n\t%s", buf);
       }
}
```

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

void   ListClasses L0()
{
       AtomId class;
       Char   buf[255];

       NXP_GetAtomInfo((AtomId)0, NXP_AINFO_NEXT, (AtomId)0,
                    NXP_ATYPE_CLASS, NXP_DESC_ATOM,(Str)&class, 0);

       while (class) {

              /* get class name and print it */
              NXP_GetAtomInfo(class, NXP_AINFO_NAME, (AtomId)0, 0,
                                        NXP_DESC_STR, buf, 255);
              printf("\n%s", buf);

              /* display list of instances */
              ListInstances(class);

              /* get next class */
              NXP_GetAtomInfo(class, NXP_AINFO_NEXT, (AtomId)0,
                     NXP_ATYPE_CLASS, NXP_DESC_ATOM, (Str)&class, 0);
       }
}
```

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 (`NXP_AINFO_NEXT`), a NULL atom is passed as input to the first routine. The first atom in the list is returned by the first routine. Then the current atom id is passed as input to `NXP_GetAtomInfo` which returns the next atom id in the list or NULL if the end of list has been reached.
- With the second protocol (`NXP_AINFO_CHILDOBJECT`, `NXP_AINFO_SLOT`), `NXP_GetAtomInfo` is first called with -1 as the fourth parameter. This first routine returns the number of atoms in the list. Then an integer i ranging from 0 to `len-1` (where len is the number of atoms returned by the first routine) is passed as the fourth argument and the id of the `(i+1)th` atom is returned.

Remarks on NXP_GetAtomInfo

You can experiment with other `NXP_GetAtomInfo` codes 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.

The `NXP_GetAtomInfo` routines take seven arguments and are thus difficult to read and write. The `nxppub.h` file contains macro definitions

for the most useful NXP_GetAtomInfo routines. Our last example hello12.c uses the macros instead of NXP_GETATOMINFO routines.

Using NXP_GetAtomInfo with other information codes should not raise any special problems. Confusing the different atom id types (classes, objects, properties, slots, hypotheses, data, etc.) causes most of the problems encountered by developers during early stages of their development. You should refer to Chapter One, "Overview" for a precise classification of the atom ids. The most common sources of confusion are:

- Slots (hypotheses or data are slots) and objects. Slots have values (obtained with NXP_AINFO_VALUE), 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 NXP_GetAtomInfo retrieves the current information from the working memory. It never triggers the inference or inheritance mechanisms. For example the order of sources methods are not triggered when you call NXP_GetAtomInfo with the NXP_AINFO_VALUE code.

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.

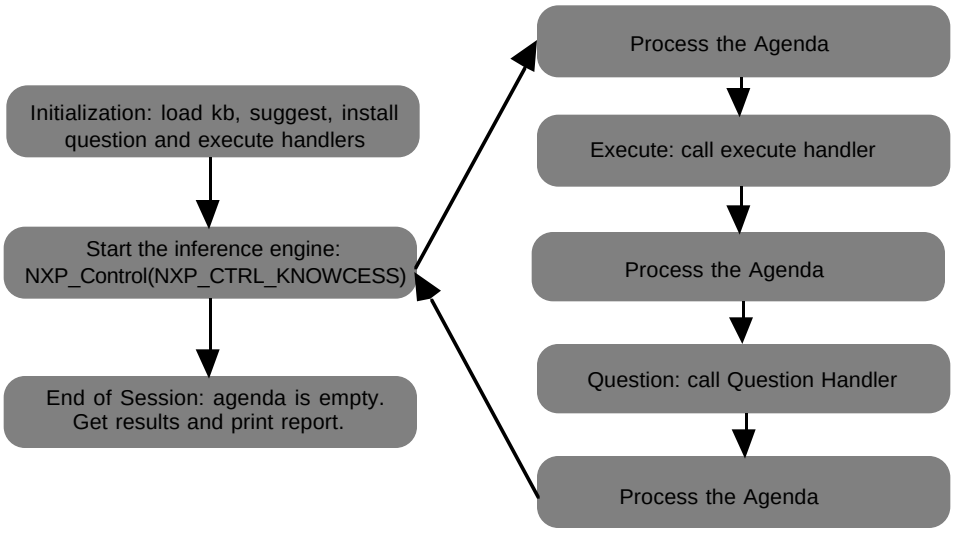
Interrupting a Session (hello10 - Part 1)

When you call NXP_Control(NXP_CTRL_KNOWCESS) to start a session your NXP_Control routine starts the inference engine and returns to its caller only when the session is finished (the agenda of the inference engine is empty) unless you interrupt the session with a NXP_Control(NXP_CTRL_STOPSESSION) during the session.

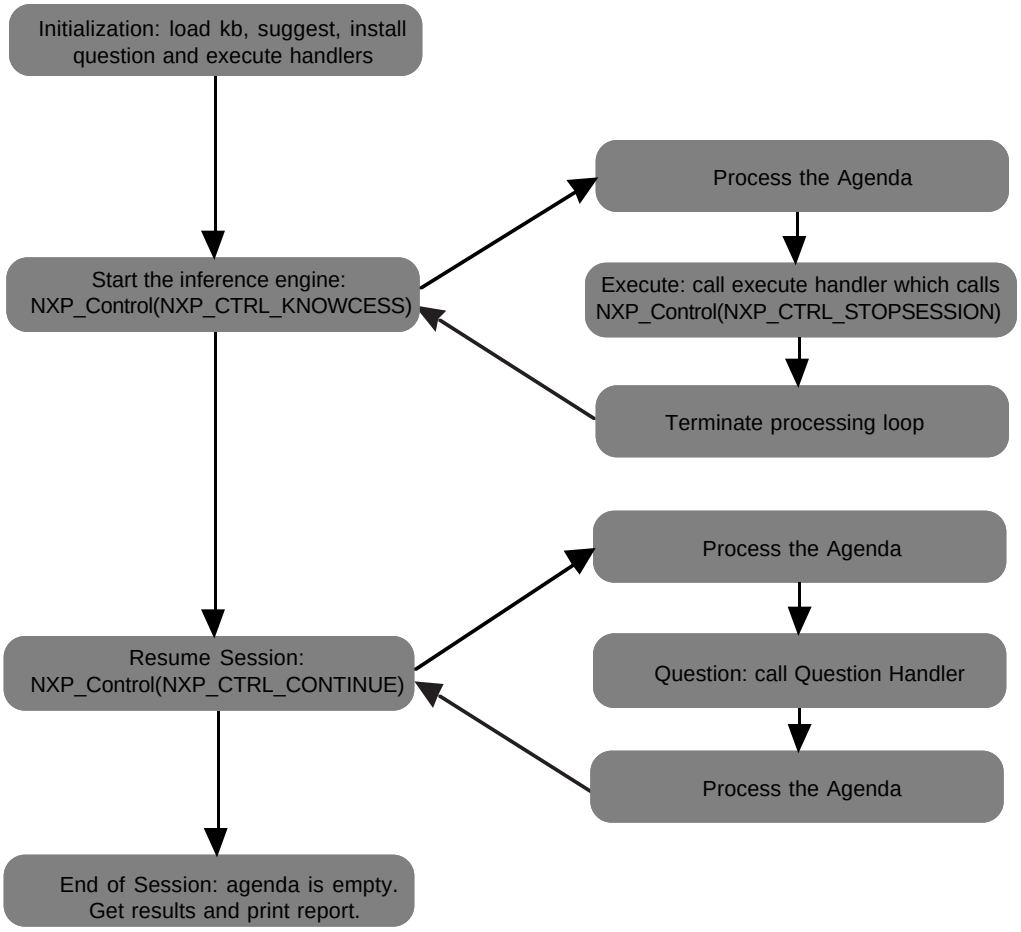
Since the caller of NXP_Control(NXP_CTRL_KNOWCESS) doesn't receive control until the knowcess is complete, you can only stop the session from a handler that you have set and that will be called by the inference engine.

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 `NXP_Control` with the `NXP_CTRL_STOPSESSION` code from the execute routine. The execution flow becomes:



We can demonstrate the use of NXP_CTRL_STOPSESSION with the following Execute routine and rule:

```
Int    hello L3(Str, theStr, Int, nAtoms, AtomId*, 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':
                     NXP_Control(NXP_CTRL_STOPSESSION);
                     return 1;
              case 'n':
                     return 1;
              case '\n':
                     break;
              default:
                     printf("\nInvalid answer");
                     break;
              }
       }
}

              (@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 NXP_Control(NXP_CTRL_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, Str*, argv)
{
       int    running= 1;
       int    restarted = 1;
       AtomId testHypo;
       KBId   testKB;

       HELLO_Init("hello10")

       /* startup: same as before without error handling */
       ND_Init(argc, argv);

       NXP_SetHandler(NXP_PROC_EXECUTE, hello, "hello");
       NXP_SetHandler(NXP_PROC_QUESTION, MyQuestion, (Str)0);

       printf("loading hello10.tkb");
       if (!NXP_LoadKB("hello10.tkb",&testKB)) {
              printf("Main: error %d while loading KB\n", NXP_Error());
              ND_Exit();
              return EXIT_FAIL;
       }
```

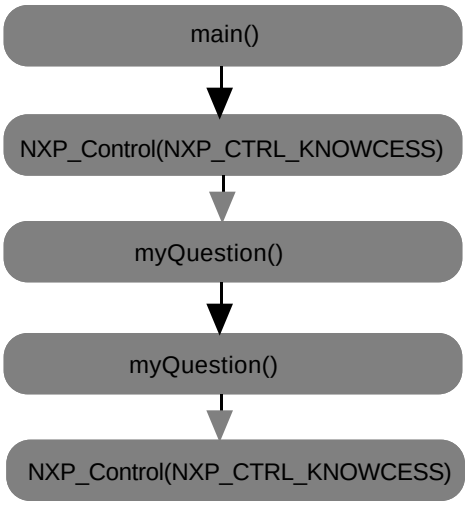
```
	if (!NXP_GetAtomId("test_hello", &testHypo, NXP_ATYPE_SLOT)) {
		printf("Main: error %d in get hypo id\n", NXP_Error());
		ND_Exit();
		return EXIT_FAIL;
	}

	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':
			NXP_Suggest(testHypo, NXP_SPRIO_SUG);
			break;
		case 'k':
			if (restarted) {
				restarted = 0;
				NXP_Control(NXP_CTRL_KNOWCESS);
			} else {
				NXP_Control(NXP_CTRL_CONTINUE);
			}
			break;
		case 'r':
			NXP_Control(NXP_CTRL_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 `NXP_Control(NXP_CTRL_KNOWCESS)` 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 `NXP_Control(NXP_CTRL_KNOWCESS)` routines.

The remedy is to have the question handler interrupt the session and return TRUE without having volunteered an answer. Then the initial `NXP_Control(NXP_CTRL_KNOWCESS)` 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(AtomId, 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');
                     NXP_Control(NXP_CTRL_STOPSESSION);
                     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 */
       NXP_Volunteer(slot, NXP_DESC_STR, answer, NXP_VSTRAT_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];

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

              /* eventually use NXP_CreateObject to create */
              /* a new object  */
              NXP_CreateObject(...);

              /* volunteer theStr into a slot (dynamic or not) /*
              /* with appropriate strategy  */
              NXP_Volunteer(...);
       }
       return 1;
}
```

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

```
NXP_SetHandler(NXP_PROC_POLLING, (NxpIProc)MyPolling, (Str)0);
```

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 `NXP_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:

- NXP_PROC_ALERT
- NXP_PROC_APROPOS
- NXP_PROC_DECRYPT
- NXP_PROC_ENCRYPT
- NXP_PROC_GETDATA
- NXP_PROC_GETSTATUS
- NXP_PROC_NOTIFY
- NXP_PROC_PASSWORD
- NXP_PROC_QUESTION
- NXP_PROC_SETDATA

The Question handler has already been described in this primer.

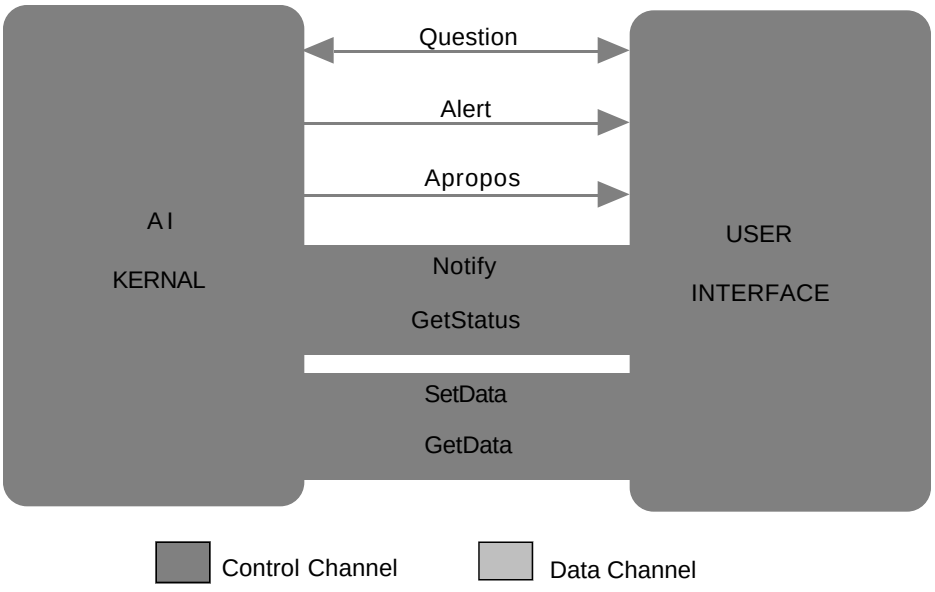
The Alert handler is called by the Rules Element kernel when an error occurs, or if the user needs to confirm an action (in the development environment an alert dialog appears on the screen).

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 bidirectional communication channels between the kernel and the interface:

- A control channel which notifies (`NXP_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 (`NXP_PROC_GETSTATUS`).
- A data channel which allows the kernel to send information to a window (`NXP_PROC_SETDATA`), and to request information from a window (`NXP_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 `NXP_SetHandler` in the C library manual 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)

In this example, we will use one of the existing communication channels. We will write an execute routine that writes a message to the transcript 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 NEXPERT

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

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

/*****************************************************************
       hello: Execute routine
*****************************************************************/

Int     hello L3(Str, theStr, Int, nAtoms, AtomId*, theAtoms)
{
              NXP_SetData(NXP_WIN_TRAN, NXP_ITEM_NONE, -2, theStr);
              return 1;
}

/*****************************************************************
       main
*****************************************************************/

Int    main L2(Int, argc, Str*, argv)
{
              HELLO_Init("hello11")

              ND_Init(argc, argv);
              NXP_SetHandler(NXP_PROC_EXECUTE, hello, "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 environ.
               */

#if !defined(MAC) && ND_GUI
              NXPGFX_Control(NXPGFX_CTRL_INIT);
              NXPGFX_Control(NXPGFX_CTRL_START);
              NXPGFX_Control(NXPGFX_CTRL_EXIT);
#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. If your transcript window is open, the hello world message should be logged along with the trace information when we run the session.

Note: If you want to write your message only to the transcript, you should pass -1 instead of -2 as the third argument to NXP_Notify (the transcript must still be enabled).

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)
{
       if (winId != NXP_WIN_TRAN) return 0;
       if (thePtr == 0) return 0;
       prinf("\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 `NXPGFX_Control` 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 `NXP_SetHandler` routine in the initialization of our program:

```
NXP_SetHandler(NXP_PROC_SETDATA, (NxpIProc)MySetData, (Str)0);
```

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 != NXP_WIN_TRAN || code != NXP_GS_ENABLED)
              return 0;
       *(IntPtr)thePtr = 1;
       return 1;
}
```

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

```
NXP_SetHandler(NXP_PROC_GETSTATUS, (NxpIProc)MyGetStatus,
(Str)0);
```

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 `NXP_Edit` and `NXP_Compile` routines are described in Chapter Six, "NXP_Edit Functions."

With the `NXP_Compile` function, if you can guarantee that your input buffer is syntactically correct and complete, you do not need to install communication handlers. Otherwise you must provide handlers which will treat the errors and the ambiguities (i.e. a data type which cannot be determined from the context).

With the `NXP_SaveKB` routine, you can save knowledge bases which have been created or modified by your program.

Monitoring a Session

You can also install communication handlers to monitor a session. The most interesting handler in that case is the Notify handler. The Rules Element kernel sends notification to the user interface when atoms are created or deleted, when links are modified, and when values are changed. By tapping into the control channel from the kernel to the interface, you can monitor the modifications of the working memory during a session.

Your Notify handler should only process notifications intended for the `NXP_WIN_DDE` window. It should return FALSE for notifications directed to other windows so that they will be processed by the default Notify handler which guarantees the integrity of the development interface. The `nxpmon` example (`nxpmon.c` and `nxpmon.tkb`) demonstrates this capability.

Chapter

3 The C Library

This chapter describes the C library calls as follows.

C Library Calls List

Following is the list of the Rules Element C library calls in alphabetical order. The four most complex functions are decribed in a different chapter: NXP_GetAtomInfo (Chapter Four), NXP_SetAtomInfo (Chapter Five), NXP_Edit (Chapter Six), and NXP_Context (Chapter Seven).

Library Call	Short Description
NXP_BwrdAgenda	Allows for the queueing of events on the backward agenda.
NXP_Compile	Compiles a text buffer of KB definitions.
NXP_Control	Controls the inference engine.
NXP_CreateObject	Creates objects and / or links between objects and classes.
NXP_DeleteObject	Deletes objects and / or links between objects and classes.
NXP_Edit	Initiates the compilation of an atom through the editor protocol (see chapter 6).
NXP_Error	Returns the error code of the last API call.
NXP_GetAtomId	Returns the atom Id from a name.
NXP_GetAtomInfo	Returns different pieces of information about an atom or a knowledge base (see chapter 4).
NXP_GetHandler	Gets procedures set with NXP_SetHandler.
NXP_GetHandler2	Gets procedures set with NXP_SetHandler2.
NXP_GetStatus	Queries the status of the interface.
NXP_Journal	Controls the journaling.
NXP_LoadKB	Loads a knowledge base.
NXP_SaveKB	Saves a knowledge base.
NXP_SetAtomInfo	Provides some control over knowledge bases (see chapter 5).
NXP_SetClientData	Associates client information with an atom.
NXP_SetData	Sends data to the interface.
NXP_SetHandler	Installs user-written procedures.
NXP_SetHandler2	Installs user-written procedures.
NXP_Strategy	Changes the inference strategy.
NXP_Suggest	Puts a hypothesis on the agenda.
NXP_UnloadKB	Unloads or disables a knowledge base.
NXP_Volunteer	Changes the value of a slot.
NXP_WalkNodes	Allows the application of a user-defined function at each node along the inheritance links of an atom.
NXPGFX_Control	Controls the interactive interface of the Rules Element.

NXP_BwrdAgenda

Purpose

This allows for the queueing of events on the Backward agenda. Currently, only slots (data, hypotheses, etc.) can be queued for immediate evaluation. This function will force the processing of the Order of Sources of the atom by the inference engine. This occurs immediately after you give control back to the Rules Element. If you are using the agenda monitor, the slot appears in the current evaluation list.

C Format

The C format is as follows:

NXP_BwrdAgenda (*atom***,** *code***,** *from***);**

Arguments

The following list shows the valid arguments:

```
AtomId    atom;
int       code;
AtomId    from;
```

atom is the slot Id to be queued on the agenda.

code specifies the placement of the queuing of the backward chaining event. It can be set to NXP_CTRL_ATTOP or NXP_CTRL_ATBOTTOM. The default is NXP_CTRL_ATTOP.

from is not used at this time and should be set to 0.

Return Codes

NXP_BwrdAgenda returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid slot Id.
NXP_ERR_NOERR	Call was successful.

Examples

The following example shows a generic Execute procedure to force the Order of Sources of a slot. You can call this procedure from a rule passing a slot (or a list of slots):

```
Execute "ForceOS" @ATOMID=Object.Prop (or <Class>.Prop)
```

Warning: You cannot get the value of a slot that is still unknown by forcing the engine to execute the OS methods "right away" with NXP_BwrdAgenda. The execute routine must return to the Rules Element first before you can have a chance to call NXP_GetAtomInfo / NXP_AINFO_VALUE to get the value.

```
int           ForceOS(theStr, nAtoms, theAtoms)
Str           theStr;
int           nAtoms;
AtomId *theAtoms;
```

```
{
       AtomId   theSlot;
       int      err, ret;

/* We treat the case of 1 atom argument here (nAtoms=1) */
       theSlot = theAtoms[0];
       ret = NXP_BwrdAgenda( theSlot, 0, (AtomId)0 );

       if( ret == 0 ) {
       err = NXP_Error();
       /* Must not be a valid slot */
       ...
       return FALSE;
       }
       return TRUE; /* Execute successful */
}
```

See Also

`NXP_GetAtomInfo / NXP_AINFO_FOCUSPRIO`	Returns the priority of an hypothesis on the agenda

NXP_Compile

Purpose

NXP_Compile compiles a text buffer containing knowledge base definitions. The text buffer must have the format .TKB (see the Text KB Syntax in the User's Guide or look at text knowledge bases generated by the Rules Element).

C Format

The C format is as follows:

int NXP_Compile(*theStr***);**

Arguments

The following list shows the valid arguments:

```
Str    theStr;
```

theStr points to the buffer which contains the knowledge base definitions.

Notes

The compilation takes place in working memory. The atoms created by NXP_Compile become part of the current knowledge base. Use NXP_GetAtomInfo to get information on a KB and NXP_SetAtomInfo to change the current KB. Use NXP_SaveKB to save the knowledge base file if you want your changes to be permanent.

Compilation error messages will typically be passed to you through the Alert mechanism. You can provide an Alert handler to intercept any messages, if desired, see NXP_SetHandler. You will also need a GetData handler to provide a type of property or a nature of object if it is undefined during the compilation.

Return Codes

NXP_Compile returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling `NXP_Error` immediately after the call which has failed. `NXP_Error` returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_ABORT	Compilation was aborted by user or because the description was incomplete and no interface was provided to prompt the user.
NXP_ERR_INTERNAL	Some internal consistency check failed.
NXP_ERR_INVARG1	theStr is NULL.
NXP_ERR_NOERR	Call was successful.
NXP_ERR_NOMEMORY	Memory allocation failed.
NXP_ERR_SYNTAX	The text buffer contained a syntax error.

Examples

The following example illustrates how to compile a hard-coded rule explicitly provided in a C string. Note that the double backslash is required to pass on the interpretation ("\\n\\") since the backslash is an escape sequence in C, and the backslash continuation character at the end of a line indicates that the string is not finished. A complete compilation unit must be sent to the Rules Element this way. A partial text with only the LHS or HYPO, for example, would be incorrect. The example is as follows:

```
NXP_Compile(
  "(@RULE= R1 \
   (@LHS= (>= (n) (0)) (CreateObject ('obj_'\\n\\) (|c|))) \
   (@HYPO= h) \
   (@RHS= (Reset (h)) (Assign (n+1) (n))) )" );
```

You could use the following example if the string theStr contained TKB information (for example, independently read in elsewhere):

```
NXP_Compile( theStr );
```

The Text KB syntax is described in an appendix of the User's Guide manual. You should also look at the text KB files generated by the Rules Element.

See Also

NXP_CreateObject	Create dynamic objects and links.
NXP_SaveKB	Save a knowledge base to a file.
NXP_SetAtomInfo / NXP_SAINFO_CURRENTKB	Set the current knowledge base.
NXP_PROC_GETDATA handler	Used during NXP_Compile in case the Rules Element needs more information.

NXP_Control

Purpose

NXP_Control controls the inference engine of the Rules Element.

C Format

The C format is as follows:

int NXP_Control(*code*);

Arguments

The following list shows the valid arguments:

```
int     code;
```

code can be one of the following values:

Code	Description
NXP_CTRL_CLEARKB	Clears ALL knowledge bases from the Rules Element's memory. This is the same as the "Clear All" command from the Development System's Expert Menu
NXP_CTRL_CONTINUE	Restarts a session which had been stopped by a NXP_CTRL_STOPSESSION code. (Same as NXP_CTRL_KNOWCESS)
NXP_CTRL_EXIT	Notifies the Rules Element that the application does not need to communicate anymore. The Rules Element will clean up any structures allocated.
NXP_CTRL_INIT	Initializes the Rules Element's working memory. This call should be done once, before any other call to the Rules Element library.
NXP_CTRL_KNOWCESS	Starts the inference engine. The call will return only at the end of session or after an execute routine or a non-modal question handler has stopped the session with the NXP_CTRL_STOPSESSION code.
NXP_CTRL_RESTART	Restarts the session by resetting to UNKNOWN all the slots in the knowledge base.
NXP_CTRL_STOPSESSION	Suspends the current engine execution. The session can be restarted with a NXP_CTRL_CONTINUE code. This code can be used to implement a non modal question handler (see the Hello10 example in the Primer).

code can also be a combination of the following to effect ONLY the current evaluation stack:

Code	Description
NXP_CTRL_SETSTOP	Places a special "stop" context in the engine queue (the exact placement in the stack is defined by the codes below). When the engine processes this context, it will stop just as if a NXP_CTRL_STOP had been called. Doing a KNOWCESS will cause the system to resume where it left off. If NXP_CTRL_SETSTOP is used by itself, NXP_CTRL_ATTOP is assumed.
NXP_CTRL_ATTOP	Specifies where the actions NXP_CTRL_SETSTOP or NXP_CTRL_SAVESTRAT will get placed. ATTOP makes it the next item to be processed (unless more items get queued in front of it).
NXP_CTRL_ATBOTTOM	Specified where the actions NXP_CTRL_SETSTOP or NXP_CTRL_SAVESTRAT will get placed. ATBOTTOM makes it the last item to be processed in the current evaluation stack of the engine.
NXP_CTRL_SAVESTRAT	Causes a special "save strategies" context to be saved on the queue by the engine at the placement in the current evaluation stack specified by the two codes mentioned above. The Agenda Monitor will display in the current evaluation stack a NXP_CTRL_SAVESTRAT mark. This allows the developer to modify the engine strategies with NXP_Strategy for instance. However, when the engine re-encounters this context "mark" during its processing, the strategies will be restored to what they were at the time the save context was executed. If NXP_CTRL_SAVESTRAT is used by itself, NXP_CTRL_ATTOP is assumed.

Notes

NXP_CTRL_CLEARKB unload *all* knowledge bases. Use NXP_UnloadKB to unload knowledge bases selectively.

NXP_CTRL_CONTINUE and NXP_CTRL_KNOWCESS will return only after the session is stopped (it can be either during a non-modal question, at the end of session, during a break-point or after an execute routine has called NXP_CTRL_STOPSESSION).

NXP_CTRL_STOPSESSION is not a blocking call, it doesn't stop the session right away! It raises a flag that tells the engine to stop at the next inference cycle. So the session is actually stopped only after your routine returns to the Rules Element, and then the Rules Element returns to the caller of NXP_CTRL_CONTINUE or NXP_CTRL_KNOWCESS.

NXP_CTRL_SAVESTRAT applies to the "current strategies," not the "default strategies."

Return Codes

NXP_Control returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	Invalid code.
NXP_ERR_INVSTATE	code is NXP_CTRL_KNOWCESS but the engine was already started.
	code is NXP_CTRL_STOPSESSION but the engine was not started.
	code is NXP_CTRL_CONTINUE but the engine was not stopped.
NXP_ERR_MATHERROR	A floating point error occurred.
NXP_ERR_NOERR	Call was successful.

Examples

The following example shows how to initialize the Rules Element and start a session with NXP_Control. See the Hello programs in the Primer for example using other codes and to get more information on the flow of control during a session.

```
KBId          theKBId;
AtomId        theAtom;
int           ret;
/* Initialize the Rules Element, always the first call! */
ret = NXP_Control ( NXP_CTRL_INIT );
if(ret == 0)  {  ...  error ...  };
/* Load a knowledge base */
NXP_LoadKB ("Primer.KB", &theKBId );
/* suggest the first hypothesis in the KB */
NXP_GetAtomInfo ( (AtomId)NULL, NXP_AINFO_NEXT, (AtomId)NULL,
                  NXP_ATYPE_HYPO, NXP_DESC_ATOM, (Str)&theAtom, 0 );
NXP_Suggest( theAtom, NXP_SPRIO_SUG );
/* Start the session
 * This call will return when the session is stopped
 * (during a non-modal question for instance)
 */
NXP_Control( NXP_CTRL_KNOWCESS );
```

The following example updates the agenda to execute the message "Execute_Schedule" and stops right after.

```
int    MyExecute()
{
       AtomId schedulerId;
       NXP_Control( NXP_CTRL_SETSTOP);
       NXP_GetAtomId( "Scheduler", &schedulerId,
NXP_ATYPE_CLASS);
       NXP_SendMessage( "Execute_Schedule", schedulerId,
                     (VoidPtr *)NULL, (int *)NULL, 0,
NXP_CTRL_ATTOP);
```

NXP_CreateObject

Purpose

NXP_CreateObject creates dynamic objects in the working memory and / or creates links between objects and other objects or classes (same effect as the operator CreateObject in a rule).

C Format

The C format is as follows:

int NXP_CreateObject(*theAtom***,** *objName***,** *parentAtom***,** *newId***,** *flags***);**

Arguments

The following list shows the valid arguments:

```
AtomId              theAtom;
Str                 objName;
AtomId              parentAtom;
AtomId C_FAR*       newId;
int                 flags;
```

theAtom is either the atom id of an existing object or class or NULL.

If theAtom is NULL, then an object will be created. In that case, *objName* must be specified and will be the name of the new object (objName must be a valid object name).

If theAtom is NULL and there is already an object with the name objName, no object will be created and the call will perform as if the object id had been passed in theAtom (it is more efficient to pass the object id in theAtom than to pass NULL in theAtom and the object name in objName).

If theAtom is not NULL, objName is ignored.

parentAtom is the id of a class or an object to which the (eventually new) object will be linked to. If parentAtom is NULL, the object will not be attached to any new class or object.

If *newId* is not NULL, it should be pointing to a memory AtomId space where the id of the newly created object will be returned.

flags is reserved for future use, and should be set to 0.

Notes

Objects and links created with NXP_CreateObject are dynamic (versus permanent). They belong to the special knowledge base `temporary.kb`

and will be removed at the next restart session (they have a + in front of their name in the interface). You can avoid that by merging `temporary.kb` with your current knowledge base and by making the links permanent: use the call NXP_SetAtomInfo with code NXP_SAINFO_MERGEKB, NXP_SAINFO_PERMLINKKB or NXP_SAINFO_PERMLINK.

You cannot create a class with NXP_CreateObject. Use NXP_Compile or NXP_Edit.

When a new object is created, it will inherit slots only if it is linked to a class or another object (parentAtom not NULL) and the downward inheritability is enabled for these slots (this is the default for slots inherited from classes, not for slots inherited from parent objects). When NXP_CreateObject is used to create a link between an existing object and a class or a parent object, the Rules Element also creates the new slots which are inherited downwards along the new link.

Return Codes

NXP_CreateObject returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theAtom is not a valid class or object id.
`NXP_ERR_INVARG2`	theAtom and objName are NULL or ObjName is an invalid atom name.
`NXP_ERR_INVARG3`	parentAtom is not a valid class or object id.
`NXP_ERR_NOERR`	Call was successful.

Examples

An example that uses the call follows.

```
AtomId       ChildObject;
AtomId       ParentObject;
AtomId       theClass;

/*
 * create two new objects ViewPoint and SubViewPoint.
 */
NXP_CreateObject((AtomId)NULL, "ViewPoint", (AtomId)NULL,
                 &ParentObject, 0);
NXP_CreateObject((AtomId)NULL, "SubViewPoint", ParentObject,
                 &ChildObject, 0);

/* attach SubViewPoint to ViewPoint and ViewPoint to
 * the class of ViewPoints.
 */
NXP_GetAtomId("ViewPoints", &theClass, NXP_ATYPE_CLASS)
NXP_CreateObject(ParentObject, (Str)NULL, theClass,
                 (AtomId C_FAR *)NULL, 0);
```

See Also

`NXP_DeleteObject`	Delete an object or a link.
`NXP_Compile`	Compile KB definition.
`NXP_SetAtomInfo / NXP_SAINFO_MERGEKB, NXP_SAINFO_PERMLINK`	Create permanent objects or links.

NXP_DeleteObject

Purpose

NXP_DeleteObject deletes objects in the working memory and / or deletes links between objects and other objects or classes (same effect as the DeleteObject operator in the interface).

C Format

The C format is as follows:

int NXP_DeleteObject(*theAtom*, *parentAtom***);**

Arguments

The following list shows the valid arguments:

```
AtomId       theAtom;
AtomId       parentAtom;
```

theAtom must be a valid object id.

If *parentAtom* is NULL and if the object is a temporary object (created during the session) it will be removed from the knowledge base. Otherwise only the link between theAtom and parentAtom will be destroyed.

If the link between theAtom and parentAtom is a permanent link (part of the knowledge base), the link is only temporarily unlinked. During the session, theAtom is not considered as belonging to parentAtom any more, but the link will be restored at restart session.

Notes

You cannot delete a class with NXP_DeleteObject. You must use NXP_Edit.

Calling NXP_GetAtomInfo with NXP_AINFO_CHILDOBJECT to query link information for the children objects of an object or class returns permanent links of any deleted objects as well as intact permanent links. To check the type of link, call NXP_GetAtomInfo with NXP_AINFO_LINKED.

Return Codes

NXP_DeleteObject returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error

immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid class or object id.
NXP_ERR_INVARG2	parentAtom is not NULL and is not a valid class or object id.
NXP_ERR_NOERR	Call was successful.

Examples

The following example deletes all the subobjects of theClass. The links which are deleted are always the links indexed by 0. This is possible since the links are deleted immediately.

```
AtomId       theAtom;
AtomId       ParentClass;/* atom id of theClass */
int          Count;

/* Get the atom id */
NXP_GetAtomId("theClass", &ParentClass, NXP_ATYPE_CLASS);

/* Get the number of children */
NXP_GetAtomInfo( ParentClass, NXP_AINFO_CHILDOBJECT,
                          (AtomId)NULL, -1, NXP_DESC_INT, (Str)&Count, 0);

/* loop for all children, delete the link */
while (--Count >= 0) {
      NXP_GetAtomInfo(ParentClass,  NXP_AINFO_CHILDOBJECT,
                     (AtomId)NULL, 0, NXP_DESC_ATOM, (Str)&theAtom, 0);
      NXP_DeleteObject(theAtom, ParentClass);
}
NXP_GetAtomInfo(ParentClass, NXP_AINFO_CHILDOBJECT,
                (AtomId)NULL, -1, NXP_DESC_INT, (Str)&Count, 0);
if (Count != 0) {
      /* there was a problem */
}

/* WARNING !
 * as the links are deleted immediately,
 * the following would be incorrect
 */
for (i = 0; i < Count; i++) {
  /*
   * WARNING
   * for i > Count / 2, the number of sub objects
   * of theClass will be less than Count / 2 because
   * of the links just deleted.  Therefore
   * NXP_GetAtomInfo will return an error.
   */
      NXP_GetAtomInfo(ParentClass, NXP_AINFO_CHILDOBJECT,
                    (AtomId)NULL, i, NXP_DESC_ATOM, (Str)&theAtom, 0);
      NXP_DeleteObject(theAtom, ParentClass)
}
/* but the following would work as expected */
for( i = Count - 1; i >= 0; i-- ) {
      NXP_GetAtomInfo(ParentClass, NXP_AINFO_CHILDOBJECT,
                    (AtomId)NULL, i, NXP_DESC_ATOM, (Str)&theAtom, 0);
      NXP_DeleteObject(theAtom, ParentClass);
}
```

See Also

`NXP_CreateObject`	Create dynamic objects or links.

NXP_Edit

Purpose

NXP_Edit allows a program to edit (create, modify, or delete) objects, classes, rules, or meta-slots. It is maintained in this version mostly to ensure compatibility with Version 1.0. See a full description in Chapter Six.

NXP_Edit can still be used to delete atoms, but NXP_Compile should be called instead to create new atoms. Also, NXP_Edit creates permanent (rather than temporary) objects, attached to a knowledge base. To create temporary objects, use NXP_CreateObject.

C Format

The C format is as follows:

int NXP_Edit(*winId***,** *inAtom***,** *outAtom***,** *mode***);**

Arguments

The following list shows the valid arguments:

```
int           winId;
AtomId        inAtom;
AtomId C_FAR* outAtom;
int           mode;
```

winId describes which editor is invoked. It is one of the following codes:

Code	Description
`NXP_WIN_CLASSEDIT`	To edit a class.
`NXP_WIN_CNTXEDIT`	To edit contexts.
`NXP_WIN_METAEDIT`	To edit meta-slots.
`NXP_WIN_OBJEDIT`	To edit an object.
`NXP_WIN_PROPEDIT`	To edit a property.
`NXP_WIN_RULEDIT`	To edit a rule.

inAtom is either NULL or a valid AtomId.

outAtom is a pointer on a valid memory location where the created or modified AtomId will be returned.

*mod*e is one of the following constants:

Code	Description
`NXP_EDIT_COPY`	Identical to NXP_EDIT_NEW.
`NXP_EDIT_DELETE`	To delete an existing atom.
`NXP_EDIT_MODIFY`	To modify an existing atom.
`NXP_EDIT_NEW`	To create a new atom.

NXP_Edit is described in more detail in chapter Six of this manual.

Return Codes

NXP_Edit returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_ABORT	Compilation was aborted by user or because the description was incomplete and no interface was provided to prompt the user.
NXP_ERR_COMPILEPB	Compilation of new atom did not succeed but the error was not reported correctly.
NXP_ERR_INTERNAL	Some internal consistency check failed.
NXP_ERR_INVARG1	Editor Id is invalid.
NXP_ERR_INVARG2	Input atom is invalid or has the wrong type.
NXP_ERR_INVARG3	outAtom is NULL and a value should be returned.
NXP_ERR_NOMEMORY	Memory allocation failed.
NXP_ERR_SYNTAX	One string received from the interface contained a syntax error.

NXP_Error

Purpose

Returns an error code indicating why the last call to the Rules Element library failed.

C Format

The C format is as follows:

int NXP_Error();

Arguments

None.

Notes

The Rules Element API calls do not return a specific error code, they return 1 on success and 0 on error. It is your responsability to check this returned value and if it is 0 to call NXP_Error to get more information. (In "debugging mode" it is recommended to check systematically all the error codes and display messages with printf, NXP_Alert, etc., to help you solve problems. In "production mode" you should judge how much error checking is needed in your application).

Return Codes

NXP_Error returns the error code of the last called function (apart from any calls to NXP_Error). It returns one of the following values defined in the file `nxpdef.h`:

NXP_Error() Return Code	Explanation
NXP_ERR_ABORT	Compilation was aborted by user or because the description was incomplete and no interface was provided to prompt the user.
NXP_ERR_COMPILEPB	Compilation of new atom did not succeed and the error was not reported correctly.
NXP_ERR_FILEEOF	End of file encountered unexpectedly.
NXP_ERR_FILEOPEN	File could not be opened.
NXP_ERR_FILEREAD	Error reading the file.
NXP_ERR_FILESEEK	Error seeking the file.
NXP_ERR_FILEWRITE	Error writing the file.
NXP_ERR_FORMATERROR	File header is invalid.
NXP_ERR_INTERNAL	Some internal consistency check failed.
NXP_ERR_INVARG1	First argument of routine is invalid.
NXP_ERR_INVARG2	Second argument of routine is invalid.
NXP_ERR_INVARG3	Third argument of routine is invalid.
NXP_ERR_INVARG4	Fourth argument of routine is invalid.
NXP_ERR_INVARG5	Fifth argument of routine is invalid.
NXP_ERR_INVARG6	Sixth argument of routine is invalid.
NXP_ERR_INVATOM	Some atoms saved in the file are invalid.
NXP_ERR_INVSTATE	Argument is in an invalid state.
NXP_ERR_MATHERROR	A floating point error occurred.
NXP_ERR_NOERR	Call was successful.
NXP_ERR_NOMEMORY	Memory allocation failed.
NXP_ERR_NOTFOUND	No atom of the right type was found.
NXP_ERR_NOTKNOWN	The value of the atom is NOTKNOWN.
NXP_ERR_SYNCERROR	The parser lost its synchronization. The contents of the file may be corrupted.
NXP_ERR_SYNTAX	The text file contains a syntax error.
NXP_ERR_UNKNOWN	The value of the atom is UNKNOWN.

NXP_ERR_NOERR means the last called function was successful (the last called function had returned TRUE). NXP_ERR_INVARG[1,2,3,4,5,6] indicates that one argument was incorrect. The other error codes are function dependent. See the description of individual functions for detail.

Example

Example for reporting errors:

```
int   err;

if( NXP_Control( NXP_CTRL_INIT )  == 0 ) {
       err = NXP_Error();
       printf("Error initializing the Rules Element, NXP_Error
= %d", err)
}
```

See Also

`NXP_ErrorIndex`	Returns the index of the array or list argument which failed.

NXP_ErrorIndex

Purpose

If an API call which gets passed either an array or a list of items fails, NXP_ErrorIndex in conjunction with NXP_Error can be used to determine exactly which argument is invalid.

C Format

The C format is as follows:

int NXP_ErrorIndex();

Arguments

None.

Return Codes

For the functions NXP_GetAtomValueArray, NXP_SendMessageArray, NXP_GetAtomValueLengthArray, and NXP_VolunteerArray, NXP_ErrorIndex will return the array index of the invalid argument passed. The index counter starts at 1, not 0, so if 1 is returned by NXP_ErrorIndex, this means the first element of the array is invalid.

Example: If a call to NXP_GetAtomValueArray fails, NXP_Error returns NXP_ERR_INVARG3 and NXP_ErrorIndex returns 2 this means that in the AtomId array (i.e. the third argument passed to NXP_GetAtomValueArray), the second element is invalid.

For the functions NXP_GetAtomValueList, NXP_GetAtomValueLengthList, and NXP_VolunteerList, NXP_ErrorIndex will return the index into the list which has an invalid element.

Example: If a call to NXP_VolunteerList fails, NXP_Error returns NXP_ERR_INVARG4 and NXP_ErrorIndex returns 2, this will mean that in the desc list (the fourth type of argument in the argument list) the second element is invalid.

See Also

NXP_GetAtomValueArray	Getting the info array of atoms.
NXP_SendMessageArray	Sends a message to a list.
NXP_GetAtomValueLengthArray	Getting the value lengths of an array of atoms.
NXP_VolunteerArray	Volunteering an array of values.
NXP_GetAtomValueList	Getting the info list of atoms.
NXP_GetAtomValueLengthList	Getting the value lengths of a list of atoms.
NXP_VolunteerList	Volunteering a list of values.

NXP_GetAtomId

Purpose

NXP_GetAtomId returns the atom Id of an atom given its name and type. Atom Ids are used in all other functions of the Rules Element API.

C Format

The C format is as follows:

int NXP_GetAtomId(*atomName***,** *theAtom***,** *type***);**

Arguments

The following list shows the valid arguments:

```
Str                 atomName;
AtomId C_FAR*       theAtom;
int                 type;
```

atomName is the name of the Atom to be found (or a knowledge base file when type is NXP_ATYPE_KB).

theAtom is a pointer to a valid memory location where the Id of theAtom will be returned.

type is an integer which designates what kind of AtomId should be returned in case there are any conflicts, e.g. if an object and a property had the same name, the call wouldn't know whose AtomId to return without the type argument.

type can be any of the following codes:

Code	Description
NXP_ATYPE_CLASS	The AtomId of a class with the name atomName will be returned in theAtom.
NXP_ATYPE_DATA	The AtomId of a Datum (visible in the DATA notebook) will be returned.
NXP_ATYPE_HYPO	The AtomId of a Hypothesis (visible in the HYPOTHESIS notebook) will be returned.
NXP_ATYPE_KB	The AtomId of a knowledge base will be returned. The string passed in atomName should be the knowledge base name (the file name as it was passed to LoadKB).
NXP_ATYPE_NONE	The Rules Element will first look for an Object or a Class, then for a Slot, and finally for a Property with the name atomName. This option should not be used by applications as they should know what kind of AtomId they are looking for. The search may be slower if this code is used.

Code	Description
NXP_ATYPE_OBJECT	The AtomId of an object will be returned.
NXP_ATYPE_PROP	The AtomId of a Property will be returned.
NXP_ATYPE_RULE	The AtomId of a rule will be returned. The GetAtomId() function matches on rule names when searching for a rule id.
NXP_ATYPE_SLOT	The id of a slot (includes data and hypotheses) will be returned.

Notes

NXP_GetAtomId does not create the atom if it is not in the working memory, (i.e. if it doesn't exist in any of the knowledge bases loaded). Creating atoms can be done with NXP_CreateObject or NXP_Compile.

The id of an atom is a logical reference of a structure used internally by the Rules Element. It is not necessarily a pointer or a handle, so you should not try to use it as a pointer to anything meaningful. Values can be assigned only with the NXP_Volunteer call.

Usually you need to call NXP_GetAtomId only once while working on an atom. In all subsequent calls to the Rules Element you will use the atomId of this atom. However an atomId is not guaranteed to remain valid during the whole session since the atom can be destroyed with DeleteObject, UnloadKB, etc. AtomIds of dynamic objects are not persistent across sessions because dynamic objects are deleted during a restart session.

Return Codes

NXP_GetAtomId returns 1 if *theAtom is a valid AtomId (success), 0 otherwise (in this case, *theAtom is NULL). In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG3	The type argument is invalid.
NXP_ERR_NOTFOUND	No Atom of the right type was found.
NXP_ERR_NOERR	Call was successful.

Examples

Here are several examples:

```
AtomId theAtom, theRule, theKB; /* atomIds to be returned */
NXP_GetAtomId("Sensors", &theAtom, NXP_ATYPE_CLASS)
```

returns in theAtom the Id of the class Sensors (or NULL if it does not exist in the knowledge base).

```
NXP_GetAtomId("Sensor1", &theAtom, NXP_ATYPE_OBJECT)
```

returns in theAtom the Id of the object Sensor1.

```
NXP_GetAtomId("Sensor1.Pressure", &theAtom, NXP_ATYPE_SLOT)
```

returns in theAtom the id of the slot Pressure of the object Sensor1. theAtom is then referring to a slot (it can be boolean, float, string, date, ...).

```
NXP_GetAtomId("Sensor1.Pressure", &theAtom, NXP_ATYPE_DATA)
```

returns in theAtom the id of the slot Pressure of the object Sensor1 only if this slot is a datum in the knowledge base, i.e. it is used in a rule or in a meta slot. theAtom will be NULL even if Sensor1.Pressure exists but is not a datum.

```
NXP_GetAtomId("Sensor1", &theAtom, NXP_ATYPE_SLOT)
```

returns in theAtom the id of the slot Value of the object Sensor1 (Sensor1.Value). Value is a special property that can be omitted in the name.

```
NXP_GetAtomId("Pressure", &theAtom, NXP_ATYPE_PROP)
```

returns in theAtom the id of the property Pressure.

```
NXP_GetAtomId(theStr, &theRule, NXP_ATYPE_RULE)
```

returns in theRule the Id or rule named "foo" if theStr equals "foo". If theStr equals "myRule", it returns the Id of rule myRule in theRule.

```
NXP_GetAtomId("Primer.kb", &theKB, NXP_ATYPE_KB)
```

returns in theKb the Id of the knowledge base Primer.kb if it is loaded. theKB can be used later in calls handling KBs such as NXP_SetAtomInfo + NXP_AINFO_CURRENTKB. Note: you may need to include a directory name within the file name, depending on how the file was loaded.

NXP_GetAtomInfo

Purpose

NXP_GetAtomInfo is a multi-purpose call giving access to any type of information stored in the working memory related to a particular atom. Almost everything visible in the Development System interface can be returned by NXP_GetAtomInfo.

See Chapter Four for a detailed description of this function.

C Format

The C format is as follows:

int NXP_GetAtomInfo(*theAtom***,** *code***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments:

```
AtomId       theAtom;
int          code;
AtomId       optAtom;
Int32        optInt;
int          desc;
Str          thePtr;
int          len;
```

theAtom specifies the atom you want information about. theAtom is an atomId obtained by a previous call to NXP_GetAtomId or by another call to NXP_GetAtomInfo or received as an argument by an Execute routine.

code specifies which type of information is requested. The different values for code are described in Chapter Four.

optAtom is an additional argument with different meanings depending on the value of code.

optInt is an additional argument with different meanings depending on the value of code.

desc is a code which describes the return data type expected by the caller (pointed to by thePtr). It must be one of the NXP_DESC_XXX codes defined in `nxpdef.h`: NXP_DESC_INT, NXP_DESC_FLOAT, NXP_DESC_DOUBLE, NXP_DESC_STR, NXP_DESC_ATOM, etc.

thePtr should point to a valid memory location where the information will be returned.

len is the maximum number of characters that can be returned in thePtr when it is pointing to a string (desc = NXP_DESC_STR). len is not used otherwise.

NXP_GetAtomValueArray

Purpose

NXP_GetAtomValueArray allows the user to obtain the values of an array of Atoms. The Atoms whose values will be obtained can be specified as either an array of AtomId's or an array of character strings which contain the atom names. In the latter case, the character string should contain a slot name.

C Format

The C format is as follows:

int NXP_GetAtomValueArray(*count***,** *type***,** *atoms***,** *descs***,** *ptrs***,** *lens***);**

Arguments

The following list shows the valid arguments:

```
int          count;
int          type;
VoidPtr      atoms;
int          *descs;
VoidPtr      *ptrs;
int          *lens;
```

count is the number of atoms whose values should be returned (size of the array).

type indicates whether atoms is an array of AtomIds or an array of atom names. NXP_DESC_ATOM if atoms points to an array of AtomId's; NXP_DESC_STR if atoms points to an array of character string pointers.

atoms is either an array of AtomId's, or an array of character string pointers.

descs is an array of descriptors for the data being retrieved -- NXP_DESC_INT, NXP_DESC_STR, etc.

Code	Description
NXP_DESC_DATE	This is a new descriptor to speed up the volunteering (and retrieval) of dates in the Rules Element. It describes an array of 6 integers and contains Month, Day, Year (19xx), Hour, Minute, and Second. Obviously, patterns are not involved. Internally, the Rules Element stuffs the integers into a DateRec and calls DateFix() to fill in the day-of-week stuff.

ptrs is an array of pointers to where the values will be returned.

lens is an array of integers for the length of the buffers where the string values will be returned. If some values returned are not strings the matching values in the array len should be set to 0.

Return Codes

NXP_GetAtomValueArray returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero
NXP_ERR_INVARG2	Neither NXP_DESC_ATOM nor NXP_DESC_STR was passed for type.
NXP_ERR_INVARG3	atoms was invalid or the wrong type - not a string slot.
NXP_ERR_INVARG4	descs was invalid.
NXP_ERR_INVARG5	ptrs was invalid or a conversion problem occurred with the data passed.
NXP_ERR_INVARG6	lens was invalid.

If the invalid argument is arg 3,4, 5,or 6 NXP_ErrorIndex (see above) may be used as an index into the array to determine which element was invalid.

Examples

The following example gets the length of string values to allocate memory for an array of buffers to retrieve the string values to.

```
/* Example of NXP_GetAtomValueArray */
static int ReportValues (char *theStr, intnAtoms, AtomId*theAtoms)
{
       int i;
       int *theDescs;
       int *theLens;
       char **newPtrs;
       theLens = (int *)PTR_New(sizeof(int )*nAtoms);
       theDescs = (int *)PTR_New(sizeof(int )*nAtoms);
      /* Get the value lengths of some string slots */
        for (i=0;i<nAtoms;i++)
       theDescs[i]= NXP_DESC_INT;
          NXP_GetAtomValueLengthArray(nAtoms,NXP_DESC_ATOM,
                          theAtoms, theDescs, theLens);
       theLens[2]=  theLens[3]=  theLens[4]= theLens[5]=255;
      /* Allocate the memory for the returned strings*/
       newPtrs = (char **)PTR_New(sizeof(char *)*nAtoms);
       for (i=0;i<nAtoms;i++) {
         theDescs[i]= NXP_DESC_STR;
              newPtrs[i]= (char *)PTR_New(theLens[i]);
   }
NXP_GetAtomValueArray(nAtoms,NXP_DESC_ATOM,(VoidPtr)theAtoms,
             theDescs,   (VoidPtr *)newPtrs,theLens);
}
```

See Also

`NXP_GetAtomValueLengthList`	Getting the value lengths of a list.
`NXP_GetAtomValueLengthArray`	Getting the value lengths of an array of atoms.
`NXP_GetAtomValueList`	Getting the info list of atoms.
`NXP_GetAtomInfo`	Getting the info of an atom.

NXP_GetAtomValueLengthArray

Purpose

NXP_GetAtomValueLengthArray allows the user to obtain the value lengths of an array of Atoms. The Atoms whose value lengths will be obtained can be specified as either an array of AtomId's or an array of character strings which contain the atom names. In the latter case, the character string should contain a slot name.

Note: The value length includes the null terminator character.

C Format

The C format is as follows:

int NXP_GetAtomValueLengthArray(*count***,** *type***,** *atoms***,** *descs***,** *ptrs***);**

Arguments

The following list shows the valid arguments:

```
int          count;
int          type;
VoidPtr      atoms;
int          *descs;
VoidPtr      *ptrs;
```

count is the number of atoms whose value length should be returned (size of the array).

type indicates whether atoms is an array of AtomIds or an array of atom names. NXP_DESC_ATOM if atoms points to an array of AtomId's; NXP_DESC_STR if atoms points to an array of character string pointers.

*atom*s is either an array of AtomId's, or an array of character string pointers.

descs is an array of Ints describing the format of the program's data. Should be NXP_DESC_INT.

*ptrs i*s an array of integer pointers where the lengths will be returned. This array can then be used when calling NXP_GetAtomValueArray.

Return Codes

NXP_GetAtomValueLengthArray returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling

NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero
NXP_ERR_INVARG2	Neither NXP_DESC_ATOM not NXP_DESC_STR was passed for type.
NXP_ERR_INVARG3	atoms was invalid or the wrong type - not a string slot.
NXP_ERR_INVARG4	descs was invalid.
NXP_ERR_INVARG5	ptrs was invalid or a conversion problem occurred with the data passed.

If the invalid argument is arg 3,4,or 5, NXP_ErrorIndex (see above) may be used as an index into the array to determine which element was invalid.

Examples

The following example gets the length of string values to allocate memory for an array of buffers to retrieve the string values to.

```
/* Example of NXP_GetAtomValueLengthArray */
static int ReportValues (char  *theStr, int nAtoms, AtomId* theAtoms)
{
       int i;
       int *theDescs;
       int *theLens;
       char **newPtrs;

       if (nAtoms !=  5) return 0;
       theLens = (int *)PTR_New(sizeof(int )*nAtoms);
       theDescs = (int *)PTR_New(sizeof(int )*nAtoms);
      /* Get the value lengths of some string slots */
        for (i=0;i<nAtoms;i++)
       theDescs[i]= NXP_DESC_INT;
          NXP_GetAtomValueLengthArray(nAtoms,NXP_DESC_ATOM, theAtoms,
theDescs,
                                theLens);
       theLens[2]=  theLens[3]=  theLens[4]= theLens[5]=255;
      /* Allocate the memory */
       newPtrs = (char **)PTR_New(sizeof(char *)*nAtoms);
       for (i=0;i<nAtoms;i++) {
            theDescs[i]= NXP_DESC_STR;
             newPtrs[i]= (char *)PTR_New(theLens[i]);
     }  NXP_GetAtomValueArray(nAtoms,NXP_DESC_ATOM, (VoidPtr)theAtoms,
            theDescs, (VoidPtr *)newPtrs,theLens);
}
```

See Also

NXP_GetAtomValueLengthList	Getting the value lengths of a list.
NXP_GetAtomValueArray	Getting the info array of atoms.
NXP_GetAtomValueList	Getting the info list of atoms.
NXP_GetAtomInfo	Getting the info of an atom.

NXP_GetAtomValueLengthList

Purpose

NXP_GetAtomValueLengthList allows the user to obtain the value lengths of a list of atoms. The atoms whose value lengths will be obtained can be specified as either a list of AtomId's or a list of character strings which contain the atom names. In the latter case, the character string should contain a slot name.

C Format

The C format is as follows:

int NXP_GetAtomValueLengthList(*count***,** *type***,** *atom1***,** *desc1***,** *ptr1***,** *atom2***,** *desc2***,** *ptr2***, ...);**

Arguments

The following list shows the valid arguments:

```
int       count;
int       type;
void      atomx;
int       descx;
VoidPtr   ptrx;
```

*coun*t is the number of atoms to be volunteered.

type is NXP_DESC_ATOM if atoms will be passed as AtomId's. NXP_DESC_STR if atoms will be passed as character strings.

atomx (where x is 1, 2 ,3,...) is the AtomId, or the string name of an Atom, whose value length will be retrieved, depending on whether NXP_DESC_ATOM or NXP_DESC_STR was passed in the type.

descx is the descriptor for the data being received, should be NXP_DESC_INT.

ptrx is a pointer to an integer where the length is to be returned.

Return Codes

NXP_GetAtomValueLengthList returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero

If the invalid argument is arg 3,4,5, or 6, NXP_ErrorIndex (see above) may be used as an index into the list to determine which item was invalid.

Examples

The following example gets the value length of two slots before allocating buffer size and retrieving their values.

```
/* Example of NXP_GetAtomValueLengthList */
      Str          ptr1, ptr2;
      int          len1, len2;
      Char   *str1 = "valve_1.problem";
```

```
      Char   *str2 = "valve_2.problem";
      NXP_GetAtomValueLengthList( 2, NXP_DESC_STR, str1, NXP_DESC_INT,
                                  &len1, str2, NXP_DESC_INT, &len2);
      if ( (len1 ==0) && (len2 ==0 ) return 0;
      ptr1 = PTR_New( len1);
      ptr2 = PTR_New( len2);
      NXP_GetAtomValueList( 2, NXP_DESC_STR, str1, NXP_DESC_STR,
            (VoidPtr)ptr1, len1, str2, NXP_DESC_STR, (VoidPtr)ptr2, len2);
...
```

See Also

`NXP_GetAtomValueLengthArray`	Getting the value lengths of an array.
`NXP_GetAtomValueArray`	Getting the info array of atoms.
`NXP_GetAtomValueList`	Getting the info list of atoms.
`NXP_GetAtomInfo`	Getting the info of an atom.

NXP_GetAtomValueList

Purpose

NXP_GetAtomValueList allows the user to obtain the values of a list of atoms. The atoms whose values will be obtained can be specified as either a list of AtomId's or a list of character strings which contain the atom names. In the latter case, the character string should contain a slot name.

C Format

The C format is as follows:

int NXP_GetAtomValueList(*count***,** *type***,** *atom1***,** *desc1***,** *ptr1***,** *len1***,** *atom2***,** *desc2***,** *ptr2***,** *len2***, ...);**

Arguments

The following list shows the valid arguments:

```
int          count;
int          type;
VoidPtr      atomx;
int          descx;
VoidPtr      ptrx;
int          lenx;
```

count is the number of atoms to be volunteered.

type is NXP_DESC_ATOM if atoms will be passed as AtomId's. NXP_DESC_STR if atoms will be passed as character strings.

atomx (where x is 1, 2, 3 ...) is the AtomId, or the string name of an Atom, whose value length will be retrieved, depending on whether NXP_DESC_ATOM or NXP_DESC_STR was passed in the type.

descx is the descriptor for the data being received, should be NXP_DESC_INT.

ptrx is a pointer to an integer where the length is to be returned.

*lenx i*s the descriptor for the length of the buffer ptrx where the data is being received, and should be set to 0 if the value is not a string.

Return Codes

NXP_GetAtomValueList returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero

If the invalid argument is arg 3,4,5, or 6, NXP_ErrorIndex (see above) may be used as an index into the list to determine which item was invalid.

Examples

The following example gets the value length of two slots before allocating buffer size and retrieving their values.

```
/* Example of NXP_GetAtomValueList */
      Str          ptr1, ptr2;
      int          len1, len2;
      Char         *str1 = "valve_1.problem";
      Char         *str2 = "valve_2.problem";
      NXP_GetAtomValueLengthList( 2, NXP_DESC_STR, str1,
            NXP_DESC_INT, (VoidPtr)&len1, str2, NXP_DESC_INT,
                        (VoidPtr)&len2);
      if ( (len1 ==0) && (len2 ==0 ) return 0;
      ptr1 = PTR_New( len1);
      ptr2 = PTR_New( len2);
      NXP_GetAtomValueList( 2, NXP_DESC_STR, str1, NXP_DESC_STR,
            (VoidPtr)ptr1, len1, str2, NXP_DESC_STR, (VoidPtr)ptr2, len2);
...
```

See Also

NXP_GetAtomValueLengthArray	Getting the value lengths of an array.
NXP_GetAtomValueArray	Getting the info array of atoms.
NXP_GetAtomValueLengthList	Getting the value lengths of a list of atoms.
NXP_GetAtomInfo	Getting the info of an atom.

NXP_GetHandler

Purpose

NXP_GetHandler returns a handler procedure previously set by NXP_SetHandler. It can be used in conjunction with NXP_SetHandler to temporarily change a handler (NXP_GetHandler gets the old handler, NXP_SetHandler is called twice: first to set the new handler and then to restore the old one). See NXP_SetHandler for more information on handlers.

C Format

The C format is as follows:

int NXP_GetHandler(*theCode***,** *theProc***,** *theName***);**

Arguments

The following list shows the valid arguments:

```
Int                 theCode;
NxpIProc C_FAR*     theProc;
Str                 theName;
```

theCode is one of the NXP_PROC codes described in the NXP_SetHandler call description.

theProc is a pointer to a procedure which will receive the result (the handler).

theName is the name of the execute handler if theCode is NXP_PROC_EXECUTE, NULL otherwise.

Return Codes

NXP_GetHandler returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theCode is not a valid code.
NXP_ERR_INVARG2	theProc is NULL.
NXP_ERR_NOERR	Call was successful.

Examples

The following example illustrates how to retrieve the current Alert handler so you can install your own. You can call NXP_SetHandler() later to restore the first one:

```
NxpIProc  fcn;

/* get the Alert handler */
NXP_GetHandler( NXP_PROC_ALERT, &fcn, 0 );
/* change it with our own alert function */
NXP_SetHandler( NXP_PROC_ALERT, myAlert, 0 );

/* change it back to the original function */
NXP_SetHandler( NXP_PROC_ALERT, fcn, 0 );
```

The following example illustrates how to retrieve an Execute handler having the name "myExecute":

```
NxpIProc  fcn;

NXP_GetHandler( NXP_PROC_EXECUTE, &fcn, "myExecute" );
```

See Also

NXP_GetHandler2	Gets handlers set with NXP_SetHandler2.
NXP_SetHandler2	Installs handlers.

NXP_GetHandler2

Purpose

NXP_GetHandler2 returns a handler procedure set with NXP_SetHandler2. NXP_GetHandler2 has the same functionalities as NXP_GetHandler except that it takes two additional arguments, type and arg.

C Format

The C format is as follows:

int NXP_GetHandler2(*theCode***,** *theProc***,** *theName***,** *type***,** *arg***);**

Arguments

The following list shows the valid arguments:

```
int          theCode;
NxpIProc     *theProc;
Str          theName;
int          *type;
unsigned long C_FAR *arg;
```

theCode is one of the NXP_PROC codes described in the NXP_SetHandler or NXP_SetHandler2 call description.

theProc is a pointer to a procedure where the address of the handler will be returned.

theName is used only if theCode equals NXP_PROC_EXECUTE. It is the name of the Execute handler to get information from (after calling NXP_GetAtomInfo with NXP_AINFO_PROCEXECUTE to get the list of Execute handler names, for instance).

type is a pointer to an integer where the type of handler will be returned (as set with NXP_SetHandler2). Your application usually will not need this information.

arg is a pointer to an unsigned long where the custom information passed to NXP_SetHandler2 will be returned.

Notes

If you have installed a handler using NXP_SetHandler and then use NXP_GetHandler2 on it, *type will be set to NXP_HDLTYPE_SETHANDLER and *theProc will NOT be set to your function pointer. It is recommended that you call back using NXP_GetHandler to get the correct *theProc value.

Return Codes

NXP_GetHandler2 returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theCode is not a valid code.
NXP_ERR_INVARG2	theProc is NULL.

NXP_Error() Return Code	Explanation
NXP_ERR_NOTFOUND	The Execute handler was not found (theCode was equal to NXP_PROC_EXECUTE and no Execute was installed with name theName.

Examples

The following example shows how to install an Execute handler and get it back with NXP_GetHandler2.

```
/* generic Execute function */
int myExecute(arg, str, nAtoms, atoms)
ULong        arg;
Str          str;
int          nAtoms;
AtomId C_FAR* atoms;
{
      ...
}

/* install myExecute as a user handler with the name doThis */
NXP_SetHandler2(NXP_PROC_EXECUTE, myExecute, "doThis",
                  NXP_HDLTYPE_USER, (unsigned long)0);

/* use NXP_GetHandler2 to get back the "doThis" handler */

NxpIProc     theProc;     /* function address */
int          type;        /* type of handler */
unsigned long arg;        /* custom argument */

NXP_GetHandler2(NXP_PROC_EXECUTE, &theProc,"doThis", &type, &arg);
/* theProc will be set to myExecute */
/* type will be set to NXP_HDLTYPE_USER = 0x0100 */
/* arg will be set to 0 */
```

See Also

NXP_GetHandler	Gets handlers set with NXP_SetHandler
NXP_SetHandler2	Installs handlers.

NXP_GetMethodId

Purpose

NXP_GetMethodId returns the atom Id of a method given the atom it is attached to and its type. Atom Ids are used in most functions of the Rules Element API.

C Format

The C format is as follows:

int NXP_GetMethodId(*methodName***,** *theMethod***,** *theAtom***,** *type***);**

Arguments

The following list shows the valid arguments:

```
Str                methodName;
AtomId C_FAR*      theMethod;
```

```
AtomId              theAtom;
int                 type;
```

methodName is the name of method to be found.

theMethod is a pointer to a valid memory location where the Id of theMethod will be returned.

theAtom is the AtomId of the atom theMethod is attached to.

type is an integer which designates what kind of methods should be returned in case there are any conflicts, (for example, when there is a public and a private method).

type can be any of the following codes:

Code	Description
NXP_AINFO_PRIVATE	theMethod is private and not inheritable by any relative.
NXP_AINFO_PUBLIC	theMethod is public and inheritable by any child.

Notes

NXP_GetMethodId does not create the method if it is not in the working memory, (i.e. if it doesn't exist in any of the knowledge bases loaded). Creating a method can be done with NXP_Compile.

The id of a method is a logical reference of a structure used internally by the Rules Element. It is not necessarily a pointer or a handle, so you should not try to use it as a pointer to anything meaningful.

Usually you need to call NXP_GetMethodId only once while working on a method. In all subsequent calls to the Rules Element you can use the atomId of this method. However an atomId is not guaranteed to remain valid during the whole session since the method can be destroyed with UnloadKB, etc.

Return Codes

NXP_GetMethodId returns 1 if *theMethod is a valid AtomId (success), 0 otherwise (in this case, *theMethod is NULL). In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	methodName was invalid.
NXP_ERR_INVARG2	methodId was a NULL pointer.
NXP_ERR_INVARG3	TheAtom is not a valid atom - not a slot, an object, a class or a property.
NXP_ERR_INVARG4	The type argument is invalid.
NXP_ERR_NOTFOUND	No Method of the right type was found.
NXP_ERR_NOERR	Call was successful.

Examples

The following example returns the atomid of the public method called "Execute_Schedule" attached to the class "scheduler".

```
/* Example of NXP_GetMethodId */
AtomId atom, methodId;
NXP_GetAtomId( "scheduler", &atom, NXP_ATYPE_CLASS);
NXP_GetMethodId( "Execute_Schedule", &methodId, atom,
NXP_AINFO_PUBLIC);
```

See Also

NXP_VolunteerArray	Volunteering an array of values.
NXP_Volunteer	Volunteering a single value.

NXP_GetStatus

Purpose

NXP_GetStatus queries the status of the interface. This call is valid only with the development system version (the runtime library doesn't have an interface).

C Format

The C format is as follows:

int NXP_GetStatus(*winId***,** *code***,** *thePtr***);**

Arguments

The following list shows the valid arguments:

```
Int          winId;
Int32        code;
IntPtr       thePtr;
```

winId is the id of the window to which data is sent.

code is an integer describing which information is requested.

thePtr is a pointer to a memory location where the information will be returned.

NXP_GetStatus can be used to query the "write mode" of the following windows (other values of winId are reserved for Neuron Data's internal use):

Window	Code
Transcript	winId = NXP_WIN_TRAN
Current rule	winId = NXP_WIN_RULE
Current hypothesis	winId = NXP_WIN_HYPO
Conclusions	winId = NXP_WIN_CONC

For these windows, code must be NXP_GS_ENABLED. thePtr must point to an integer which will be set to 1 if the window is write enabled, 0 otherwise.

Notes

Typically you would call NXP_GetStatus before you decide to write a message into the Transcript (see the example below).

You can send information to a window with NXP_SetData even if the window is disabled. By calling NXP_GetStatus first, you gain some speed in case the window is disabled because you do not need to format the message before sending it.

Return Codes

NXP_GetStatus returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return the following:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	winId is invalid.
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG3	thePtr is NULL.
NXP_ERR_NOERR	Call was successful.

Examples

This example shows how to check the Transcript's status before sending the string.

```
int enabled = 0;

/* query the status of the transcript window */
NXP_GetStatus(NXP_WIN_TRAN, NXP_GS_ENABLED, &enabled);

if (enabled) {
       /* write a message to the transcript */
       NXP_SetData (NXP_WIN_TRAN, NXP_CELL_NONE, -1, "This is
a test");
}
```

NXP_Journal

Purpose

NXP_Journal controls the journaling mechanism of the Rules Element. For more information see the description of the Journal window in the Users Guide and Reference manuals.

C Format

The C format is as follows:

int NXP_Journal(*code***,** *theAtom***,** *theStr***);**

Arguments

The following list shows the valid arguments:

```
int          code;
AtomId       theAtom;
Str          theStr;
```

code describes the journaling action:

Code	Description
NXP_JRNL_PLAYEVENT	Reserved for Neuron Data's internal use.
NXP_JRNL_PLAYSTART	Starts a replay. theStr is the filename specification of the input journal file. theAtom is ignored.
	This code can be combined with NXP_JRNL_PLAYSTEP to indicate a step by step replay.
	NXP_JRNL_PLAYSKIPSHOW to indicate that Show conditions should be skipped.
	NXP_JRNL_PLAYNOSCAN to disable the scanning of the file for each value.
NXP_JRNL_PLAYSTOP	Stops the replaying of the current file. theStr and theAtom are ignored.
Code	Description
NXP_JRNL_RECORDEVENT	Reserved for Neuron Data's internal use.
NXP_JRNL_RECORDSTART	Starts recording. theStr is the filename specification of the file into which the session will be recorded. theAtom is ignored.
NXP_JRNL_RECORDSTOP	Stops the current recording. theStr and theAtom are ignored.
NXP_JRNL_STATERESTORE	Restores the state from a file. The state should have been saved previously with a NXP_JRNL_STATESAVE call or from the journaling interface. theStr is the filename specification of the file from which the state will be restored. theAtom is ignored.
NXP_JRNL_STATESAVE	Saves the current state in a machine dependent file. theStr is the filename specification of the file into which the state will be saved. theAtom is ignored.
NXP_JRNL_VALUESSAVE	Saves all the current working memory values in the order they were set in a NXP format file. theStr is the filename specification of the file into which the values will be saved. theAtom is ignored.

Return Codes

NXP_Journal returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_FILEEOF	End of file encountered unexpectedly.
NXP_ERR_FILEOPEN	File could not be opened.
NXP_ERR_FILEREAD	Error reading the file.
NXP_ERR_FILESEEK	Error seeking the file.
NXP_ERR_FILEWRITE	Error writing the file.
NXP_ERR_FORMATERROR	File header is invalid.
NXP_ERR_INVARG1	code is invalid.
NXP_ERR_INVARG2	theAtom is not NULL and is not a valid slot id.
NXP_ERR_INVATOM	Some atoms saved in the file are invalid (NXP_JRNL_STATERESTORE only).
NXP_ERR_NOERR	Call was successful.
NXP_ERR_SYNCERROR	The parser lost its synchronization. The contents of the file may be corrupted.

Examples

The following example illustrates how to save a state in a file (`myfile.sta` in this case):

```
NXP_Journal(NXP_JRNL_STATESAVE, (AtomId)0, "myfile.sta");
```

Here is how to restore from a state file:

```
NXP_Journal(NXP_JRNL_STATERESTORE, (AtomId)0, "myfile.sta");
```

Here is how to replay a journal file step by step:

```
NXP_Journal(NXP_JRNL_PLAYSTART | NXP_JRNL_PLAYSTEP,
             (AtomId)0, "myJournal");
```

NXP_LoadKB

Purpose

NXP_LoadKB loads a knowledge base file.

C Format

The C format is as follows:

int NXP_LoadKB(*kbName***,** *theKBId***);**

Arguments

The following list shows the valid arguments:

```
Str                 kbName;
KBId C_FAR          *theKBId;
```

kbName is a knowledge base file name.

If *theKBId* is not NULL, it must be a pointer to a KBId where the id of the knowledge base will be returned.

Notes

If you are not interested by the KBId you can pass 0. You can get the id later with NXP_GetAtomId and kbName, or with NXP_GetAtomInfo and an atom belonging to the KB.

The Rules Element must be able to find the file kbName in order to load it. If you don't use a full pathname kbName must be located in one of the PATHS directories.

You can trap errors - file I/O error or compilation error - by installing an Alert handler.

Return Codes

NXP_LoadKB returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error

immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_ABORT	Compilation was aborted by user or because the description was incomplete and no interface was provided to prompt the user.
NXP_ERR_FILEEOF	End of file encountered unexpectedly.
NXP_ERR_FILEOPEN	File could not be opened.
NXP_ERR_FILEREAD	Error reading the file.
NXP_ERR_FILESEEK	Error seeking the file.
NXP_ERR_FORMATERROR	File header is invalid.
NXP_ERR_INTERNAL	Some internal consistency check failed.
NXP_ERR_INVARG1	kbName is NULL or points to an empty string.
NXP_ERR_NOERR	Call was successful.
NXP_ERR_NOMEMORY	Memory allocation failed.
NXP_ERR_SYNCERROR	Compiler lost its synchronization. The contents of the file may be corrupted.
NXP_ERR_SYNTAX	The text file contained a syntax error.

Examples

The following code gives a simple example:

```
KBId          theKBId;
int           ret;

/* loads the primer knowledge base */
ret = NXP_LoadKB("primer.kb", &theKBId);
if(ret == 0) {
       printf("Error while loading primer.kb, NXP_Error = %d
\n", NXP_Error());
}
```

See Also

NXP_UnloadKB	Unload a knowledge base.
NXP_SaveKB	Save a knowledge base into a file.

NXP_SaveKB

Purpose

NXP_SaveKB saves a knowledge base into a file.

C Format

The C format is as follows:

int NXP_SaveKB(*kbId***,** *theStr***,** *mode***);**

Arguments

The following list shows the valid arguments:

```
KBId          kbId;
Str           theStr;
int           mode;
```

kbId is the id of the knowledge base to save (see the notes below).

theStr is the filename specification of the knowledge base file.

mode describes saving options. If mode equals 0, the knowledge base is saved in text format without comments. The following bits can be set in the mode argument (see the example below):

Code	Description
NXP_MODE_COMMENTS	Knowledge base saved with comments.
NXP_MODE_COMPILED	Knowledge base saved in compiled form.

Notes

There are several ways of getting the kbId of a knowledge base:

- If the knowledge base was loaded with NXP_LoadKB, its kbId was returned by this function.
- If the knowledge base already exists and you know its name use NXP_GetAtomId.
- If you know an atom belonging to this knowledge base use NXP_GetAtomInfo with NXP_AINFO_KBID.
- The special knowledge base `undefined.kb` containing all the atoms referenced but not defined yet has a kbId equals to 0.
- The special knowledge base `temporary.kb` containing all the atoms created dynamically has a kbId equals to 1.
- The special knowledge base `untitled.kb` containing all the atoms created before any other KB was loaded has a kbId equals to 2.

theStr must be a valid filename for the operating system. If you use a partial pathname (such as "foo.tkb") the file will be saved in the current directory. You can also use full pathnames to save the file in another directory.

NXP_SaveKB will fail if Save has been disabled previously with NXP_SetAtomInfo + NXP_SAINFO_DISABLESAVEKB (useful if you are delivering a protected knowledge base and don't want it saved after your application decrypts it).

Return Codes

NXP_SaveKB returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_FILEOPEN	File could not be opened.
NXP_ERR_FILESEEK	Error seeking the file.
NXP_ERR_FILEWRITE	Error writing the file.

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	kbId is not a valid knowledge base Id.
NXP_ERR_SAVEDISABLED	KB cannot be saved because Save has been disabled
NXP_ERR_NOERR	Call was successful.

Examples

Example of saving the knowledge base "mykb" with various options (we suggest to use the `.tkb` extensions for the text format and the `.ckb` extension for the compiled format):

```
KBId   myKB;

/* saves the KB in text format without the comments */
NXP_SaveKB( myKB, "mykb.tkb", 0 );

/* saves the KB in text format with the comments */
NXP_SaveKB( myKB, "mykb.tkb", NXP_MODE_COMMENTS );

/* save the KB in compiled format without the comments */
NXP_SaveKB( myKB, "mykb.ckb", NXP_MODE_COMPILED );

/* save the KB in compiled format with the comments */
NXP_SaveKB( myKB, "mykb.ckb", NXP_MODE_COMPILED |
NXP_MODE_COMMENTS );
```

Saving the internal KB "temporary.kb" containing all the atoms created dynamically (after a Retrieve or a CreateObject for instance):

```
NXP_SaveKB( 1, "newObjects.tkb", 0);
```

See Also

NXP_LoadKB	Loads a knowledge base file.
NXP_AINFO_KBID	Gets the id of a KB to which an atom belongs.

NXP_SendMessage

Purpose

This call is updating the engine current evaluation stack with a message to an addressee with eventually an array of parameters. The SendMessage action is executed when the inference engine hits this action in the stack.

C Format

The C format is as follows:

int NXP_SendMessage(*messageName***,** *addresseeId***,** *args***,** *argTypePtr***,** *num***,** *strat***);**

Arguments

The following list shows the valid arguments:

```
Str          messageName;
AtomId       addresseeId;
VoidPtr      *args;
int          *argTypePtr;
```

```
int          num;
int          strat;
```

messageName is a string containing the name of the method to trigger.

*addresseeI*d is the atom Id to which the message is sent.

args is an array of pointers to the arguments (which can be a string, a float, an integer, an atomid, and so forth).

argTypePtr is an array of the types of the arguments. The types are one of the following: NXP_DESC_STR, NXP_DESC_INT, NXP_DESC_ATOM, NXP_DESC_FLOAT, NXP_DESC_DOUBLE, NXP_DESC_LONG, NXP_DESC_DATE and NXP_DESC_TIME.

num number of arguments

strat is the placement of the SendMessage in the current evaluation stack of the inference engine. The placements are one of the following: NXP_CTRL_ATTOP, and NXP_CTRL_ATBOTTOM. The default is NXP_CTRL_ATTOP.

Return Codes

NXP_SendMessage returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	messageName was invalid (empty or non existing).
NXP_ERR_INVARG2	addresseeId was invalid or the wrong type - not a slot, an object, a class or a property.
NXP_ERR_INVARG7	strat1 was invalid.

Examples

The following example

```
/* Example of NXP_SendMessage */
       AtomId atom;
       int          argInt = 5;
       VoidPtrargs[100];
       int          types[100] = {NXP_DESC_INT};
       NXP_GetAtomId("c.p", &atom, NXP_ATYPE_SLOT);
       NXP_GetAtomId("o.p", &atom, NXP_ATYPE_SLOT);
       args[0] = &argInt;
       NXP_SendMessage("factorial", atom, (VoidPtr *)args,
types, 1,
                    NXP_CTRL_ATTOP);
...
```

See Also

`NXP_SendMessageArray`	Sends a message to a list.

NXP_SendMessageArray

Purpose

This call is updating the engine current evaluation stack with a message to several addressees with eventually an array of parameters. The SendMessage action is executed when the inference engine hits this action in the stack.

C Format

The C format is as follows:

int NXP_SendMessageArray(*messageName***,** *addresseesIdsArray***,** *numAtoms***,** *args***,** *argTypePtr***,** *num***,** *strat***);**

Arguments

The following list shows the valid arguments:

```
Str          messageName;
AtomId *addresseeIdsArray;
int          numAtoms;
VoidPtr*args;
int          *argTypePtr;
int          num;
int          strat;
```

messageName is a string containing the name of the method to trigger.

*addresseeId*s is an array to the atom Ids of the addressees to which the message is sent.

num Atoms number of atoms in addresseeIds array.

args is a C array of pointers to the arguments to be sent with the message (which can be a string, a float, an integer, an atomid, and so forth).

ArgTypePtr is an array of the types of the arguments. The types are one of the following: NXP_DESC_STR, NXP_DESC_INT, NXP_DESC_ATOM, NXP_DESC_FLOAT, NXP_DESC_DOUBLE, NXP_DESC_LONG, NXP_DESC_DATE and NXP_DESC_TIME.

num number of arguments.

strat is the placement of the SendMessage in the current evaluation stack of the inference engine. The placements are one of the following: NXP_CTRL_ATTOP, and NXP_CTRL_ATBOTTOM. The default is NXP_CTRL_ATTOP.

Return Codes

NXP_VolunteerArray returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error

immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	messageName was invalid - empty or message doesn't exist
NXP_ERR_INVARG2	The list of Atoms is NULL or there is at least one atom of wrong type -not a slot, object, class or property.
NXP_ERR_INVARG3	The first atomid or atom name in the list pointed to by atoms was invalid, or the wrong type.
NXP_ERR_INVARG7	The strategy pointed to by strats is invalid.

Examples

The following example is sending the message to compute a series of factorials, the list of factorials to be computed being passed as an argument of the Execute.

```
/* Example of NXP_SendMessageArray */
       int          MyExecute( char *theStr, int nAtoms, AtomId, *theAtoms)
{
       int          argInt = 5;
       VoidPtrargs[100];
       int          types[100] = {NXP_DESC_INT};
       args[0] = &argInt;
       NXP_SendMessageArray("factorial", theAtoms, nAtoms,
                    (VoidPtr*)args, types, 1, NXP_CTRL_ATTOP);
...
}
```

See Also

NXP_SendMessage	Sends a message to a single atom.

NXP_SetAtomInfo

Purpose

NXP_SetAtomInfo provides some control over knowledge bases allowing you to change information associated with individual atoms or entire knowledge bases. This function is the opposite of NXP_GetAtomInfo (although all possible codes are not implemented yet).

See Chapter Five, "The NXP_SetAtomInfo Routine".

C Format

The C format is as follows:

int NXP_SetAtomInfo(*atom***,** *code***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments:

```
AtomId       atom;
int          code;
AtomId       optAtom;
```

```
int          optInt;
int          desc;
Str          ptr;
```

atom is the Id of an atom or a knowledge base.

optAtom, *optInt*, d*esc*, *ptr* are additional parameters whose meaning depends on the value of code.

code: the codes used by NXP_SetAtomInfo are categorized as follows:

Controlling the knowledge bases

NXP_SAINFO_CURRENTKB	This sets the current (or "default") knowledge base.
NXP_SAINFO_DISABLESAVEKB	This disables the saving of knowledge bases from the Application Programming Interface.
NXP_SAINFO_INKB	This sets the knowledge base that an atom belongs to.

Controlling the knowledge bases

NXP_SAINFO_MERGEKB	This merges two knowledge bases into one.

Setting/unsetting break points

NXP_SAINFO_AGDVBREAK	This sets / unsets agenda break points on hypotheses.
NXP_SAINFO_INFBREAK	This sets / unsets inference break points on atoms.

Changing permanent/temporary links

NXP_SAINFO_PERMLINK	This changes the links of an atom to permanent.
NXP_SAINFO_PERMLINKKB	This changes all links in a knowledge base to permanent.

These codes are described in detail in Chapter Five, “The NXP_SetAtomInfo Routine.”

NXP_SetClientData

Purpose

NXP_SetClientData allows you to associate a longword of information with any Rules Element atom. This information can be retrieved later with the NXP_GetAtomInfo call and the NXP_AINFO_CLIENTDATA code. For example, if you want to interface the Rules Element with a graphic package you can associate pointers to a data structure representing a graphic object with Rules Element objects and use this information in an If Change Execute routine to update the graphic object as the Rules Element value changes.

C Format

The C format is as follows:

int NXP_SetClientData(*theAtom***,** *theInfo***);**

Arguments

The following list shows the valid arguments:

```
AtomId        theAtom;
unsigned long theInfo;
```

theAtom is the id of a class, an object, a slot, a property, a rule, a condition or an action (RHS, Order of Sources, If Change).

theInfo can take any 4 byte value.

Notes

The ClientData longword is initialized to 0 when an atom is created.

Return Codes

NXP_SetClientData returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is invalid.
NXP_ERR_NOERR	Call was successful.

Examples

Assume you are using a graphic package with routines: CreateDial, UpdateDial, ...

```
AtomId  slot;
Dial    dial;  /* longword variable of your application */

/* create dial with your application */
dial = CreateDial(10, 10, 50, 50, "tank1");

/* associate dial with slot tank1.pressure */
NXP_GetAtomId("tank1.pressure, &slot, NXP_ATYPE_SLOT);
NXP_SetClientData(slot, (unsigned long)dial);

/* Assume that the If Change of tank1.pressure is the action:
 *           Execute "UpdateInterface" @ATOMID=SELF.pressure;
 *
 * Here is how you could code the UpdateInterface routine
 */

int     UpdateInterface(str, natoms, atoms)
Str     str;
int     natoms;
AtomId C_FAR * atoms;
{
       DialPtr dial = 0;
       double dval;

       if(natoms != 1) {  /* error...  */   return 0; }
```

```
       /* retrieve dial associated with atoms[0].
        *     Note: sizeof(unsigned long) == sizeof(DialPtr) == 4 bytes.
        */
       NXP_GetAtomInfo(atoms[0], NXP_AINFO_CLIENTDATA, (AtomId)0, 0,
NXP_DESC_LONG, (Str)&dial, 0);
       if(dial == 0) {  /* error...  */   return 0; }

       /* Get the double value of the dial slot */
       NXP_DOUBLEVAL(atoms[0], &dval);

       /* Update the dial in your application */
       UpdateDial(dial, dval);

       return 1;/* Execute routine was successful */
}
```

NXP_SetData

Purpose

NXP_SetData sends information to the interface. This call is valid only when the development system is up and running because the runtime library doesn't have any interface.

C Format

The C format is as follows:

int NXP_SetData (*winId***,** *ctrlId***,** *index***,** *thePtr***);**

Arguments

The following list shows the valid arguments:

```
Int          winId;
Int32        ctrlId;
Int32        index;
Str          thePtr;
```

winId is the id of the window to which data is sent.

ctrlId is an integer describing which sub part of the window is involved.

index is an additional integer whose meaning depends on winId and ctrlId.

thePtr is a pointer to a memory location where the information is stored.

NXP_SetData can be used to write text in the following windows:

Window	Code
Transcript	winId = NXP_WIN_TRAN
Current rule	winId = NXP_WIN_RULE
Current hypothesis	winId = NXP_WIN_HYPO
Conclusions	winId = NXP_WIN_CONC
Banner (messages)	winId = NXP_WIN_BANNER

Other values of winId are reserved for Neuron Data's internal use.

Banner messages are messages which appear on the screen while a knowledge base is being loaded, and disappear automatically, as opposed

to dialog windows where the user must click OK to have the window disappear.

For the first four windows, Transcript, Rule, Hypothesis, Conclusions, ctrlId is ignored. If thePtr is NULL, the previous content of the window is cleared. Otherwise thePtr must point to a null terminated string. The third parameter index controls where the string will be written. If index is -1 or -2, the string will be written at the end of the existing text; if 0, it will be inserted at the beginning of the existing text. If the window is write disabled, the text will not be written unless index equals -2.

In the case of the banner window, index is ignored. If thePtr is NULL, the banner is removed from the screen. Otherwise, thePtr must point to a null terminated string. ctrlId specifies where the string will be written. It can take the following values:

Code	Description
NXP_CELL_COL1:	Top line of the banner window
NXP_CELL_COL2:	Left part of the bottom line
NXP_CELL_COL3:	Right part of the bottom line.

If the banner is not open and thePtr is not NULL, the call will cause the banner window to appear on the screen.

Return Codes

NXP_SetData returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	winId is invalid.
NXP_ERR_INVARG2	ctrlId is invalid
NXP_ERR_INVARG3	index is invalid
NXP_ERR_INVARG4	thePtr is NULL.
NXP_ERR_NOERR	Call was successful.

Examples

Example of writing text to the Transcript:

```
/* message will be written at the end of transcript
 * only if the transcript is write enabled
/*
NXP_SetData(NXP_WIN_TRAN, NXP_CELL_NONE, -1, "Testing NXP_SetData with -1");

/* message will be written at the end of transcript
 * even if transcript is not write enabled
/*
NXP_SetData(NXP_WIN_TRAN, NXP_CELL_NONE, -2, "Testing NXP_SetData with -2");
```

Example of controlling the banner:

```
/* bring up banner with message */
NXP_SetData(NXP_WIN_BANNER, NXP_CELL_COL1, 0,
                          "Computing Fast Fourier Transform");
```

```
ComputeFFT(); /* will take a few seconds */

/* remove banner */
NXP_SetData(NXP_WIN_BANNER, NXP_CELL_NONE, 0, (Str)0);
```

See Also

`NXP_SetHandler` / `NXP_PROC_GETDATA`	Installs the GetData handler.
`NXP_SetHandler` / `NXP_PROC_SETDATA`	Installs the SetData handler.

NXP_SetHandler

Purpose

NXP_SetHandler allows you to install user-written procedures to be called by the Rules Element (call out). They will either replace built-in procedures, like the Question handler, or become new Execute procedures. These procedures are usually installed with NXP_SetHandler during the initialization of the application so that they can be called later by the inference engine. The most frequent use of the NXP_SetHandler call is with the NXP_PROC_EXECUTE code. Procedures installed with this code can be called from Execute statements in rules or methods.

NXP_SetHandler can also install the procedures which allow the Rules Element kernel to communicate with its interface. The most obvious example is the Question procedure: you can use a user-defined procedure to prompt the user when a question arises during a session. You could for example prompt an operator on a remote terminal or display the question in a custom graphic environment.

The Rules Element uses some built-in procedures to communicate with its interface (i.e. a default question procedure which prompts in the session control window). When you install a custom handler other than an Execute with NXP_SetHandler, your handler will be called instead of the built-in procedure. If your handler returns FALSE (0), the default built-in procedure will be called after your handler. Otherwise, your handler should return TRUE (1) and it will completely override the built-in procedure. In the case of an Execute handler, the returned value is used only if the Execute is in the LHS of a rule (conditions). The condition is FALSE if your routine returns 0, and TRUE otherwise.

See also NXP_SetHandler2 for additional information. It is similar to NXP_SetHandler except that it takes two additional arguments that allow you to store your own information and get it back when the Rules Element calls your procedure.

Note: You cannot install the same type of handler twice using NXP_SetHandler and NXP_SetHandler2. There can be only one Question handler, one Alert handler, etc (except for Execute handlers).

C Format

The C format is as follows:

int NXP_SetHandler(*code***,** *theProc***,** *theName***);**

Arguments

The following list shows the valid arguments:

```
Int                 code;
NxpIProc            theProc;
Str                 theName;
```

`NxpIProc` type is defined as follows (see file `nxpdef.h`) :

```
typedefi            nt (C_FAR *NxpIProc)();
```

code describes which procedure will be trapped. It is one of the following:

Code	Description
NXP_PROC_ALERT	Alert box brought on the screen.
NXP_PROC_APROPOS	Show action.
NXP_PROC_CANCEL	Interrupt handler.
NXP_PROC_ENDOFSESSION	Called upon the end of a session.
NXP_PROC_EXECUTE	"Execute" routine.
NXP_PROC_FORMINPUT	Get control before a form is open.
NXP_PROC_GETDATA	Gets data from the interface.
NXP_PROC_GETSTATUS	Checks the availability of an interface.
NXP_PROC_MEMEXIT	Exits when no more memory is available.
NXP_PROC_NOTIFY	Notifies the interface when something changes in the working memory.
NXP_PROC_PASSWORD	Prompts for an encrypted knowledge base password.
NXP_PROC_POLLING	Polling procedure called at each inference engine cycle.
NXP_PROC_QUESTION	Question asked by the engine.
NXP_PROC_QUIT	Called when the Rules Element is going to exit.
NXP_PROC_SETDATA	Sends data to the interface.
NXP_PROC_VALIDATE	Supplies your data validation function from the meta-slot editor.
NXP_PROC_VOLVALIDATE	Supplies your data validation function. These codes and the arguments received by the handlers are described in detail hereafter.

These codes and the arguments received by the handlers are described in detail in the following sections.

theProc is the address of the procedure which will be called. If theProc is NULL, any handler previously installed with the same code is removed and the Rules Element kernel will call its built-in procedure.

Note: The handler theProc should process the exact same parameters as the default Rules Element function and always return an integer. It returns FALSE if the default Rules Element function should be called anyway, and TRUE if the event was entirely handled by the handler.

theName is used to specify the name of Execute handlers (first argument of the Execute statements inside rules or methods). It is ignored if code is not NXP_PROC_EXECUTE.

Notes

Use NXP_GetHandler to get back a handler previously set by NXP_SetHandler.

If NXP_SetHandler is called twice with the same code, only the last entry will be kept (except for Execute handlers which are differentiated by theName).

Macintosh Users: The handler routine theProc will receive the extra argument ExtInfo just like external routines the Rules Element calls. For more information, see the Macintosh API manual.

Examples

The following example illustrates handler installations at the beginning of an application to modify the Rules Element's interface:

```
/* first call to the Rules Element */
NXP_Control(NXP_CTRL_INIT);
...
/* Set up a custom Question handler */
NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)myQuestion, (Str)0);

/* Set up a custom Apropos handler */
NXP_SetHandler(NXP_PROC_APROPOS, (NxpIProc)myShow, (Str)0);

/* Set up a custom Alert handler */
NXP_SetHandler(NXP_PROC_ALERT, (NxpIProc)myAlert, (Str)0);

/* Set up 2 Execute handlers */
NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)doThisExec, "doThis");
NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)doThatExec, "doThat");
```

`myQuestion`, `myAlert`, `myShow`, `doThis`, and `doThat` represent the addresses of custom procedures that you must define elsewhere in your application. As long as no other NXP_SetHandler calls are made, the Rules Element uses its default handlers (default behavior of the interface).

Although it is generally the case, it is not required to install handlers during application initialization. It can be done at any point before the time they are actually used (before the beginning of a session, for instance). You can also control the installation of handlers through Execute procedures to be called from the rules.

For instance, while using the Development System, you may want to deinstall your Question handler at some point during the session and reinstall it later (maybe because your Question handler does not know how to process some questions and you want to take advantage of the default Question window of the Rules Element's interface. You cannot do this if you are just using the runtime library since there is no "default" question handler). The following examples illustrate this:

Step 1: Set up two Execute procedures, one to install your Question handler and one to deinstall it:

```
NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)InstallQuestion,
                        "InstallQuestion");
NXP_SetHandler(NXP_PROC_EXECUTE, (NxpIProc)DeinstallQuestion,
                        "DeinstallQuestion");

InstallQuestion()
{
```

```
        NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)myQuestion,
(Str)0);
        return TRUE;
}

DeinstallQuestion()
{
        NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)0, (Str)0);
        return TRUE;
}
```

Step 2: Call "EXECUTE InstallQuestion" in a rule of a meta-slot's method whenever you want to install your custom question.

Step 3: Call "EXECUTE DeinstallQuestion" in a rule of a meta-slot's method whenever you want to remove your custom question and use the Rules Element's default question.

Note: You could also use only one Execute procedure, with a string or atom argument as a flag for Install / Deinstall. See NXP_PROC_EXECUTE for more information.

Another way to handle the previous example is to leave your Question handler installed and simply have it return FALSE when it does not know how to process a question. As explained previously, a user handler returning FALSE forces the Rules Element to immediately call its default handler.

You can use the following technique to temporarily replace a handler with another one:

```
NxpIProc   oldQuestion, newQuestion;

/* Get the address of the current Question handler */
NXP_GetHandler(NXP_PROC_QUESTION, (NxpIProc C_FAR *)&oldQuestion, (Str)0);

/* Change to a new handler */
NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)newQuestion, (Str)0);

/* ...  later on in the application, restore the old handler */
NXP_SetHandler(NXP_PROC_QUESTION, (NxpIProc)oldQuestion, (Str)0);
```

See the individual SetHandler calls and the chapter "Primer" for more examples.

See Also

`NXP_GetHandler`	Get handlers set with NXP_SetHandler.
`NXP_GetHandler2`	Get handlers set with NXP_SetHandler2.
`NXP_SetHandler2`	Install handlers with additional arguments.

NXP_SetHandler2

Purpose

NXP_SetHandler2 allows you to install user-written procedures. It has the same functionalities as NXP_SetHandler except it takes two additional arguments, an intteger and an unsigned long integer. The int is kept in memory as the "type" of handler. The unsigned long is passed back to the handler unchanged by the Rules Element.

Each SetHandler has its own SetHandler2 equivalent. You cannot install the same type of handler twice using NXP_SetHandler and NXP_SetHandler2. There can only be one Question handler, one Alert handler, and so on.

C Format

The C format is as follows.

int NXP_SetHandler2(*code***,** *theProc***,** *theName***,** *type***,** *arg***);**

Arguments

The following list shows the valid arguments.

```
int          code;
NxpIProc     theProc;
Str          theName;
int          type;
ULong        arg;
```

code describes which procedure will be trapped. It is one of the following:

Code	Description
NXP_PROC_ALERT	Alert box brought on the screen.
NXP_PROC_APROPOS	Show action.
NXP_PROC_CANCEL	Interrupt handler.
NXP_PROC_ENDOFSESSION	Called upon the end of a session.
NXP_PROC_EXECUTE	"Execute" routine.
NXP_PROC_FORMINPUT	Get control before a form is open.
NXP_PROC_GETDATA	Gets data from the interface.
NXP_PROC_GETSTATUS	Checks the availability of an interface.
NXP_PROC_MEMEXIT	Exits when no more memory is available.
NXP_PROC_NOTIFY	Notifies the interface when something changes in the working memory.
NXP_PROC_PASSWORD	Prompts for an encrypted knowledge base password.
NXP_PROC_POLLING	Polling procedure called at each inference engine cycle.
NXP_PROC_QUESTION	Question asked by the engine.
NXP_PROC_QUIT	Called when the Rules Element is going to exit.
NXP_PROC_SETDATA	Sends data to the interface.
NXP_PROC_VALIDATE	Supplies your data validation function from the meta-slot editor.
NXP_PROC_VOLVALIDATE	Supplies your data validation function.

These codes and the arguments received by the handlers are described in detail in the following sections.

theProc is the address of the procedure which will be called. If theProc is NULL, any handler previously installed with the same code is removed and the Rules Element kernel will call its built-in procedure.

theName is used to specify the name of Execute handlers (first argument of the Execute statements inside rules or methods). It is ignored if code is not NXP_PROC_EXECUTE.

type is the type of handler. This information is kept in memory and is not passed to the handler. It is returned by NXP_GetHandler2. Usually, your application won't need this information so use the predefined constant NXP_HDLTYPE_USER = 0x0100. Codes from 0x0000 to 0x00FF are reserved for the Rules Element's internal use, for when handlers are installed by the Rules Element itself (reserved constants NXP_HDLTYPE_xxx are defined in the `nxpdef.h` file).

arg is not interpreted by the Rules Element. It can contain any information your application wishes (a pointer to a buffer, a pointer to a custom structure, etc.).

Notes

A type information can be useful if you wish to display handler information in a client / server application. It is also useful if you are installing handlers in several languages (C, Pascal, Fortran, etc.) and need to know the language when the handlers are called.

theProc has the same interface as the corresponding handler installed with NXP_SetHandler, except for the extra parameter arg passed as first argument. theProc should return 1 to prevent the Rules Element from calling its default handler afterward, or 0 otherwise. See the following sections for a description of each type of handler, installed with NXP_SetHandler or NXP_SetHandler2.

Return Codes

NXP_SetHandler2 returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theCode is an invalid number.

Examples

Example of installing a Question handler with NXP_SetHandler2. myQuestion is the procedure that will be called during a question. It receives the extra argument arg unmodified by the Rules Element.

```
int    myQuestion(arg, atom, str)
ULong         arg;
AtomId        atom;
Str           str;
{
       ...
}

NXP_SetHandler2(NXP_PROC_QUESTION, myQuestion, (Str)NULL,
                            NXP_HDLTYPE_USER, arg)
```

Example of installing an Execute handler with NXP_SetHandler2. We pass the pointer myStructPtr to NXP_SetHandler2 so that the Execute routine receives the same argument when it is called by the Rules Element.

```
int    doThisExec(myStructPtr, theStr, nAtoms, theAtoms)
unsigned long       myStructPtr;
Str                 theStr;
```

```
int                 nAtoms;
AtomId C_FAR*theAtoms;
{
      ...
}

NXP_SetHandler2(NXP_PROC_EXECUTE, doThisExec, "doThis",
                    NXP_HDLTYPE_USER, myStructPtr)
```

See Also

NXP_GetHandler	Get handlers set with NXP_SetHandler.
NXP_GetHandler2	Get handlers set with NXP_SetHandler2.
NXP_SetHandler	Install handlers without additional arguments.

NXP_SetHandler (2) / NXP_PROC_ALERT

Purpose

The Alert handler is called each time an alert box is brought on the screen. It can be either an error message with an OK button or a dialog box with 2 or 3 choices (Ok-Cancel or Yes-No-Cancel). The Alert handler is very useful for reporting compilation or runtime errors while using the Rules Element library which doesn't have a default interface. It is called with the arguments described below.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpAlert (*code***,** *theStr***,** *ret***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpAlert (*arg***,** *code***,** *theStr***,** *ret***);**

Arguments

The following list shows the valid arguments:

```
ULong         arg;
int           code;
Str           theStr;
IntPtr        ret;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

code is one of the following:

Code	Description
NXP_ALRT_OK	The alert box called just has an OK button.
NXP_ALRT_OKCANCEL	The alert box has an OK and a CANCEL button.
NXP_ALRT_YESNOCANCEL	The alert box has YES, NO and CANCEL buttons.

theStr is the message normally displayed in the text part of the dialog box.

ret is a pointer to an integer where the code of the button clicked on should be returned. The possible return values are:

- NXP_RET_CANCEL, NXP_RET_NO, NXP_RET_OK, NXP_RET_YES

For alerts with just the OK button the value returned in ret is not used. For alerts with 2 or 3 buttons this value will be used by the Rules Element to take the proper action after the alert.

Return Codes

Your Alert handler should return FALSE if you want the Rules Element's default alert box to be displayed afterwards. It should return TRUE otherwise.
(This only matters in the Development System since the Runtime library doesn't have any default interface).

Examples

Since the runtime library doesn't have any interface you should install an Alert handler to report compilation or runtime errors:

```
            /* Example of Alert handler */
int    myAlert(code, theStr, ret)
int           code;
Str           theStr;
IntPtr ret;
{
       if(code == NXP_ALRT_OK) {
              /* Print message to console and doesn't stop */
              printf("ALERT: %s\n", theStr);
              *ret = NXP_RET_OK;/* not really necessary...  */

              /* Note: On a graphical window system you could open
                     your own modal dialog box to display the message */
       }
       else if(code == NXP_ALRT_OKCANCEL) {
              /* Ask the user to enter OK or CANCEL
                     to answer the question */
              ...
              *ret = ...;
       }
       else if(code == NXP_ALRT_YESNOCANCEL) {
              /* Ask the user to enter OK or CANCEL
                     to answer the question */
              ...
              *ret = ...;
       }
       return TRUE;/* avoid the Rules Element's default Alert */
}

/* Installation of the handler */
NXP_SetHandler(NXP_PROC_ALERT, (NxpIProc)myAlert, (Str)0);
```

The Alert handler traps all the error messages so you can also use it to filter some of these messages and let your application deal with the error. For instance your application could detect a "file not found..." error and take further steps to fix the problem or find a work-around.

NXP_SetHandler (2) / NXP_PROC_APROPOS

Purpose

This handler is called each time an Apropos call is done (Show operator in a condition or an action) . It is called with the arguments described below.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpApropos(*fileName***,** *wait***,** *keep***,** *or***,** *ext***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpApropos(*arg***,** *fileName***,** *wait***,** *keep***,** *or***,** *ext***);**

Arguments

The following list shows the valid arguments.

```
ULong               arg;
Str                 fileName;
int                 wait;
int                 keep;
NXP_PtRec C_FAR*    or;
NXP_PtRec C_FAR*    ext;
```

where NXP_PtRec is a point record:

```
typedef struct NXP_PtRec {
  int         x;
  int         y;
} NXP_PtRec;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

fileName is the name of the file to be displayed.

wait is TRUE if the called function should wait for user input before returning (corresponds to the wait option in the Show dialog box).

keep is TRUE if the displayed file should remain on the screen (corresponds to the keep option in the Show dialog box).

or is a pointer to a NXP_PtRec structure describing the top left point where the drawing should be positioned (corresponds to the left and top parameters in the Show dialog box). or is NULL if the file should be displayed at the default location.

ext is a pointer to a NXP_PtRec structure representing the diagonal extent of the image (corresponds to the width and height parameters in the Show dialog box). ext is NULL if the default size should be used.

Return Codes

Your Apropos handler should return FALSE if you want the Rules Element's default Apropos function to be used afterwards. It should return TRUE otherwise.
(This only matters for the Development System since the Runtime library doesn't have any default interface).

Notes

The Rules Element's default Apropos function displays ascii text and graphic files (graphic format depends on your platform). You can use the Apropos handler to add support for other formats.

wait , keep , or and ext are options used in the development system interface. You don't have to use them in your own Apropos handler. or and ext coordinates are in pixels.

Examples

Simple Apropos handler that prints the filename to the console:

```
int myApropos(fileName, wait, keep, or, ext)
Str                 fileName;
int                 wait;
int                 keep;
NXP_PtRec C_FAR *or;
NXP_PtRec C_FAR *ext;
{
       printf("Show file %s\n", fileName);
       return TRUE;
}

/* Installation of the handler */
NXP_SetHandler(NXP_PROC_APROPOS, (NxpIProc)myApropos, (Str)0);
```

NXP_SetHandler (2) / NXP_PROC_CANCEL

Purpose

This handler is called during time consuming operations of the Rules Element. Its purpose is to allow you to cancel the current operation: loading or saving a knowledge base, performing a database access (Retrieve or Write), doing a full expansion of the rule or object network.

If no handler is installed the development version uses its default procedure which traps Control-\ on Unix and VAX, Control-Alt on PC and Command-<period> on Macintosh, and brings up an alert box where the user must confirm the cancellation (the Runtime library doesn't have a default Cancel procedure). By installing your own handler you can customize this procedure, change the keyboard binding or avoid the alert box.

C Format

The C format is as follows if the handler is installed with `NXP_SetHandler`:

int NxpCancel(*prompt***,** *retVal***);**

or as follows if it is installed with `NXP_SetHandler2`:

int NxpCancel(*arg***,** *prompt***,** *retVal***);**

Arguments

The following list shows the valid arguments.

```
ULong         arg;
int           prompt;
IntPtr        retval;
```

*arg i*s passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

prompt is TRUE if the Rules Element expects a confirmation dialog.

retval is a pointer to an integer where TRUE or FALSE should be returned. Setting retval to TRUE tells the Rules Element to cancel the current operation. Setting retval to FALSE allows the Rules Element to continue where it was stopped.

Return Codes

Your Cancel handler should always return TRUE otherwise the Rules Element's default Cancel function will be used afterwards.

Notes

A Cancel handler is called by the Rules Element only during a limited set of operations:

- Load and Save KB.
- Database access (grouped Retrieve and Write).
- full extension of the rule or object network.

You cannot use a Cancel handler to interrupt the inference engine during a session. For that you must install a Polling handler and call NXP_Control with NXP_CTRL_STOPSESSION.

Examples

Simple Cancel handler detecting when the mouse is pressed:

```
int myCancel(prompt, retval)
Str           prompt;
IntPtr        retval;
{
       /*
        * Button() is the Mac toolbox call to check if the
        * mouse is pressed.  Change it for other platforms
        */
#ifdef MAC
       if (Button())*retval = TRUE;
       else                *retval = FALSE;
#endif
       return TRUE;
}

/* Installation of the handler */
NXP_SetHandler(NXP_PROC_CANCEL, (NxpIProc)myCancel, (Str)0);
```

See Also

`NXP_PROC_POLLING`	Polling handler.
`NXP_Control / NXP_CTRL_STOPSESSION`	Stops a session.

NXP_SetHandler (2) / NXP_PROC_ENDOFSESSION

Purpose

.This handler is called at the end of a session, when there is nothing left to process.

The end of a session does not signify an exit from the Rules Element; you can still perform actions and call the Rules Element from your application when the inference session is over (particularly, you can investigate the Rules Element's working memory with NXP_GetAtomInfo to get results). Use the NXP_PROC_QUIT handler if you want to be notified when the Rules Element is going to exit.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int MyEndOfSession();

or as follows if it is installed with NXP_SetHandler2:

int MyEndOfSession(*arg***);**

Arguments

The following list shows the valid arguments.

`unsigned long arg;`

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

Examples

The following example illustrates how to get the value of hypothesis H when the session is over and display it in the Transcript window:

```
int   MyEndOfSession()
{
       AtomId  hypoId;
       Char    hypoValue[20];

       /* Get AtomId of hypothesis "H" */
       NXP_GetAtomId("H", &hypoId, NXP_ATYPE_HYPO);

       /* Get value as a string, "TRUE", "FALSE", "UNKNOWN", or "NOTKNOWN" */
       NXP_GetAtomInfo(hypoId, NXP_AINFO_VALUE, (AtomId)0, 0,
                    NXP_DESC_STR, hypoValue, 20);

       /* Send this string to the Transcript */
       NXP_SetData(NXP_WIN_TRAN, 0, -1, hypoValue);
}
```

You must install the EndOfSession handler with the following call (the last argument, name, is ignored for codes other than NXP_PROC_EXECUTE):

```
NXP_SetHandler(NXP_PROC_ENDOFSESSION, (NxpIProc)MyEndOfSession, (Str)0);
```

See Also

`NXP_Control / NXP_CTRL_KNOWCESS`	Start a session.
`NXP_SetHandler / NXP_PROC_QUIT`	Be notified when the Rules Element exits.

NXP_SetHandler (2) / NXP_PROC_EXECUTE

Purpose

Execute handlers are called when the inference engine processes Execute statements placed in rules or methods. They are called with the string and atom arguments described below.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpExecute (*theStr***,** *nAtoms***,** *theAtoms***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpExecute (*arg***,** *theStr***,** *nAtoms***,** *theAtoms***);**

Arguments

The following list shows the valid arguments.

```
ULong               arg;
Str                 theStr;
int                 nAtoms;
AtomId C_FAR*       theAtoms;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

The other arguments are directly related to the 2nd argument of the Execute operator. The format of an Execute statement is the following:

```
Execute proc_name arguments_description
```

proc_name must match the name under which the Execute handler was registered with the NXP_SetHandler or NXP_SetHandler2 call (theName argument of NXP_SetHandler).

The *arguments_description* describes the arguments which will be passed to the Execute handler. Two types of arguments can be specified: a string and / or a list of atoms. When the knowledge base is edited with the rule or meta-slot editors, this information is entered in a special dialog. When the text of the knowledge base is edited directly, this information is prefixed by two keywords: @STRING for the string and @ATOMID for the list of atoms.

The first argument (t*heSt*r) passed to the execute handler corresponds to the @STRING string. If a string was specified with the @STRING keyword in the Execute statement, theStr points to this string (null-terminated buffer of characters), otherwise, theStr is a NULL pointer.

nAtoms and *theAtoms* correspond to the @ATOMID atom list. The atom list is passed as an array of AtomIds to the Execute handler. nAtoms is the number of elements in the array and theAtoms is the pointer to the first

element in the array. If the second argument of the Execute statement did not specify an @ATOMID list, nAtoms is 0 and theAtoms is a NULL pointer.

Return values

When the Execute is called from the Left Hand Side of a rule, the value returned by the Execute handler is used to set the Execute condition to TRUE (1) or FALSE (0). If the Execute is called from the RHS of a rule or from methods, its return value is ignored by the Rules Element.

Notes

The atoms passed in the list theAtoms can have different types (class, object, slot, ...) and that the Value slot of an object must be passed as "object.value" in the interface. For instance if N is an integer the statement:

```
Execute  myFunction  @ATOMID=N;
```

will pass the atom id of the object N and not the slot N.value to the handler! One must write explicitly N.Value. In the same way one should write H.Value for a hypothesis H which has a boolean slot Value (your handler can check that common mistake, see the example below). Value is a special property in the Rules Element; in most places it can be ignored because no confusion is possible, in the Execute interface it is necessary to distinguish the object name from its value slot.

Interpretations in the arguments @STRING and @ATOMID are expanded before the Execute handler is called. For instance if the string argument is "The color of the car is @V(car.color)" and the value of car.color is "red" when the Execute handler is called, theStr will contain "The color of the car is red". If the atoms argument contains `\object.prop\.color` and the value of object.prop is "car", then the atom id of car.color will be passed to the handler.

Examples

(The following examples report errors with the printf() function. While running in the development system you could use NXP_SetData() to write the messages into the Transcript).

This is an example of an Execute handler accepting only one atom argument. It checks the type and number of arguments and gets the Value slot of objects.

```
int           myExecute(theStr, nAtoms, theAtoms)
Str           theStr;
int           nAtoms;
AtomId C_FAR* theAtoms;
{
       int          type;
       AtomId theSlot = theAtoms[0];/* first atom */
       AtomId valueProp;                /* Value property */
       AtomId theObject;

       if(nAtom != 1 || theStr != 0) {
              printf("Wrong arguments in myExecute: pass only one atom"
                     " and no string");
              return FALSE;
       }

       /* check the type of atom */
```

```
        ret = NXP_GETINTINFO(theSlot, NXP_AINFO_TYPE, &type);
        if(ret == 0) {
                printf("Error in myExecute while getting type of 1st atom,
                        NXP_Error = %d \n", NXP_Error());
                return FALSE;
        }
        /* must mask the type to get only the last bits */
        switch(type & NXP_ATYPE_MASK) {

        case NXP_ATYPE_SLOT:        /* good type */
                break;

        case NXP_ATYPE_CLASS:       /* user forgot the ".value" */
        case NXP_ATYPE_OBJECT:      /* we'll add for him! */
                theObject = theSlot;
                printf("Warning in myExecute, argument is a class or an
                                object.  Taking Value slot instead"\n);
                /* get the Id of the special Value property */
                NXP_GetAtomId("Value", &valueProp, NXP_ATYPE_PROP);
                /* get the Id of slot object.value or class.value */
                NXP_GetAtomInfo(theObject, NXP_AINFO_SLOT, valueProp,
                        (AtomId)0, NXP_DESC_ATOM, (Str)&theSlot, 0);
                break;

        default:
                printf("Error in myExecute, argument has wrong type = %x \n",
                          type);
                return FALSE;
        }

        /* perform some custom code */
        ....

        return TRUE;  /* Execute successful */
}
```

NXP_SetHandler (2) / NXP_PROC_GETDATA

Purpose

This handler is called when the Rules Element gets data from a window. You need to set a GetData handler only if you are using the NXP_Compile call or the NXP_Edit call. Only advanced Rules Element programmers should use GetData. See Chapter Six, “Edit Functions” for details.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpGetData(*winId***,** *ctrlId***,** *index***,** *thePtr***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpGetData(*arg***,** *winId***,** *ctrlId***,** *index***,** *thePtr***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
Int          winId;
Int32        ctrlId;
```

```
Int32         index;
Str           thePtr;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

winId is the id of the window from which data is requested.

ctrlId is an integer describing which sub part of the window is involved.

index is an additional integer whose meaning depends on winId and ctrlId.

thePtr is a pointer to a memory location where the information should be returned.

Return Values

The GetData handler should return TRUE if it processed the call. It should return FALSE if you want the default GetData function to be called afterward, for instance if winId is the id of a window you don't want to process.

NXP_SetHandler (2) / NXP_PROC_GETSTATUS

Purpose

This GetStatus handler is called when the Rules Element checks the availability of an interface. For example when starting or resuming a session it calls the GetStatus handler to find out whether the Transcript is enabled or not so that it can avoid formatting the strings when the Transcript is disabled.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpGetStatus(*winId***,** *code***,** *ret***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpGetStatus(*arg***,** *winId***,** *code***,** *ret***);**

Arguments

The following list shows the valid arguments.

```
ULong         arg;
Int           winId;
Int32         code;
IntPtr        ret;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application requires.

winId identifies which window the Rules Element is querying.

code describes which status information the kernel needs to get from the interface.

ret is a pointer to an integer where the state of the window will be returned.

Your GetStatus handler should test the second argument (code). If the second argument is different from NXP_GS_ENABLED, the handler must

return FALSE. Otherwise the kernel is querying whether or not the window winId is enabled for output. Your handler can then test the value of winId and set the value of *ret to TRUE (1) or FALSE (0) if it is one of the following codes:

Code	Description
NXP_WIN_TRAN	Transcript window
NXP_WIN_CONC	Conclusions text window
NXP_WIN_HYPO	Hypothesis text window
NXP_WIN_RULE	"Current rule" window

It should return TRUE if winId is one of these codes and FALSE otherwise.

Notes

It is important that the function returns FALSE if it cannot process the call (i.e. code is not NXP_GS_ENABLED or winId is not one of the previous windows) so that the Rules Element uses its internal GetStatus handler. Failing to do so will produce unpredictable results in the interface.

The GetStatus handler is not called every time the Rules Element wants to format a string for the Transcript or another window (for obvious performance issues)! It is called only once after a Knowcess or Continue call, i.e. when a session starts or after an interruption due to a question or a breakpoint. It means that you cannot use a GetStatus handler to modify the status of the Transcript while the engine is running.

The Rules Element's kernel calls the GetStatus handler even if there is no interface, such as in the Runtime library. Although the text windows (Transcript, Current Rule, Current Hypothesis and Conclusions) do not exist as such in the runtime version, you can trap the messages with the SetData handler to display them the way you want.

Examples

Here is how to use the GetStatus handler in conjunction with the SetData handler to trap Transcript strings. Notice that if you forget to install a GetStatus handler that enables the Transcript, the SetData handler won't receive any Transcript strings at all since the Rules Element will think it is not write-enabled.

```
            /* Install the SetData and GetStatus handler to trap Transcript
            */
NXP_SetHandler( NXP_PROC_SETDATA, SetDataHandler, (char *)0);
NXP_SetHandler( NXP_PROC_GETSTATUS, GetStatusHandler, (char *)0);
            /* GetStatus handler */
int    GetStatusHandler(winId, code, ret)
Int           winId;
Int32         code;
IntPtr        ret;
{
       /*
        * Check first that GetStatusHandler is called for
        * Transcript's status otherwise it MUST return 0 to let
        * the Rules Element do its own business!
        */
       if(code != NXP_GS_ENABLED || winId != NXP_WIN_TRAN)
            return 0;
       /*
```

```
        * Set the "enabled" status code to TRUE to get the Transcript
        * strings then return 1 to avoid the default handler
        */
       *ret = TRUE;
       return 1;
}
              /* SetData handler */
int    SetDataHandler(windId, ctrlId, index, str)
Int           winId;
Int32         ctrlId;
Int32         index;
Str           str;         /* string sent to Transcript */
{
       /* doesn't handle other windows than transcript */
       if(winId != NXP_WIN_TRAN)  return 0;

       /* your code to display or use the string str */
       ...

       return 1;
}
```

See Also

`NXP_PROC_SETDATA`	Trap strings for the Transcript and other windows.
`NXP_SetData`	Send strings to the Transcript and other windows.

NXP_SetHandler (2) / NXP_PROC_MEMEXIT

Purpose

This handler is called when a Rules Element memory allocation request fails. By default, when this occurs a message indicating that no more memory is available is displayed and the Rules Element exits. You may specify an alternate procedure to call under this circumstance. This procedure takes no arguments (except arg when installed with NXP_SetHandler2).

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

void NxpMemExit ();

or as follows if it is installed with NXP_SetHandler2:

void NxpMemExit (*arg***);**

Arguments

The following list shows the valid arguments.

`unsigned long arg;`

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

Notes

You can use this procedure for putting up an error message, finding a way to release more memory, cleaning up your environment, and other related purposes.

Your actions during this procedure are limited because anything requiring additional memory is likely to fail (unless you find a way to free up some memory).

If your handler returns 0, the Rules Element will put up its own message and exit. It returns 1, the Rules Element will call it back unless enough memory was freed to continue.

Examples

The following example illustrates how to set up a memory exit handler:

```
NXP_SetHandler(NXP_PROC_MEMEXIT, myMemExit, 0);
void  myMemExit()/* MemExit handler */
{
       /*
       * free up memory ...if possible
       * maybe close files/databases/network links gracefully
       * then exit
       */
       printf("memory exhaustion cleanup/exit triggered\n");
       exit(0);
}
```

In the case of a type 2 handler, the syntax would be as follows:

```
void  MemExitHandler(arg)
ULong  arg;
/* private info, see NXP_SetHandler2 */
{
...
}

NXP_SetHandler2(NXP_PROC_MEMEXIT, MemExitHandler, (Str)0,
                 NXP_HDLTYPE_USER, arg);
```

NXP_SetHandler (2) / NXP_PROC_NOTIFY

Purpose

The notify handler is called each time something changes in the working memory (values, creation and deletion of objects or links). The default notify handler uses this information to keep the consistency of the user interface.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpNotify(*winId***,** *code***,** *theAtom***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpNotify(*arg***,** *winId***,** *code***,** *theAtom***);**

Arguments

The following list shows the valid arguments:

```
ULong        arg;
Int          winId;
Int32        code;
AtomId       theAtom;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

winId identifies the window to which the notification is addressed.

code describes the nature of the notification.

theAtom describes which Atom is involved in the change.

Your Notify handler should first test the value of winId. If winId is different from NXP_WIN_DDE, your handler must return FALSE to let the Rules Element use its default Notify function. Not returning FALSE in that case could lead to serious interface problems! NXP_WIN_DDE is a reserved code for your program to receive notifications. All the other codes represent windows of the interface and they must be handled by the Rules Element.

The second and third arguments passed to your notify handler describe the notification. *code* can take the following values:

Code	Description
NXP_NF_CREATE	If theAtom is NULL you are notified that a knowledge base was loaded. Otherwise you are notified that theAtom has just been created. theAtom can be an object, class, property, slot or rule id.
NXP_NF_DELETE	If theAtom is NULL you are notified that the knowledge base was cleared. Otherwise you are notified that theAtom will be deleted and should no longer be referenced (the id becomes invalid). theAtom can be an object, class, property, slot or rule id.
NXP_NF_MODIFY	This notification informs you that a structural change occurred in the object base. theAtom is the id 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 id of the class as third argument. It is also called when subobjects are created or deleted (the third argument is the parent object in that case).
NXP_NF_REDRAW	This notification is sent every time the system is interrupted or pauses for a question.
NXP_NF_RESTART	The session has been restarted.
NXP_NF_UPDATE	This notification informs you that the value of a slot has changed. theAtom is the id of the slot whose value has changed. This notification differs from an If Change 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.

Examples

Using the code NXP_NF_UPDATE a demon can check for changes in slot values. This example detects changes in the pressure of tank objects:

```
int          myNotify( winId, code, theAtom )
Int          winId;
Int32        code;
AtomId       theAtom;
```

```
{
      AtomId prop, atomProp;

      /* Let the Rules Element do its own business with other
windows */
      if( winId != NXP_WIN_DDE)  return 0;

      if( code != NXP_NF_UPDATE )return 1;

      /* Check that the property is Pressure */
      /* (prop could be a global computed before) */
      NXP_GetAtomId("Pressure", &prop, NXP_ATYPE_PROP);
      NXP_GETATOMINFO( theAtom, NXP_AINFO_PROP, &atomProp);
      if( atomProp != prop)      return 1;

      /* ...  more code ...  */
      return 1;
}
```

NXP_SetHandler (2) / NXP_PROC_PASSWORD

Purpose

This handler is called each time the Rules Element needs to get a password because it is loading an encrypted knowledge base. By default, the Rules Element prompts the user for this password (special dialog window in the Development System).

Your password handler can provide the password directly, query the user for it, or do whatever you prefer. The procedure is called with a 255-character buffer (descriptor for Fortran), and the user should return the password in this buffer. It must be NULL or "0" terminated.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpPassword(*fileName***,** *buf***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpPassword(*arg***,** *fileName***,** *buf***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
Str          fileName;
Str          buf;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

fileName is the name of the knowledge base needing a password. It is passed to the handler and must not be modified.

buf is the character string being passed to the routine, and in which the routine returns the password.

Examples

To set up a Password handler use the following call:

```
NXP_SetHandler(NXP_PROC_PASSWORD, myPassword, 0);
```

The following password handler provides a hard-coded password and then disables the SaveKB function so that nobody can save the decrypted KB:

```
int    myPassword(filename, password)
Str    filename;
Str    password;
{
       printf("providing password for: %s\n", filename);
       strcpy(password, "hello");

       NXP(SetAtomInfo( (AtomId)0, NXP_SAINFO_DISABLESAVEKB,
                                        (AtomId)0, 0, 0, 0);

       return 1;/* success */
}
```

In the above example, the filename is merely printed out for information. Other possibilities include using it to derive the password or to simply ignore it. The password is hard-coded here to be the string "hello". Other options include basing it on the filename, getting it from an environment variable, retrieving it from a database, etc.

This example is not "secure" in that someone may be able to browse the executable image, notice the word "hello", and take steps to break your password. This example, although it illustrates the principle, is not ideal if high security levels are desired.

In the case of a type 2 handler, the syntax would be as follows:

```
int PasswordHandler(arg, filename, buf)
ULong               arg;
Str                 filename;
Str                 buf;
{
...
}

NXP_SetHandler2(NXP_PROC_PASSWORD, (NxpIProc)PasswordHandler,
                (char *)0, NXP_HDLTYPE_USER, arg);
```

See Also

`NXP_SetAtomInfo / NXP_SAINFO_DISABLESAVEKB`	Disable the saving of KBs.

NXP_SetHandler (2) / NXP_PROC_POLLING

Purpose

The Polling handler is called after each inference engine cycle. The processing done by the inference engine can be viewed as a succession of processing cycles. Between two cycles, the engine is in a stable state and information can be exchanged with the outside (read by the interface program with the NXP_GetAtomInfo code or modified with NXP_Volunteer, NXP_Suggest, NXP_CreateObject, ...).

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpPolling();

or as follows if it is installed with NXP_SetHandler2:

int NxpPolling(*arg***);**

Arguments

```
ULong  arg;
```

*arg i*s passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

Return Value

The handler should return FALSE if you want the default polling procedure to be called afterwards, TRUE otherwise. In the development system, the default procedure checks if a click occurred in the interrupt button of the session control window. The runtime library doesn't have any default polling.

Notes

The Polling handler can call any Rules Element function. It can interrupt the engine by calling NXP_Control with the code NXP_CTRL_STOPSESSION. Remember that the more time consuming the handler is the more the engine will be slowed down.

An inference cycle is a basic operation of the inference engine like changing the status of a variable or performing one internal step in the evaluation of an expression (for instance the expression "ASSIGN N+1 N" contains internally several steps). It can't be described and timed precisely, it all depends on the current operation and the speed of the CPU. The best way for you to know how fast these ticks happen in your application running on your computer is to write a small polling handler that just counts how many times it is called! The result can only be interpreted as an average: the Rules Element doesn't guarantee any particular frequency. Neither are there possibilities of timeout for inferences.

In order to "synchronize" the engine with a realtime clock, a possible (but non ideal) solution is the following:

1. Evaluate the maximum delay between two inference ticks while running your application without file I/O operations (see warning below). Let's say it is K ticks per second.
2. Write a Polling handler that does nothing but wait until the current tick takes 1/K second.

Warning: This should work fairly well if your evaluation in (1) was good, except during file I/O operations such as Loading a KB and Retrieving data from a database. During those operations the engine is not running per se and thus the polling handler is not called. However, these file I/O operations call repeatedly the "Cancel" handler, so you could install your own Cancel handler to perform the same code as the Polling handler during the inference.

Examples

Monitoring of a (fast) changing value.

```
AtomId  SlotAtom;   /* global variable */

/* polling handler reads the value of slotAtom at every cycle */
int myPollingFct()
{
       float  theFloat;

       NXP_GetAtomInfo(SlotAtom, NXP_AINFO_VALUE, (AtomId)NULL, 0,
                           NXP_DESC_FLOAT, (Str)&theFloat, 0);
       ...
       /* code could draw the current value on a graphic ...  */
       ...
       return( FALSE )  /* default polling function will be called */
}

main()
{
       NXP_GetAtomId( "Ferrari.Speed", &SlotAtom, NXP_ATYPE_SLOT );
       NXP_SetHandler( NXP_PROC_POLLING, (NxpIProc)myPollingFct, (Str)0 );
       ...
}
```

NXP_SetHandler (2) / NXP_PROC_QUESTION

Purpose

The Question handler is called each time a new question is asked by the inference engine. It is one of the most important handlers to install in your application because, generally, it provides the main part of the end-user interface.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpQuestion(*qAtom***,** *qStr***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpQuestion(*arg***,** *qAtom***,** *qStr***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
AtomId       qAtom;
Str          qStr;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

qAtom is the id of the slot on which the current question is asked.

qStr is the question prompt. It is either the prompt line string entered in the meta-slot of qAtom, the prompt-line inherited from a parent object or class, or the default the Rules Element prompt-line "What is the <property> of <object>?".

Return Value

The question procedure should return TRUE if it has set the value asked for (with NXP_Volunteer), and FALSE if you want the Rules Element to bring up the default question. The default question comes in the Session Control window in the development system. There is no default question in the runtime library so you must provide a Question handler!

Notes

The value of qAtom should be set with the NXP_Volunteer call and the NXP_VSTRAT_QFWRD priority (forward with the current priority).

A modal question handler is a handler that doesn't return to the Rules Element until the value is volunteered (either the value was found "automatically" by the program or it was entered by the user in a modal dialog window).

The question can be made non modal by calling NXP_Control with code equal to NXP_CTRL_STOPSESSION and returning 1 to the Rules Element (See examples in the Primer). The session stops, the Rules Element returns from the initial NXP_Control(NXP_CTRL_KNOWCESS) call, and your application gets the control (if the application is the Rules Element itself, it comes back to the main event loop of the interface). After the user or the application finds an answer to the question you can call NXP_Volunteer to set the value and then call NXP_Control(NXP_PROC_CONTINUE) to resume the session, until the next question or the end of session.

If the question handler returns 1 without answering to the question it will loop and come back with the same question.

A Rules Element question is always for one slot at a time. For instance if 10 properties of an object must be known by the system it will call the question handler 10 times with a different slot qAtom each time. You may want to provide the end-user with only one form containing 10 fields to be filled. One way to do this is to bring up that form at the first question and then to volunteer the 10 values before returning from the question. The Rules Element won't call you back with the 9 other questions now that the values are set.

Examples

See the Hello examples of Chapter Two, "Primer".

NXP_SetHandler (2) / NXP_PROC_QUIT

Purpose

This handler is called when the Rules Element is going to exit.

This handler is useful if your application needs to do some cleaning up when the Rules Element quits. It can also be used in a client-server type of application, where you need to be notified of the status of the server (the Rules Element). If the server quits, you can close the connection gracefully.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpQuit(*thePtr***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpQuit(*arg***,** *thePtr***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
IntPtr       thePtr;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

thePtr is a pointer to an integer. It should be set to TRUE (1) if the Rules Element is allowed to quit, and to FALSE (0) if not (for instance, if a transaction controlled from your code was still active).

The function should return TRUE if it processes the call, and FALSE if not (in which case, the Rules Element's default Quit handler is used and it allows the Rules Element to exit).

Notes

This handler can only be called from the interface where there is a Quit command. If your application is linked with the runtime library, such a Quit event will not come from the Rules Element. Conversely, you must call NXP_Control / NXP_CTRL_EXIT if your application does not need the Rules Element anymore.

Examples

The following example illustrates a standard Quit handler:

```
int QuitHandler(ret)
IntPtr       ret;
{
  /* If some transaction is still active, etc., don't allow the
Rules Element to quit */
       *ret = FALSE;
       return TRUE;

       /* Else clean up stuff or close connection */
       ...
       /* and let the Rules Element exit */
       *ret = TRUE;
       return TRUE;
}
```

You must install the Quit handler with the following call (the last argument, name, is ignored for codes other than NXP_PROC_EXECUTE):

```
NXP_SetHandler(NXP_PROC_QUIT, QuitHandler, (char *)0);
```

In the case of a type 2 handler, the syntax would be as follows:

```
int QuitHandler(arg, ret)
ULong        arg;  /* private info, see NXP_SetHandler2 */
IntPtr       ret;
{
...
}
NXP_SetHandler2(NXP_PROC_QUIT, (NxpIProc)QuitHandler, (Str)0,
NXP_HDLTYPE_USER, arg);
```

See Also

`NXP_Control / NXP_CTRL_EXIT`	Notify the Rules Element that an application does not need to communicate anymore.
`NXP_Control / NXP_CTRL_INIT`	Initialize the Rules Element.
`NXP_PROC_ENDOFSESSION`	To be notified of the end of a session.

NXP_SetHandler (2) / NXP_PROC_SETDATA

Purpose

This handler is called when the Rules Element sends data to a window. It can be used mainly to trap Transcript or Banner strings, or in conjunction with the GetData handler to perform complex edit functions (see Chapter Six).

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpSetData(*WinId***,** *CtrlId***,** *Index***,** *thePtr***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpSetData(*arg***,** *WinId***,** *CtrlId***,** *Index***,** *thePtr***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
Int          winId;
Int32        ctrlId;
Int32        index;
Str          thePtr;
```

*arg i*s passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

winId is the id of the window to which the message is sent.

ctrlId is an integer describing which sub part of the window the message is sent to.

index is an additional integer describing how the information should be displayed.

thePtr is a pointer to a string where the string being sent is stored.

This handler can do some processing and return TRUE in the following cases:

Code	Description
winId = NXP_WIN_BANNER	A new string should be displayed in the banner window. This window is used for temporary messages requiring no user interaction (It is called for displaying the "Load Knowledge Base" "Clear Knowledge base" "Retrieving from..." messages). The default window is divided into three parts corresponding to three different values for code & NXP_CELL_MASK. It is the work of the called procedure to put up the banner on the screen at the first call. When the Rules Element is done with the messages, it will call NxpSetData with CtrlId & H_CELL_MASK equal to H_CELL_NONE and thePtr equal to NULL. The banner window should then be removed.
winId = NXP_WIN_TRAN	A new string is sent to the Transcript window. If index is -1 or -2, the string should be pasted at the end of the window. If index is 0, the string should be pasted at the beginning of the window.
winId = NXP_WIN_CONC	Same as NXP_WIN_TRAN except that the message is sent to the conclusion text window.
winId = NXP_WIN_HYPO	Same as NXP_WIN_TRAN except that the message is sent to the hypothesis text window.
winId = NXP_WIN_RULE	Same as NXP_WIN_TRAN except that the message is sent to the"current rule" window.

For any other value of winId, this handler should return FALSE, otherwise you will have serious interface problems.

Examples

See the example in the GetStatus handler section. In order to trap Transcript strings you must first install a GetStatus handler to tell the Rules Element that the Transcript is write-enabled.

See also

NXP_PROC_GETDATA	GetData handler.
NXP_PROC_GETSTATUS	GetStatus handler.

NXP_SetHandler (2) / NXP_PROC_VALIDATE

Purpose

The Validate handler is called when the inference engine is triggering the data validation of the Atom which has a user-defined routine for data validation. It is called with the arguments described below after the boolean data validation expression defined in the meta-slot editor if any has been triggered (whether it has been satisfied or not).

Note: Do not call NXP_Volunteer on the Atom being evaluated.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpValidate (*dataType***,** *Atom***,** **Ptr***,** *Status***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpValidate (*arg***,** *dataType***,** *Atom***,** **Ptr***,** *Status***);**

Arguments

The following list shows the valid arguments:

```
ULong        arg;
int          dataType;
AtomId       Atom;
VoidPtr      Ptr;
IntPtr       Status;
```

*arg i*s passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

dataType is the type of Atom. Potential values are: NXP_DESC_INT, NXP_DESC_FLOAT, NXP_DESC_STR, NXP_DESC_DOUBLE, NXP_DESC_LONG.

Atom is the slot on which the data validation is done. It should be a valid slot Id.

Ptr is a pointer to the value the user / engine is trying to assign to Atom.

Status is a pointer to an integer containing the current decision based on the decision from the boolean data validation expression or NXP_ERROR_NOERROR in case of no boolean data validation expression. Potential values are:

Status	Description
NXP_ERR_NOERROR	If the value was satisfied by the boolean data validation expression or if the boolean data validation expression couldn't be completely evaluated and the strategy was set to ON / ACCEPTED.
NXP_ERR_VALIDATEMISSING	If the expression could not be evaluated due to missing information, and the strategy was not ON / ACCEPT.
NXP_ERR_VALIDATEERROR	If the boolean data validation expression would cause the validation to fail.
NXP_ERR_VALIDATEUSER	If the value proposed by the user is rejected (boolean data validation expression not satisfied).

Return Codes

Your Validate handler returns:

NXP_ERR_NOERROR	If the value should be accepted.
NXP_ERR_VALIDATEUSER	If the value is rejected.

Examples

Here is how to use Validate handler to systematically reject string values which are within the slot options list:

```
/* Example of Validate handler */
int MyValidate( int dataType, AtomId atom, VoidPtr ptr, IntPtr status)
{
      int          n, i;
      Char   str[256];
```

```
        if (dataType != NXP_DESC_STR) return = 1;
        *status = NXP_ERR_VALIDATEUSER;
        NXP_GETLISTLEN( atom, NXP_AINFO_CHOICE, &n);
        for (i=0; i < n; i++) {
                NXP_GETLISTELTSTR( atom, NXP_AINFO_CHOICE, i, str, 255);
                if (STR_Cmp( str, ptr) == BOOL_TRUE) *status = NXP_ERR_NOERROR;
        }
        return 1;
}
```

NXP_SetHandler (2) / NXP_PROC_VOLVALIDATE

Purpose

This handler lets you supply your data validation function, which is application-dependent. (VOLVALIDATE stands for Volunteer-Validate.)

This handler is called from the Rules Element during a Volunteer with the same first four parameters (atom, desc, ptr, prio) that are passed to NXP_Volunteer. There is one additional argument, ret, to return the result of the data validation.

C Format

The C format is as follows if the handler is installed with NXP_SetHandler:

int NxpVolValidate(*theAtom***,** *desc***,** *thePtr***,** *prio***,** *ret***);**

or as follows if it is installed with NXP_SetHandler2:

int NxpVolValidate(*arg***,** *theAtom***,** *desc***,** *thePtr***,** *prio***,** *ret***);**

Arguments

The following list shows the valid arguments.

```
ULong        arg;
AtomId       theAtom;
int          desc;
Str          thePtr;
int          prio;
IntPtr       ret;
```

arg is passed to the procedure as it was passed to NXP_SetHandler2. It can contain any information your application wishes.

The VolValidate handler does not trap the internal setting of values as the inference engine would. It is intended to trap the calls to NXP_Volunteer made from the interface (Question window) or any program using the Application Programming Interface. The first four arguments received by the handler are the ones that were just passed to the NXP_Volunteer call being trapped:

theAtom is the slot Id being modified.

desc describes the format of the value referenced by thePtr. desc can be one of the following codes: NXP_DESC_DOUBLE, NXP_DESC_FLOAT, NXP_DESC_INT, NXP_DESC_NOTKNOWN, NXP_DESC_STR, or NXP_DESC_UNKNOWN.

thePtr is a pointer to the new value associated with theAtom.

prio describes the priority to be used by the inference engine when forwarding the new value.

See NXP_Volunteer for more information on the previous four arguments.

ret is a pointer to an integer where the result of the data validation will be returned. *ret should be set to TRUE (1) if the value is considered valid, and set to FALSE (0) if it is invalid.

If the handler recognizes the value as invalid, NXP_Volunteer won't modify theAtom's value and will return FALSE. NXP_Error() subsequently returns NXP_ERR_VOLINVAL.

If you find an invalid value and wish to modify the value in your VolValidate handler, you should call NXP_Volunteer (see the following example). Warning: In this case, your VolValidate handler will have been called a second time within the NXP_Volunteer call, so the function should be made reentrant!

Return Value

This handler returns TRUE if it processes the data validation, and returns FALSE otherwise. If the handler returns FALSE to indicate that it does not process the data validation, the Rules Element's default handler automatically validates the value (so this is equivalent to setting *ret to TRUE and returning TRUE).

Examples

The following example shows how your VolValidate handler can handle different cases:

```
int MyVolValidate(atom, desc, ptr, prio, ret)
AtomId       theAtom;
int          desc;
Str          thePtr;
int          prio;
IntPtr       ret;
{
       /* returns immediately for atoms not being checked */
       ...
       return FALSE;
       /* code to check whether the value for atom is correct */
       ...

       /* case 1: value correct and will be volunteered */
       *ret = TRUE;
       return TRUE;

       /* case 2: value is incorrect and don't volunteer */
       *ret = FALSE;
       return TRUE;

       /* case 3: modify the value, don't volunteer current one
*/
       NXP_Volunteer(atom, newDesc, newPtr, newPrio);
       *ret = FALSE;
       return TRUE;
}
```

Additionally, you could use the ClientData information to store the valid values with each atom and see whether or not an atom needs to be checked (see NXP_SetClientData, NXP_GetAtomInfo /

NXP_AINFO_CLIENTDATA). You can also keep this information as global data in your program, or pass it to the VolValidate handler by using NXP_SetHandler2 (see the following example).

You must install the VolValidate handler with the following call (the last argument, name, is ignored for codes other than NXP_PROC_EXECUTE):

```
NXP_SetHandler(NXP_PROC_VOLVALIDATE, MyVolValidate,   (char
*)0);
```

Using NXP_SetHandler2 to pass additional information to the VolValidate handler, you can get the list of atoms that needs to be checked and the list of valid values. The following example shows a fixed list of N atoms of type integer and K possible values for each of them:

```
struct myInfo  {
       AtomId  atomsToCheck[ N ];
       int     validValues[ K ];
       int     validPrio;
}      *structPtr;

int MyVolValidate2(arg, atom, desc, ptr, prio, ret)
ULong         arg;
AtomId        atom;
int           desc;
Str           thePtr;
int           prio;
IntPtr        ret;
{
       struct myInfo    *structPtr = arg;
       int i;

       /* See if atom needs to be checked */
       for(i = 0; i < N; i++) {
       if( structPtr->atomsToCheck[i] == atom)
       goto checkIt;
       }
       return FALSE;

checkIt:
       /* check that it is volunteered as an integer */
       if( desc != NXP_DESC_INT )   {
              *ret = FALSE;
              return TRUE;
       }

       /* check that it is volunteered with the right priority
*/
       if( prio != structPtr->validPrio)  {
              *ret = FALSE;
       return TRUE;
       }

       /* check that it has one of the K possible values */
       for( i = 0; i < K; i++) {
              if( *(int *)ptr == structPtr->validValues[i] ) {
                     *ret = TRUE;
                     return TRUE;
              }
       }
       *ret = FALSE;
       return TRUE;
}
```

NXP_SetHandler2 also allows you to install the MyVolValidate handler and pass the pointer structPtr as shown in the following example (the default value NXP_HDLTYPE_USER is used for type, which is not important here):

```
NXP_SetHandler2(NXP_PROC_VOLVALIDATE, MyVolValidate, (char *)0,
                NXP_HDLTYPE_USER, structPtr);
```

See Also

`NXP_Volunteer`	Volunteer the value of a slot.

NXP_Strategy

Purpose

NXP_Strategy changes the strategy of the inference engine. You can change either the default strategy saved with the knowledge base or the current strategy of the inference engine.

C Format

The C format is as follows.

int NXP_Strategy(*code***,** *bool***);**

Arguments

The following list shows the valid arguments.

```
int    code;
int    bool;
```

code corresponds to the multiple options of the Strategy window in the development system. It is one of the following:

Code	Description
`NXP_AINFO_BREADTHFIRST`	Breadth first versus depth first strategy.
`NXP_AINFO_CACTIONSON`	If Change methods enabled or not.
`NXP_AINFO_CACTIONSUNKNOWN`	If Change methods will be executed when the slot is set to `UNKNOWN`.
`NXP_AINFO_EXHBWRD`	Exhaustive evaluation of the backward chaining.
`NXP_AINFO_INHCLASSDOWN`	Downward inheritability of class `slots`.
`NXP_AINFO_INHCLASSUP`	Upward inheritability of class slots.
`NXP_AINFO_INHOBJDOWN`	Downward inheritability of object slots.
`NXP_AINFO_INHOBJUP`	Upward inheritability of object slots.
`NXP_AINFO_INHVALDOWN`	Downward inheritability of the value of a slot.
`NXP_AINFO_INHVALUP`	Upward inheritability of the value of a slot.
`NXP_AINFO_PARENTFIRST`	Parent first versus class first strategy.
`NXP_AINFO_PFACTIONS`	Forward LHS/RHS actions from rules.
`NXP_AINFO_PFELSEACTIONS`	Forward Else actions from rules.
`NXP_AINFO_PTGATES`	Forwarding through gates.
`NXP_AINFO_PFMETHODACTIONS`	Forward LHS/RHS actions from methods.
`NXP_AINFO_PFMETHODELSEACTIONS`	Forward Else actions from methods.
`NXP_AINFO_PWFALSE`	Context propagation on FALSE hypotheses.

Code	Description
NXP_AINFO_PWNOTKNOWN	Context propagation on NOTKNOWN hypotheses.
NXP_AINFO_PWTRUE	Context propagation on TRUE hypotheses.
NXP_AINFO_SOURCESCONTINUE	Order of Sources methods will be fully executed.
NXP_AINFO_SOURCESON	Order of Sources methods enabled or not.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of value set by the engine enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDENGINE_OFF	Validation of value set by the engine disabled.
NXP_AINFO_VALIDENGINE_ON	Validation of value set by the engine enabled.
NXP_AINFO_VALIDENGINE_REJECT	Validation of value set by the engine enabled and the value rejected automatically if the validation expression is incomplete.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of value entered by the end user enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDUSER_OFF	Validation of value entered by the end user disabled.
NXP_AINFO_VALIDUSER_ON	Validation of value entered by the end user is enabled.
NXP_AINFO_VALIDUSER_REJECT	Validation of value entered by the end user enabled and the value rejected automatically if the validation expression is incomplete.

If the NXP_AINFO_CURSTRAT bit is set in code, the current strategy is modified, otherwise, the default strategy (saved with the knowledge base) is modified.

I*f bool* is 0, the strategy setting is turned off, otherwise, it is turned on.

Current strategies can be examined through various NXP_AINFO_XXX codes dealing with strategies (See Chapter Four "NXP_GetAtomInfo Routine").

Return Codes

NXP_Strategy returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	code is invalid.
NXP_ERR_NOERR	Call was successful.

Examples

The following example reads the current strategy concerning the forwarding through gates and turns the forwarding off if it was on. It resets the current strategy later.

```
int    code = NXP_AINFO_PTGATES | NXP_AINFO_CURSTRAT;
int    curStrat;

NXP_GETINTINFO( (AtomId)0, code, &curStrat);
if(curStrat == TRUE)
       NXP_SetStrategy (NXP_AINFO_PTGATES, FALSE);
```

```
/* some code ...  */

if(curStrat == TRUE)
      NXP_SetStrategy (NXP_AINFO_PTGATES, curStrat);
```

See Also

`NXP_GetAtomInfo / NXP_AINFO_XXX` Information codes on strategies.

NXP_Suggest

Purpose

NXP_Suggest suggests a hypothesis (i.e. puts it in the Suggest list of the agenda).

C Format

The C format is as follows.

int NXP_Suggest(*theAtom***,** *prio***);**

Arguments

The following list shows the valid arguments.

```
AtomId       theAtom;
int          prio;
```

NXP_Suggest suggests further evaluation of the atom pointed to by theAtom with the priority prio. If theAtom was already queued with a priority less than prio it will be rescheduled with the higher priority.

theAtom must be a valid hypothesis Id.

prio describes when theAtom will be processed. The possible codes are:

Code	Description
`NXP_SPRIO_CNTX`	theAtom will compete with the contexts.
`NXP_SPRIO_DATAISL`	theAtom is queued in the current knowledge island with a priority less than NXP_SPRIO_HYPISL. All the hypotheses queued with NXP_SPRIO_HYPISL will be investigated before any of those queued with NXP_SPRIO_DATAISL.
`NXP_SPRIO_FORCE`	theAtom is forced for immediate evaluation. The current tasks are saved and will be resumed once theAtom has been evaluated. This code is not implemented.
`NXP_SPRIO_HYPISL`	theAtom is queued in the current knowledge island.
`NXP_SPRIO_SUG`	theAtom is queued for evaluation with the same priority as if it was suggested from the interface through a pop up menu or through the Suggest global menu. The current atom being investigated is evaluated and then control switches to theAtom.
`NXP_SPRIO_UNSUG`	theAtom will be removed from the agenda.

Return Codes

NXP_Suggest returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error

immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid hypothesis Id.
NXP_ERR_INVARG2	prio is not one of the valid priority codes.
NXP_ERR_NOERR	Call was successful.

Examples

This example checks if a hypothesis is already suggested and if not suggests it:

```
AtomId        HypoId;
int           IsSuggested;
int           ret;

/* Get the id of Hypo1 */
ret = NXP_GetAtomId("Hypo1", &HypoId, NXP_ATYPE_HYPO)
if (ret == 0) {
       /* error message...  */
}

/* Check if the Hypo is already suggested */
ret = NXP_GETINTINFO(HypoId, NXP_AINFO_SUGGEST, &IsSuggested);

if (IsSuggested == FALSE) {
       /* Suggest the hypo */
       ret = NXP_Suggest(HypoId, NXP_SPRIO_SUG)

       /* Check again...  */
       ret = NXP_GETINTINFO(HypoId, NXP_AINFO_SUGGEST, &IsSuggested);

       if (IsSuggested != TRUE) {
              /* something is wrong...  */
       } else {
              /* Hypo1 was correctly suggested...  */
       }
}
```

See Also

NXP_GetAtomInfo / NXP_AINFO_SUGGEST	To know if hypothesis is suggested.
NXP_GetAtomInfo / NXP_AINFO_FOCUSPRIO	Get focus priority of the hypothesis.

NXP_UnloadKB

Purpose

NXP_UnloadKB controls the knowledge bases loaded by NXP_LoadKB. It can unload, disable, or reenable a knowledge base.

C Format

The C format is as follows.

int NXP_UnloadKB(*theKBId***,** *level***);**

Arguments

The following list shows the valid arguments.

```
KBId        theKBId;
int         level;
```

the*KBI*d must be a valid knowledge base id (as returned by a previous call to NXP_LoadKB or NXP_GetAtomId).

level must be one of the following constants:

Code	Description
NXP_XLOAD_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.
NXP_XLOAD_DISABLESTRONG	The knowledge base is disabled and its rules, hypotheses, methods are removed from the agenda of the inference engine.
NXP_XLOAD_DISABLEWEAK	The knowledge base is disabled but the agenda of the inference engine is not modified.
NXP_XLOAD_ENABLE	The knowledge base is enabled.
NXP_XLOAD_WIPEOUT	Same as NXP_XLOAD_DELETE but the data structures associated with the objects and classes are also released.

Notes

Disabling a knowledge base with NXP_XLOAD_DISABLEWEAK or NXP_XLOAD_DISABLESTRONG and reenabling it later with NXP_XLOAD_ENABLE are very fast operations (resetting or setting a flag in memory). On the contrary, when a knowledge base is deleted (NXP_XLOAD_DELETE or NXP_XLOAD_WIPEOUT), the delete operation is not as fast and the knowledge base must be reloaded from a file if its rules or methods are required later during the session (in this case, theKBId becomes invalid after the NXP_UnloadKB call).

If there is only one knowledge base loaded using NXP_XLOAD_WIPEOUT has the same effect as calling NXP_Control(NXP_CTRL_CLEARKB).

Return Codes

NXP_UnloadKB returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theKBId is not a valid knowledge base id.
NXP_ERR_INVARG	level is invalid.
NXP_ERR_NOERR	Call was successful.

Examples

In the following example kb1 and kb2 share the same objects and classes, so we unload kb1 with NXP_XLOAD_DELETE to remove only the rules.

```
KBId kb1, kb2;

/* loads a first knowledge base */
NXP_LoadKB("kb1.tkb", &kb1);

/* suggest, knowcess, ...  */

/* unload kb and release its memory */
NXP_UnloadKB(kb1, NXP_XLOAD_DELETE);

/* load a different kb using the same classes */
NXP_LoadKB("kb2.tkb", &kb2);

/* suggest, knowcess, ...  */
```

In an execute routine, you can disable or reenable a knowledge base (the knowledge base id was obtained by a previous call to NXP_LoadKB or NXP_GetAtomId):

```
/* disabling a kb */
NXP_UnloadKB(kb, NXP_XLOAD_DISABLEWEAK);

/* reenabling kb later */
NXP_UnloadKB(kb, NXP_XLOAD_ENABLE);
```

These operations are very fast. NXP_XLOAD_DELETE followed by a NXP_LoadKB would be much slower, but you would have more memory available while the knowledge base is unloaded.

See Also

`NXP_LoadKB`	Load a knowledge base.
`NXP_Control / NXP_CTRL_CLEARKB`	Clear all knowledge bases.

NXP_Volunteer

Purpose

NXP_Volunteer volunteers the value of a slot.

C Format

The C format is as follows.

int NXP_Volunteer(*theAtom***,** *desc***,** *thePtr***,** *prio***);**

Arguments

The following list shows the valid arguments.

```
AtomId       theAtom;
int          desc;
Str          thePtr;
int          prio;
```

NXP_Volunteer will change the value associated with theAtom. Values can be passed in different formats (as text, as numbers). Priorities describe how the new value should be forwarded in the inference network.

theAtom should be a valid slot id. It is not possible to set rule or condition values directly.

thePtr is a pointer to the new value.

*des*c describes the format of *thePtr. desc can be one of the following codes:

Code	Description
NXP_DESC_DOUBLE	Same as NXP_DESC_INT except that thePtr should be a pointer to a double (64 bit IEEE standard on AT, DFloat on VAX).
NXP_DESC_FLOAT	Same as NXP_DESC_INT except that thePtr should be a pointer to a float (32 bit IEEE standard on AT, FFloat on VAX). If theAtom is a string value, the float will be converted to a string and that string value will be assigned to theAtom.
NXP_DESC_INT	thePtr should be a pointer to an integer. If theAtom is a boolean value, any value not NULL of *thePtr will set theAtom to TRUE, otherwise to FALSE. If theAtom is a numeric value, *thePtr will be transferred into theAtom. If theAtom is a string value, the integer will be converted into a string and that string value will be assigned to theAtom.
NXP_DESC_LONG	thePtr should be a pointer to a long. If theAtom is a boolean value, any value not NULL of *thePtr will set theAtom to TRUE, otherwise to FALSE. If theAtom is a numeric value, *thePtr will be transferred into theAtom. If theAtom is a string value, the long will be converted into a string and that string value will be assigned to theAtom.
NXP_DESC_NOTKNOWN	theAtom will be set to NOTKNOWN. thePtr should be NULL.
NXP_DESC_STR	thePtr should be a pointer to a string which will be used for determining the new value of theAtom. For boolean values the string can be TRUE or FALSE (case independent), for numeric values the string will be converted using the Rules Element's default formats (unless user-provided formats are available), and for string values the string will be transferred as is.
NXP_DESC_UNKNOWN	theAtom will be set back to UNKNOWN. thePtr should then be NULL. In this case, if prio is equal to NXP_VSTRAT_RESET and if theAtom is a hypothesis, the backward chaining from theAtom will be reset (having the same effect as a reset in a rule). Otherwise prio must be equal to NXP_VSTRAT_NOFWRD.
NXP_DESC_VALUE	Not implemented in this version.

prio describes the priority to be used by the inference engine when forwarding the new value. It can be any of the following codes:

Code	Description
NXP_VSTRAT_CURFWRD	Same as NXP_VSTRAT_VOLFWRD except that the global strategy setting "Forward Action Effects" will be checked first. If it is off, the value will not be forwarded.
NXP_VSTRAT_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, unless the slot was explicitly volunteered.
NXP_VSTRAT_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).
NXP_VSTRAT_RESET	Used for resetting the backward chaining on a hypothesis. The value will be set back to UNKNOWN with its backward chaining.

Code	Description
NXP_VSTRAT_RHSFWRD	The new value will be forwarded in the rule network as if it was set from inside a RHS. The engine will not examine all the possible pattern matching rules (selective forward) but will investigate the strong links.
NXP_VSTRAT_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 option 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.

Notes

The strings UNKNOWN and NOTKNOWN (case independent) are also recognized correctly by the Rules Element. Therefore, passing the string NOTKNOWN is the same as passing desc =NXP_DESC_NOTKNOWN. When the value is passed as a string, The Rules Element uses the format information associated with theAtom to convert the string into the internal data type representation.

Return Codes

NXP_Volunteer returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid slot id.
NXP_ERR_INVARG2	desc is not a valid NXP_DESC_... code.
NXP_Error() Return Code	Explanation
NXP_ERR_INVARG3	desc is equal to NXP_DESC_STR but the string is empty.
NXP_ERR_INVARG4	prio is not a valid NXP_VSTRAT_... code.
NXP_ERR_NOERR	Call was successful.

Examples

Several examples of NXP_Volunteer follow:

```
AtomId       BoolAtom;
AtomId       NumbAtom;
AtomId       StringAtom;
int          theBool;
float        theFloat;

/* set the value of Sensor1.Setup to TRUE and forward it as a RHS */
NXP_GetAtomId("Sensor1.Setup", &BoolAtom,NXP_ATYPE_SLOT)

theBool = 1;
NXP_Volunteer(BoolAtom, NXP_DESC_INT, (Str)&theBool, NXP_VSTRAT_RHSFWRD);

/* set the Pressure of Sensor1 to 9.789 but do not forward it */
NXP_GetAtomId("Sensor1.Pressure", &NumbAtom, NXP_ATYPE_SLOT);
theFloat = 9.789;
NXP_Volunteer(NumbAtom, NXP_DESC_FLOAT, (Str)&theFloat, NXP_VSTRAT_NOFWRD);

/* answer to the question about the Manufacturer of Sensor1 */
NXP_GetAtomId("Sensor1.Manufacturer", &StringAtom, NXP_ATYPE_SLOT);
NXP_Volunteer(StringAtom, NXP_DESC_STR, "Martin_Marietta",
```

```
NXP_VSTRAT_QFWRD);

/* reset all the rules leading to hypo1 */
NXP_GetAtomId("hypo1", &BoolAtom, NXP_ATYPE_HYPO);
NXP_Volunteer(BoolAtom, NXP_DESC_UNKNOWN, (Str)NULL, NXP_VSTRAT_RESET);
```

Advanced Users

When setting a new value, the Rules Element can force the value immediately into the slot or queue the value which will be processed and forwarded later. The consequences of this are that if you queue a value and read the slot value immediately after, the old value might still be there. This is because the new value is still in the forwarding queue waiting to be processed by the engine. Sometimes one wants the value to be forced in the slot immediately, for example when volunteering values from the interface and then browsing the data notebook. The extra codes NXP_VSTRAT_SET and NXP_VSTRAT_QUEUE give better control over this matter. If none of those codes is specified, then NXP_VSTRAT_QUEUE will be assumed.

Specifying NXP_VSTRAT_SET forces the new value in the slot immediately, but doesn't forward it. NXP_VSTRAT_QUEUE will also queue the new value with the forwarding priority. The value will be then set when control is passed back to the engine (return from the call and call to Knowcess).

NXP_VSTRAT_SETQUEUE (=NXP_VSTRAT_SET | NXP_VSTRAT_QUEUE) will both set the value immediately and forward it later.

Note: The forwarding queue is a first in, first out queue.

An advanced example follows. It illustrates the usage of NXP_VSTRAT_SET and NXP_VSTRAT_QUEUE.

```
NXP_GetAtomId("Sensor1.Temperature", &NumbAtom, NXP_ATYPE_SLOT);

/* force the value in the slot immediately */
NXP_Volunteer(NumbAtom, NXP_DESC_STR,
"0.987",NXP_VSTRAT_NOFWRD|NXP_VSTRAT_SET);
NXP_GetAtomInfo(NumbAtom, NXP_AINFO_VALUE, 0, (AtomId)NULL,
                NXP_DESC_FLOAT, (Str)&theFloat, 0);

if (theFloat != 0.987) {
       /* something is very wrong */
       ....
}

/* queues the value without setting it immediately */
NXP_Volunteer( NumbAtom, NXP_DESC_STR, "123.456",
NXP_VSTRAT_VOLFWRD|NXP_VSTRAT_QUEUE );
NXP_GetAtomInfo( NumbAtom, NXP_AINFO_VALUE, 0,
                 (AtomId)NULL, NXP_DESC_FLOAT, (Str)&theFloat, 0);

/* the value should still be equal to 0.987 */
if (theFloat != 0.987) {
             ...
}

/* suppose the inference engine does not change that value.
 * Give control back to the inference engine and try reading value later
 */
```

```
NXP_GetAtomInfo(NumbAtom, NXP_AINFO_VALUE, 0,
                (AtomId)NULL, NXP_DESC_FLOAT, (Str)&theFloat, 0);

/* the value should have been changed to 123.456 */
if (theFloat != 123.456) {
       /* problem...  */
}
```

NXP_VolunteerArray

Purpose

This allows you to group in one call the volunteer of several data items of different types provided within an array.

C Format

The C format is as follows:

int NXP_VolunteerArray(*count***,** *type***,** *atoms***,** *descs***,** *ptrs***,** *strats***);**

Arguments

The following list shows the valid arguments:

```
int                 count;
AtomSpecEnum        type;
VoidPtr             atoms;
int                 *descs;
VoidPtr             *ptrs;
int                 *strats;
```

count is the number of atoms to be volunteered.

type indicates whether atoms is an array of AtomIds or an array of atom names. Probably declared as an Enum of either NXP_DESC_ATOM or NXP_DESC_STR.

*atoms i*s an array of either AtomIds (if NXP_DESC_ATOM was specified for type) or strings representing atom names (if NXP_DESC_STR) was passed.

descs is an array of ints describing the format of the program's data -- NXP_DESC_INT, NXP_DESC_STR, etc. See NXP_DESC_DATE below for information on a new date descriptor.

Code	Description
NXP_DESC_DATE	This is a new descriptor to speed up the volunteering (and retrieval) of dates in the Rules Element. It describes an array of 6 integers and contains Month, Day, Year (19xx), Hour, Minute, and Second. Obviously, patterns are not involved. Internally, the Rules Element stuffs the integers into a DateRec and calls DateFix() to fill in the day-of-week stuff.

ptrs points to an array of pointers to the data to be volunteered.

strats points to an array of ints indicating the NXP_VSTRAT_XXX settings for each atom. See NXP_VSTRAT_NOCHECK below for information on a new volunteering strategy.

Code	Description
NXP_VSTRAT_NOCHECK	This is a NEW strategy which can be used with NXP_VolunteerArray and NXP_VolunteerList to speed up volunteering information into the Rules Element. It is much like NXP_VSTRAT_SET, except that it has other restrictions:

- NO notifications for the change (especially transcript)
- NO If Change actions will be honored
- NO patterns will be searched for String Slots
- NXP_DESC_STR fields being volunteered to integer or long slots will be converted by "atoi()" or "atol()", and are subject to the same restrictions.
- NXP_DESC_STR fields being volunteered to float or double slots will be converted by "atof()" and are subject to the same restrictions.
- NXP_DESC_VALUE is not supported (but this doesn't have to be true).

Notes

You have to use the NXP_DateRec structure to store the date information to use with Volunteer and the priority NXP_VSTRAT_NOCHECK:

```
typedef struct NXP_DateRec
{
  unsigned int Month;
  unsigned int Day;
  unsigned int Year;
  unsigned int Hour;
  unsigned int Min;
  unsigned int Sec;
} NXP_DateRec, *NXP_DatePtr;
```

Return Codes

NXP_VolunteerArray returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero
NXP_ERR_INVARG2	Neither NXP_DESC_STR not NXP_DESC_ATOM was passed for type.
NXP_ERR_INVARG3	One of the atomids or atom names in the list pointed to by atoms was invalid. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG4	One of the desc in the list pointed to by descs is invalid. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG5	One of the ptr in the list pointed to by ptrs is invalid, or a conversion problem occurred with the data passed. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG6	One of the strategies in the list pointed to by strats is invalid. Call NXP_ErrorIndex() to find out which atom is invalid.

Examples

The following example shows how to volunteer string slots and a date slot.

```
/* Example of NXP_VolunteerArray */
typedef struct _NXP_DateRec {
       int    Month;
       int    Day;
       int    Year;
       int    Hour;
       int    Min;
       int    Sec;
}NXP_DateRec;
static NXP_DateRec DateA;
static Char *S_ValuesA[]= { "this is a string", "this is string 2", "1234",
                            "FALSE", "12.5", (Str)&DateA};
int    i;
int    desc[6], strats[6];
char   SlotNames[6];
for (i=0;i<6;i++) {
descs[i]= NXP_DESC_STR;
strats[i]= NXP_VSTRAT_NOCHECK;
}
descs[5]= NXP_DESC_DATE;
DateA.Month = 11;  DateA.Day = 12;  DateA.Year = 91;
DateA.Hour = 0; DateA.Min = 0; DateA.Sec = 0;
SlotNames[0]= "testobj.propstr";
SlotNames[1]= "testobj.propstr1";
SlotNames[2]= "testobj.propint";
SlotNames[3]= "testobj.propbool";
SlotNames[4]= "testobj.propfloat";
SlotNames[5]= "testobj.propdate";
NXP_VolunteerArray(6,NXP_DESC_STR,(VoidPtr) SlotNames,descs,
                   S_ValuesA, strats);
```

See Also

`NXP_VolunteerList`	Volunteering a list of values.
`NXP_Volunteer`	Volunteering a single value.

NXP_VolunteerList

Purpose

This allows you to group in one call the volunteer of a couple of data items of different types.

C Format

The C format is as follows:

int NXP_VolunteerList(*count***,** *type***,** *atom1***,** *desc1***,** *ptr1***,** *strat1***,** *atom2***,** *desc2***,** *ptr2***,** *strat2***, ...);**

Arguments

The following list shows the valid arguments:

```
int          count;
int          type;
atomid       atomx;
int          descx;
```

```
VoidPtr      ptrx;
int          stratx;
```

count is the number of atoms to be volunteered.

type indicates whether atomx is an array of AtomIds or an array of atom names. Specify either NXP_DESC_ATOM or NXP_DESC_STR.

atomx is the AtomId, or pointer to the name of an atom, to be volunteered, depending on whether NXP_DESC_ATOM or NXP_DESC_STR was passed.

*descx t*he descriptor for the data being passed -- NXP_DESC_INT, NXP_DESC_STR, etc. See NXP_DESC_DATE below for information on a new date descriptor.

Code	Description
NXP_DESC_DATE	This is a new descriptor to speed up the volunteering (and retrieval) of dates in the Rules Element. It describes an array of 6 integers and contains Month, Day, Year (19xx), Hour, Minute, and Second. Obviously, patterns are not involved. Internally, the Rules Element stuffs the integers into a DateRec and calls DateFix() to fill in the day-of-week stuff.

ptrx points to the data to be volunteered.

stratx is the strategy for the atom to be volunteered. See NXP_VSTRAT_NOCHECK below for information on a new volunteering strategy.

Code	Description
NXP_VSTRAT_NOCHECK	This is a NEW strategy which can be used with NXP_VolunteerArray and NXP_VolunteerList to speed up volunteering information into the Rules Element. It is much like NXP_VSTRAT_SET, except that it has other restrictions:

- NO notifications for the change (especially transcript)
- NO If Change actions will be honored
- NO patterns will be searched for String Slots
- NXP_DESC_STR fields being volunteered to integer or long slots will be converted by "atoi()" or "atol()", and are subject to the same restrictions.
- NXP_DESC_STR fields being volunteered to float or double slots will be converted by "atof()" and are subject to the same restrictions.
- NXP_DESC_VALUE is not supported (but this doesn't have to be true).

Notes

You have to use the NXP_DateRec structure to store the date information to use with Volunteer and the priority NXP_VSTRAT_NOCHECK:

```
typedef struct NXP_DateRec
{
  unsigned int Month;
  unsigned int Day;
  unsigned int Year;
  unsigned int Hour;
  unsigned int Min;
```

```
  unsigned int Sec;
} NXP_DateRec, *NXP_DatePtr;
```

Return Codes

NXP_VolunteerList returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	count was invalid - less than zero
NXP_ERR_INVARG2	Neither NXP_DESC_STR not NXP_DESC_ATOM was passed for type.
NXP_ERR_INVARG3	atomi was invalid or the wrong type. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG4	desci was invalid. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG5	ptri was invalid or a conversion problem occurred with the data passed. Call NXP_ErrorIndex() to find out which atom is invalid.
NXP_ERR_INVARG6	strati was invalid. Call NXP_ErrorIndex() to find out which atom is invalid.

Examples

The following example shows how to volunteer 6 slots of different types provided in a list.

```
/* Example of NXP_VolunteerList */
static NXP_DateRec DateB;
static char   *S_ValuesB[]= {"this 2 is a string", "this 2 is string 2",
"1235", "TRUE", "123.5", (char *)&DateB};
int    MyExecute( char *theStr, int nAtoms, AtomId *theAtoms)
{
int          i;
int          theDescs[6], strats[6];
if (nAtoms != 6) return 0;
for (i=0;i<6;i++) {
theDescs[i]= NXP_DESC_STR;
strats[i]= NXP_VSTRAT_NOCHECK;
}
theDescs[5]= NXP_DESC_DATE;
DateB.Month = 12; DateB.Day = 12; DateB.Year = 91;
DateB.Hour = 0;  DateB.Min = 0;  DateB.Sec = 0;
NXP_VolunteerList(6,NXP_DESC_ATOM,
       theAtoms[0],theDescs[0],(VoidPtr)S_ValuesB[0],strats[0],
       theAtoms[1],theDescs[1],(VoidPtr)S_ValuesB[1],strats[1],
       theAtoms[2],theDescs[2],(VoidPtr)S_ValuesB[2],strats[2],
       theAtoms[3],theDescs[3],(VoidPtr)S_ValuesB[3],strats[3],
       theAtoms[4],theDescs[4],(VoidPtr)S_ValuesB[4],strats[4],
       theAtoms[5],theDescs[5],(VoidPtr)S_ValuesB[5],strats[5]);
}
```

See Also

NXP_VolunteerArray	Volunteering an array of values.
NXP_Volunteer	Volunteering a single value.

NXP_WalkNodes

Purpose

NXP_WalkNodes allows you to apply a user-defined function at each node along the inheritance links of an atom.

NXP_WalkNodes goes through the inheritance links starting at atom, up or down depending on code, and calls the function fct at each node with three arguments: arg, node, and property.

C Format

The C format is as follows.

int NXP_WalkNodes(*atom***,** *code***,** *fct***,** *arg***);**

Arguments

The following list shows the valid arguments.

```
AtomId        atom;
int           code;
NxpIProc      fct;
unsigned long arg;
```

atom must be a valid slot, object, or class Id. It is the starting point of the WalkNode algorithm.

code must be NXP_WALKN_UP or NXP_WALKN_DOWN. UP means going through the parent links, and DOWN means going through the children links.

arg is an unsigned long argument that will be passed unchanged to function fct. You can use it to store some custom information (pointer to a buffer, to a structure, etc.).

fct is the address of a function returning an integer that will be called at each node along the links. It is not called when node = atom. It has the following syntax:

```
int    fct( arg, node, property)
unsigned longa      rg;
AtomId              node;
AtomId              property;
```

arg is the argument passed to NXP_WalkNodes. node is the atomId of the current node (it is the only argument that changes every time fct is called). property is the property Id of atom if atom is a slot, or NULL if atom is an object of a class.

Your function fct must return one of the following:

Code	Description
NXP_WALKN_CONT	Returned if you want the inheritance to continue.
NXP_WALKN_END	Returned to terminate the inheritance immediately.
NXP_WALKN_STOP	Returned to stop the inheritance along the current link.

Warning

For internal reasons, it is dangerous to call NXP_Volunteer within fct. If you do so, do not use the priority masks NXP_VSTRAT_SET or

NXP_VSTRAT_SETQUEUE that force the value to be set immediately. The default mask, NXP_VSTRAT_QUEUE is fine. See NXP_Volunteer for more information.

Return Codes

NXP_WalkNodes returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	atom is not a valid slot, object, or class Id.
NXP_ERR_INVARG2	code is not equal to NXP_WALKN_UP or NXP_WALKN_DOWN.
NXP_ERR_INVARG3	fct is NULL.

Examples

The following examples illustrate how to get the names of all the children objects of a class and display them in the Transcript:

```
/* Function called at each node.
 * checks that the node is an object and displays its name.
 */
int  getName(arg, node, property)
unsigned long   arg;       /* ignored here */
AtomId          node;
AtomId          property;  /* ignored here */
{
       int   theType;
       char  theName[255];

       /* Get the type of node and return if its not an object */
       NXP_GETINTINFO( node, NXP_AINFO_TYPE, &theType);
       theType = theType & NXP_ATYPE_MASK;

       if( theType != NXP_ATYPEOBJECT )
              return NXP_WALKN_CONT;

       /* Get the name of the object */
       NXP_GETNAME( node, theName, 255 );

       /* Write it in Transcript */
       NXP_SetData( NXP_WIN_TRAN, 0, -1, theName);

       /* Let it go to the next node */
       return NXP_WALKN_CONT;
}


NXP_WalkNodes( theClass, NXP_WALKN_DOWN, getName, (unsigned
long) 0);
```

NXPGFX_Control

Purpose

NXPGFX_Control controls the interactive interface of the Rules Element. It is implemented in the development versions of Unix and VAX/VMS versions (in the runtime library, NXPGFX_Control is just a stub). On the PC development version, it is always automatically enabled, while with the embedded DLL, and Mac NDL, this call is just a stub.

C Format

The C format is as follows.

int NXPGFX_Control(*code***);**

Arguments

The following list shows the valid arguments.

```
int    code;
```

code can be one of the following values:

Code	Description
NXPGFX_CTRL_EXIT	Exits the interactive interface. The NXPGFX_Control call which started the interface (NXPGFX_CTRL_STARTKNOWCESS or NXPGFX_CTRL_START) will return to its caller. All the windows currently opened will be closed.
NXPGFX_CTRL_INIT	Initializes the Rules Element's interface. This call should be done before any other NXPGFX_Control call.
NXPGFX_CTRL_START	Starts the interactive interface. The call will return when the user selects the "Quit" command in the interface or after a call to NXPGFX_Control with the NXPGFX_CTRL_EXIT code (i.e. issued from an execute routine).
NXPGFX_CTRL_STARTKNOWCESS	Starts the interactive interface and starts the inference engine. The call will return when the user selects the "Quit" command in the interface or after a call to NXPGFX_Control with the NXPGFX_CTRL_EXIT code (i.e. issued from an execute routine).

Return Codes

NXPGFX_Control returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the code which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	code is invalid.
NXP_ERR_INVSTATE	code is NXPGFX_CTRL_START or NXPGFX_CTRL_STARTKNOWCESS but the interface was already started.
NXP_ERR_NOERR	Call was successful.

Examples

The following code is in the Rules Element `.c` file of the development version, that's how the Rules Element is launched!

```
/* Initialize and start the graphics development environment */
NXPGFX_Control(NXPGFX_CTRL_INIT);
NXPGFX_Control(NXPGFX_CTRL_START);
```

Chapter

4 NXP_GetAtomInfo Routine

This chapter describes the NXP_GetAtomInfo routine and the information codes you can call with it.

NXP_GetAtomInfo

Purpose

NXP_GetAtomInfo is a multi-purpose call giving access to any type of information attached to an atom or a knowledge base. Almost everything visible in the development system interface can be returned by this function.

The information returned is what is stored in the Rules Element working memory. NXP_GetAtomInfo is a "read-only" call, it doesn't modify anything nor trigger any inheritance or inference mechanisms.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *code***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId   theAtom;
int      code;
AtomId   optAtom;
int      optInt;
int      desc;
Str     thePtr;
int      len;
```

theAtom specifies the atom or knowledge base you want information about. theAtom is an atomId obtained by a previous call to NXP_GetAtomId, by another call to NXP_GetAtomInfo or received as argument in an Execute routine.

code specifies which type of information is requested. The different values for code will be described in detail in this chapter. See a short description and a list by categories in the following pages.

optAtom is an additional argument with different meanings depending on the value of code.

optInt is an additional argument with different meanings depending on the value of code.

desc is a code which describes the return data type expected by the caller (pointed to by thePtr). It must be one of the NXP_DESC_XXX codes defined in `nxpdef.h`: NXP_DESC_INT, NXP_DESC_FLOAT, NXP_DESC_DOUBLE, NXP_DESC_STR, NXP_DESC_ATOM, etc.

thePtr should point to a valid memory location where the information will be returned.

len is the maximum number of characters that can be returned in thePtr when it is pointing to a string (desc = NXP_DESC_STR). len is not used otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. More information about errors can be obtained by calling NXP_Error() immediately after the failed call. Error codes returned by NXP_Error() take the values NXP_ERR_INVARG1, NXP_ERR_INVARG2, ..., NXP_ERR_INVARG6 to show invalid arguments. More information on these values can be found within each call description.

C Macros

In many cases NXP_GetAtomInfo calls can be simplified with the help of the macros defined in `nxpdef.h`. Instead of passing all seven arguments you just pass what is really used by the call. It also prevents making mistakes! For instance getting the integer value of an atom is the simple call

```
NXP_GETINTVAL(atom, ptr)
```

instead of

```
NXP_GetAtomInfo(atom, NXP_AINFO_VALUE, (AtomId)0, 0,
                    NXP_DESC_INT, ptr, 0)
```

Following the description of each information code, we will indicate which macro can be used instead of an NXP_GetAtomInfo call. We also use macros in the examples, they are easy to spot with their name written in capital letters. See the section "NXP_GetAtomInfo macros" for the list of macros.

Macintosh Warning

When using the THINK C environment on the Macintosh, int should be replaced by long . THINK C handles int as short integer (16 bits) whereas the Rules Element uses 32-bit integers (int and long are equivalent in MPW C). Therefore, the arguments of NXP_GetAtomInfo code, optInt, desc, and len must be declared as long and numeric constants must end with L, such as in 0L. This applies to all other API functions as well. See the Macintosh API manual for more information.

See Also

NXP_SetAtomInfo	Changing information in an atom or KB.
NXP_GetAtomId	Getting the AtomId of an atom from its name.

Information Codes List

Following is the complete list of NXP_GetAtomInfo codes in alphabetical order. Each code is described in detail in the following sections.

Code	Short Description
NXP_AINFO_AGDVBREAK	Returns whether a specific hypothesis has an agenda break point set for it.
NXP_AINFO_BREADTHFIRST	Returns whether the inheritance search for theAtom is done in a breadth first or depth first manner.
NXP_AINFO_BWRDLINKS	Returns the backward links from a hypothesis to its rules.
NXP_AINFO_CACTIONS	Returns the text of the If Change actions attached to theAtom.
NXP_AINFO_CACTIONSON	Returns whether or not If Change actions are enabled.
NXP_AINFO_CACTIONSUNKNOWN	Returns whether or not If Change methods will also be executed when the slot is set to UNKNOWN.
NXP_AINFO_CURRENTKB	Returns the atomid of the current knowledge base containing a specified atom.
NXP_AINFO_CHILDCLASS	Returns information about the children classes of a class.
NXP_AINFO_CHILDOBJECT	Returns information about the children objects of a class or object.
NXP_AINFO_CHOICE	Returns the choice of values (as displayed in the session control window) for a given slot.
NXP_AINFO_CLIENTDATA	Returns the client or user information attached to an atom.
NXP_AINFO_COMMENTS	Returns the comments attached to theAtom.
NXP_AINFO_CONTEXT	Returns the hypotheses that are in the context of a given hypothesis.
NXP_AINFO_CURRENT	Returns information about the current atom ids in the inference engine.
NXP_AINFO_DEFAULTFIRST	Returns whether or not the inheritance strategy for theAtom follows the default (global strategy).
NXP_AINFO_DEFVAL	Returns the init value for the slot (if any) as a string.
NXP_AINFO_EHS	Returns information about the Else actions (Right-hand side) of a rule or a method.
NXP_AINFO_EXHBWRD	Returns whether or not exhaustive backward chaining is enabled.
NXP_AINFO_FOCUSPRIO	Returns the priority of the hypothesis on the agenda.
NXP_AINFO_FORMAT	Returns the format information attached to theAtom.
NXP_AINFO_FWRDLINKS	Returns the forward links from a slot to the conditions.
NXP_AINFO_HASMETA	Returns whether or not a slot has meta-slots defined for it.
NXP_AINFO_HYPO	Returns the hypothesis of a rule.
NXP_AINFO_INFATOM	Returns the inference priority atom attached to theAtom.
NXP_AINFO_INFBREAK	Returns whether a specific rule, condition, method, slot, object, class, or property has an inference break point set on it.

Code	Short Description
NXP_AINFO_INFCAT	Returns the inference priority attached to theAtom.
NXP_AINFO_INHATOM	Returns the inheritance priority atom attached to theAtom.
NXP_AINFO_INHCAT	Returns the inheritance priority attached to theAtom.
NXP_AINFO_INHCLASSDOWN	Returns whether or not class slots are inheritable downwards.
NXP_AINFO_INHCLASSUP	Returns whether or not class slots are inheritable upwards.
NXP_AINFO_INHDEFAULT	Returns whether or not the slot inheritability of theAtom follows the default (global strategy).
NXP_AINFO_INHDOWN	Returns whether or not the slot theAtom is downward inheritable.
NXP_AINFO_INHOBJDOWN	Returns whether or not object slots are inheritable downwards.
NXP_AINFO_INHOBJUP	Returns whether or not object slots are inheritable upwards.
NXP_AINFO_INHUP	Returns whether or not the slot theAtom is upward inheritable.
NXP_AINFO_INHVALDEFAULT	Returns whether or not the inheritability of the value of theAtom follows the default (global strategy).
NXP_AINFO_INHVALDOWN	Returns whether or not the value of theAtom is downward inheritable.
NXP_AINFO_INHVALUP	Returns whether or not the value of theAtom is upward inheritable.
NXP_AINFO_KBID	Returns the identifier of the knowledge base to which the atom belongs.
NXP_AINFO_KBNAME	Returns the name of a knowledge base from its atomid.
NXP_AINFO_LHS	Returns information about the conditions of a rule or method.
NXP_AINFO_LINKED	Returns information about the type of link between a class or an object and another class or object.
NXP_AINFO_METHODS	Returns the list of methods attached to theAtom.
NXP_AINFO_MOTSTATE	Returns information about the current state of the inference engine.
NXP_AINFO_NAME	Returns the string name of the atom described by theAtom.
NXP_AINFO_NEXT	Returns information about the lists of atoms stored in the working memory.
NXP_AINFO_PARENT	Returns information about the parent of an atom.
NXP_AINFO_PARENTCLASS	Returns information about the parent classes of a class or an object.
NXP_AINFO_PARENTFIRST	Returns whether the inheritance search for theAtom should begin by searching the parent objects of theAtom or the classes to which theAtom belongs.
NXP_AINFO_PARENTOBJECT	Returns information about the parent objects of an object.
NXP_AINFO_PFACTIONS	Returns whether or not the assignments done in the RHS of rules or in method actions are forwarded.
NXP_AINFO_PFELSEACTIONS	Returns the value to which the forward Else actions strategy is set.
NXP_AINFO_PFMETHODACTIONS	Returns the value to which the forward LHS/RHS actions from methods strategy is set.
NXP_AINFO_PFMETHODELSEACTIONS	Returns the value to which the forward Else actions from methods strategy is set.
NXP_AINFO_PREV	Returns information about the lists of atoms stored in the working memory.

Code	Short Description
NXP_AINFO_PROCEXECUTE	Returns the number of execute routines installed or the name of the execute handler.
NXP_AINFO_PROMPTLINE	Returns the prompt line information attached to theAtom.
NXP_AINFO_PROP	Returns the property id of the slot in thePtr.
NXP_AINFO_PTGATES	Returns whether or not forward chaining through gates is enabled.
NXP_AINFO_PWFALSE	Returns whether or not the context propagation is enabled on FALSE hypotheses.
NXP_AINFO_PWNOTKNOWN	Returns whether or not the context propagation is enabled on NOTKNOWN hypotheses.
NXP_AINFO_PWTRUE	Returns whether or not the context propagation is enabled on TRUE hypotheses.
NXP_AINFO_QUESTWIN	Returns the Smart Elements question window name attached to theAtom.
NXP_AINFO_RHS	Returns information about the true actions (Right-hand side) of a rule or method.
NXP_AINFO_SELF	Returns the name or atomId of the current SELF atom.
NXP_AINFO_SLOT	Returns information about the slots of a class or an object.
NXP_AINFO_SOURCES	Returns the text of the Order of Sources methods attached to theAtom.
NXP_AINFO_SOURCESCONTINUE	Returns whether or not Order of Sources methods will be fully executed even after a value is determined.
NXP_AINFO_SOURCESON	Returns whether or not Order of Sources methods are enabled.
NXP_AINFO_SUGGEST	Returns whether or not a hypothesis is suggested.
NXP_AINFO_SUGLIST	Returns the list of hypotheses kept in the suggest selection.
NXP_AINFO_TYPE	Returns the type of theAtom in thePtr.
NXP_AINFO_VALIDENGINE_ACCEPT	Returns whether or not the validation of value set by the engine is enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDENGINE_OFF	Returns whether or not the validation of value set by the engine is disabled.
NXP_AINFO_VALIDENGINE_ON	Returns whether or not the validation of value set by the engine is enabled.
NXP_AINFO_VALIDENGINE_REJECT	Returns whether or not the validation of value set by the engine is enabled and the value rejected automatically if the validation expression is incomplete.
NXP_AINFO_VALIDEXEC	Returns the validation external routine name attached to theAtom.
NXP_AINFO_VALIDFUNC	Returns the validation expression string attached to theAtom.
NXP_AINFO_VALIDHELP	Returns the validation error string attached to theAtom.
NXP_AINFO_VALIDUSER_ACCEPT	Returns whether or not the validation of value entered by the end user is enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDUSER_OFF	Returns whether or not the validation of value entered by the end user is disabled.
NXP_AINFO_VALIDUSER_ON	Returns whether or not the validation of value entered by the end user is enabled.
NXP_AINFO_VALIDUSER_REJECT	Returns whether or not the validation of value entered by the end user is enabled and the value rejected automatically if the validation expression is incomplete.

Code	Short Description
NXP_AINFO_VALUE	Returns information in thePtr about the value of theAtom.
NXP_AINFO_VALUETYPE	Returns information in thePtr about the data type of the value of theAtom.
NXP_AINFO_VERSION	Returns the names and version numbers of the software components included in the package used.
NXP_AINFO_VOLLIST	Returns the list of slots kept in the volunteer selection.
NXP_AINFO_WHY	Returns the why information attached to theAtom.

Information Codes by Categories

The pieces of information which can be obtained with the NXP_GetAtomInfo code can be categorized as follows:

Generic atom information:

NXP_AINFO_CLIENTDATA	Client or user data associated with an atom
NXP_AINFO_CURRENTKB	Atomid of the kb associated with an atom
NXP_AINFO_KBID	Knowledge base to which the atom is attached
NXP_AINFO_KBNAME	KB name to which the atom is attached
NXP_AINFO_NAME	Atom name
NXP_AINFO_TYPE	Atom type (object, class, property, slot, rule, method, ...)
NXP_AINFO_VALUE	Value of a slot, rule or condition
NXP_AINFO_VALUETYPE	Data type of a slot (boolean, integer, float, . . .)

Global lists of atoms:

NXP_AINFO_NEXT	Next atom in a list (next object, data, rule, . . .)
NXP_AINFO_PREV	Previous atom in a list
NXP_AINFO_SUGLIST	List of hypotheses selected for Suggest

Pre-suggested hypotheses:

NXP_AINFO_VOLLIST	List of data slots selected for Volunteer

Relations between atoms:

NXP_AINFO_CHILDCLASS	Children classes of a class
NXP_AINFO_CHILDOBJECT	Children objects of an object or a class
NXP_AINFO_LINKED	Type of link between theAtom and optAtom
NXP_AINFO_PARENT	Parent object or parent class of a slot
NXP_AINFO_PARENTCLASS	Parent classes of an object or a class
NXP_AINFO_PARENTOBJECT	Parent objects of an object
NXP_AINFO_PROP	Property corresponding to a slot
NXP_AINFO_SLOT	Slots of an object or a class

Rule and context information:

NXP_AINFO_CONTEXT	List of hypos in context of another hypo
NXP_AINFO_COMMENTS	Comments associated with a slot, rule, or method.
NXP_AINFO_EHS	Else actions of a rule (right-hand side).

Rule and context information:

NXP_AINFO_HYPO	Hypothesis of a rule
NXP_AINFO_LHS	Conditions of a rule (Left-hand-side)
NXP_AINFO_RHS	True actions of a rule (Right-hand-side)
NXP_AINFO_WHY	Why text associated with a slot, rule, or method.

Meta-slots - General:

NXP_AINFO_HASMETA	Whether or not a slot has meta-slots defined

Meta-slots - Categories:

NXP_AINFO_INFATOM	Inference atom of a slot or a rule
NXP_AINFO_INFCAT	Inference priority of a slot or a rule
NXP_AINFO_INHATOM	Inheritance atom (dynamic priority) of a slot
NXP_AINFO_INHCAT	Inheritance priority of a slot

Meta-slots - Inheritability settings:

NXP_AINFO_INHDEFAULT	Does the inheritability of a slot follow the default?
NXP_AINFO_INHDOWN	Downward inheritability of a slot
NXP_AINFO_INHUP	Upward inheritability of a slot
NXP_AINFO_INHVALDEFAULT	Does the inheritability of a slot's value follow the default?
NXP_AINFO_INHVALDOWN	Downward inheritability of a slot's value
NXP_AINFO_INHVALUP	Upward inheritability of a slot's value

Meta-slots - Inheritance strategy:

NXP_AINFO_BREADTHFIRST	Breadth first versus depth first strategy
NXP_AINFO_DEFAULTFIRST	Does the inheritance strategy for a slot follow the default?
NXP_AINFO_PARENTFIRST	Object first versus class first strategy

Meta-slots - Text information:

NXP_AINFO_CHOICE	Set of possible values (string slot only)
NXP_AINFO_COMMENTS	Comments associated with a slot, rule, or method
NXP_AINFO_DEFVAL	Init value of a slot (public or private)
NXP_AINFO_FORMAT	Format information for a slot or a property
NXP_AINFO_PROMPTLINE	Prompt line associated with a slot
NXP_AINFO_QUESTWIN	Smart Elements question window name attached to theAtom
NXP_AINFO_VALIDEXEC	Validation external routine name attached to theAtom
NXP_AINFO_VALIDFUNC	Validation expression string attached to theAtom
NXP_AINFO_VALIDHELP	Validation error string attached to theAtom
NXP_AINFO_WHY	Why text associated with a slot, rule, or method

Methods:

NXP_AINFO_CACTIONS	If Change actions of a slot (public or private).
NXP_AINFO_COMMENTS	Comments associated with a slot, rule, or method.
NXP_AINFO_EHS	Else actions of method (Right-hand side).
NXP_AINFO_LHS	Conditions of method (Left-hand-side).

Methods:

NXP_AINFO_METHODS	List of methods attached to an atom
NXP_AINFO_RHS	True actions of method (Right-hand-side).
NXP_AINFO_SOURCES	Order of Sources of a slot (public or private).
NXP_AINFO_WHY	Why text associated with a slot, rule, or method.

Inference state:

NXP_AINFO_CURRENT	Current rule, condition, action or slot being evaluated
NXP_AINFO_MOTSTATE	State of the inference engine ("motor").
NXP_AINFO_SUGGEST	Whether or not a hypothesis is suggested

Inference strategies:

NXP_AINFO_EXHBWRD	Exhaustive evaluation of backward chaining
NXP_AINFO_PFACTIONS	Forwarding on actions
NXP_AINFO_PFELSEACTIONS	Value to which the forward Else actions strategy is set.
NXP_AINFO_PFMETHODACTIONS	Value to which the forward LHS / RHS actions from methods strategy is set.
NXP_AINFO_PFMETHODELSEACTIONS	Value to which the forward Else actions from methods strategy is set.
NXP_AINFO_PTGATES	Forwarding through gates
NXP_AINFO_PWFALSE	Context propagation on FALSE hypo
NXP_AINFO_PWNOTKNOWN	Context propagation on NOTKNOWN hypo
NXP_AINFO_PWTRUE	Context propagation on TRUE hypotheses
NXP_AINFO_VALIDENGINE_ACCEPT	Whether or not the validation of value set by the engine is enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDENGINE_OFF	Whether or not the validation of value set by the engine is disabled.
NXP_AINFO_VALIDENGINE_ON	Whether or not the validation of value set by the engine is enabled.
NXP_AINFO_VALIDENGINE_REJECT	Whether or not the validation of value set by the engine is enabled and the value rejected automatically if the validation expression is incomplete.
NXP_AINFO_VALIDUSER_ACCEPT	Whether or not the validation of value entered by the end user is enabled and the value accepted automatically if the validation expression is incomplete.
NXP_AINFO_VALIDUSER_OFF	Whether or not the validation of value entered by the end user is disabled.
NXP_AINFO_VALIDUSER_ON	Whether or not the validation of value entered by the end user is enabled.
NXP_AINFO_VALIDUSER_REJECT	Whether or not the validation of value entered by the end user is enabled and the value rejected automatically if the validation expression is incomplete.

Inheritance strategies:

NXP_AINFO_BREADTHFIRST	Breadth first versus depth first strategy
NXP_AINFO_CACTIONSON	If Change enabled or not.
NXP_AINFO_CACTIONSUNKNOWN	If Change will be executed when the slot is set to UNKNOWN.
NXP_AINFO_INHCLASSDOWN	Downward inheritability of class slots.

Inheritance strategies:

NXP_AINFO_INHCLASSUP	Upward inheritability of class slots.
NXP_AINFO_INHOBJDOWN	Downward inheritability of object slots.
NXP_AINFO_INHOBJUP	Upward inheritability of object slots.
NXP_AINFO_INHVALDOWN	Downward inheritability of the value of a slot.
NXP_AINFO_INHVALUP	Upward inheritability of the value of a slot.
NXP_AINFO_PARENTFIRST	Parent first versus class first strategy.
NXP_AINFO_SOURCESCONTINUE	Order of Sources will be fully executed.
NXP_AINFO_SOURCESON	Order of Sources enabled or not.

Agenda and break-points information:

NXP_AINFO_AGDVBREAK	Whether a specific hypothesis has an agenda break point set on it.
NXP_AINFO_BWRDLINKS	The backward links from a hypo to its rules.
NXP_AINFO_FOCUSPRIO	The priority of the hypothesis on the agenda.
NXP_AINFO_FWRDLINKS	The forward links from a slot to the conditions.
NXP_AINFO_INFBREAK	Whether a specific rule, condition, method, slot, object, class, or property has an inference break point set on it.

SELF atom information:

NXP_AINFO_SELF	Information about the current SELF atom.

Execute handler information:

NXP_AINFO_PROCEXECUTE	The number of Execute procedures installed or the name of the Execute Handler.

Software components information:

NXP_AINFO_VERSION	Names and version numbers of the software components included in the package used.

These information codes are described in detail in the following sections.

NXP_GetAtomInfo Macros

The NXP_GetAtomInfo routine takes seven arguments and, in most cases, several arguments are unused. To facilitate program development, C macros are provided in the `nxpdef.h` header file. The macros are the following:

General Purpose Macros

Use the macros of type `NXP_GETXXXINFO` when *optAtom* and *optInt* are not used:

Macro	Description
NXP_GETATOMINFO	Returns information of type Atom in thePtr. (atom, code, ptr)
NXP_GETINTINFO	Returns information of type int in thePtr. (atom, code, ptr)

Macro	Description
NXP_GETDOUBLEINFO	Returns information of type double in thePtr. (atom, code, ptr)
NXP_GETSTRINFO	Returns information of type string in thePtr, limited by the length len. (atom,code,ptr,len)

Get Values Macros

Use the macros of type NXP_GETXXXVAL to get values of a specific type:

Macro	Description
NXP_GETINTVAL	Returns an int value in thePtr. (atom, ptr)
NXP_GETDOUBLEVAL	Returns a double value in thePtr. (atom, ptr)
NXP_GETSTRVAL	Returns a string value in thePtr. (atom, ptr)
NXP_GETUNKNOWNVAL	Tells if the value is Unknown. (atom, ptr)
NXP_GETNOTKNOWNVAL	Tells if the value is Notknown. (atom, ptr)

Get Name Macro

Use the following macro to get names of atoms in thePtr:

Macro	Description
NXP_GETNAME	Returns the name of the atom in thePtr, limited by the length len. (atom, ptr, len)

List Access Macros

Use the macros of type NXP_GETLISTXXX to access lists of atoms (for example, the list of children objects in a class):

Macro	Description
NXP_GETLISTLEN	Returns the length of the list described by code. (atom, code, ptr)
NXP_GETLISTELT	Returns the atom element of index i in the list. (atom, code, i, ptr)
NXP_GETLISTELTSTR	Returns the string element of index i in the list, (atom,code,i,ptr,len) limited by length len.
NXP_GETLISTFIRST	Returns the first atom of type type. (type, ptr)
NXP_GETLISTNEXT	Returns the next element after atom in the list. (atom, type, ptr)

Following the description of each information code, we will indicate which macro can be used instead of an NXP_GetAtomInfo call.

NXP_GetAtomInfo / NXP_AINFO_AGDVBREAK

Purpose

This returns whether a specific hypothesis has an agenda break point set.

Agenda break points can be set with the NXP_SetAtomInfo function or through the agenda monitor window in the interface. They stop the engine when the focus of a hypothesis changes (whereas inference break-points stop the engine after the evaluation of an atom).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_AGDVBREAK***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_AGDVBREAK */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid hypothesis.

code is equal to NXP_AINFO_AGDVBREAK.

desc must equal NXP_DESC_INT.

thePtr must point to an integer. *thePtr will be set to 1 if a break point is set, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid hypothesis.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO macro:

```
NXP_GETINTINFO(theHypo, NXP_AINFO_AGDVBREAK, thePtr)
```

Examples

The following example illustrates how to automatically unset all the break points in your knowledge base that have been set through the Development interface. A rule could call an Execute routine containing this code:

```
int      hasBreak;
AtomId   firstHypo, theHypo, nextHypo;

/* Get the first hypothesis in alphabetical order */
NXP_GETLISTFIRST(NXP_ATYPE_HYPO, &firstHypo);

/* Loop to get the following hypotheses */
if(firstHypo != NULL) {
       theHypo = firstHypo;
       while(theHypo != NULL) {
              /* Check if theHypo has an agenda break point set */
              /* If yes, use NXP_SetAtomInfo to unset it */
              NXP_GETINTINFO(theHypo, NXP_AINFO_AGDVBREAK,
                                  &hasBreak);
              if(hasBreak) {
                     NXP_SetAtomInfo(theHypo, NXP_SAINFO_AGDVBREAK,
                                         (AtomId)0, 0, 0, 0);
              }
              /* Get the next Hypo in the list */
              NXP_GETLISTNEXT(theHypo, NXP_ATYPE_HYPO, &nextHypo);
              theHypo = nextHypo;
       }
}
```

See Also

`NXP_SetAtomInfo / NXP_SAINFO_AGDVBREAK`	Set / unset break points from a program.
`NXP_AINFO_INFBREAK`	Information on inference break points.

NXP_GetAtomInfo / NXP_AINFO_BREADTHFIRST

Purpose

This returns whether the inheritance search is done in a breadth first or depth first manner. It can be used to get the global strategy (current or default) or the strategy for a particular atom.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_BREADTHFIRST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid slot or value id, or NULL. If theAtom is NULL, the call returns the global strategy setting for breadth first versus depth first inheritance. If theAtom is not NULL, the call returns the strategy for that atom.

code must be NXP_AINFO_BREADTHFIRST if theAtom is not NULL or if it is NULL and you want to get the global default strategy. code must be NXP_AINFO_BREADTHFIRST | NXP_AINFO_CURSTRAT to get the global current strategy ("Or" operation sets the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the search is breadth first, and set to 0 if the search is depth first.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_BREADTHFIRST, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   atom;
int      breadth;

/* returns the breadth-first strategy for atom in breadth */
NXP_GETINTINFO(atom, NXP_AINFO_BREADTHFIRST, &breadth);

/* returns the global default breadth-first strategy */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_BREADTHFIRST, &breadth);

/* returns the global current breadth-first strategy */
NXP_GETINTINFO((AtomId)0,
               NXP_AINFO_BREADTHFIRST|NXP_AINFO_CURSTRAT, &breadth);
```

See Also

NXP_Strategy	Change the default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_BWRDLINKS

Purpose

This returns the backward links from a hypothesis to its rules. Use this code to get the list of rules pointing to a particular.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_BWRDLINKS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId       theAtom;
int          code;     /* = NXP_AINFO_BWRDLINKS */
AtomId       optAtom;  /* ignored */
int          optInt;
int          desc;
Str          thePtr;
int          len;      /* ignored */
```

theAtom must be a valid atom id.

code must be NXP_AINFO_BWRDLINKS.

If *optInt* equals -1, *desc* must be NXP_DESC_INT and the number nRules of rules is returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and nRules -1. In this case, desc must be NXP_DESC_ATOM, thePtr must be a pointer to an *atomId* memory location, and the rule id with the index optInt in the list is returned.

Notes

The list of rules is returned in no special order. Once you have the atomId of a rule, you can get the rule name using NXP_AINFO_NAME.

If theAtom is not a hypothesis this call returns an empty list (nRules = 0).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid atom id.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT. optInt is between 0 and n-1 but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call use the NXP_GETLISTLEN(atom, code, ptr) macro to get the number nRules of rules in the list:

```
NXP_GETLISTLEN(theHypo, NXP_AINFO_BWRDLINKS, &nRules)
```

Then use the NXP_GETLISTELT(atom, code, index, ptr) macro, with indexbetween 0 and nRules-1, to get each element in the list:

```
NXP_GETLISTELT(theHypo, NXP_AINFO_BWRDLINKS, i, thePtr)
```

See Also

NXP_AINFO_FWRDLINKS	Forward links from a slot to its conditions.
NXP_AINFO_HYPO	Get the hypothesis of a rule.

NXP_GetAtomInfo / NXP_AINFO_CACTIONS

Purpose

This returns the atom ids of the If Change actions attached to a slot (methods specified as an If Change type in the method editor).

The text can be obtained later with NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CACTIONS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be valid slot id.

code must be NXP_AINFO_CACTIONS to get the public If Change actions. It must be NXP_AINFO_CACTIONS | NXP_AINFO_PRIVATE ("Or" operation sets the "private" bit) to get the private If Change actions, i.e. methods that can't be inherited (they appear with a * in the editor). It must be NXP_AINFO_CACTIONS | NXP_AINFO_MLHS to get the conditions of the If Change. It must be NXP_AINFO_CACTIONS | NXP_AINFO_MRHS to get the right-hand side (Then) actions of the If Change. And it must be NXP_AINFO_CACTIONS | NXP_AINFO_MEHS to get the Else actions of the If Change.

If *optInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number nActions of If Change actions is returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and nActions -1. In this case, desc should be equal to NXP_DESC_ATOM and the If Change action with the index optInt is returned in thePtr.

Once you have the *atomId* of an action you get get its text with calls to NXP_GetAtomInfo / NXP_AINFO_NAME. See the example given for NXP_AINFO_LHS.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG4	The value of optInt is not between -1 and n-1.
NXP_ERR_INVARG5	optInt equals -1 but desc does not equal NXP_DESC_INT, or optInt does not equal -1 and desc does not equal NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_CACTIONS, &nActions)
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_CACTIONS, index, thePtr)
```

Examples

The following code gives a simple example.

```
int      i, nActions;
AtomId   atom, cActions;

/* returns the public If Change actions of atom */
NXP_GETLISTLEN(atom, NXP_AINFO_CACTIONS, &nActions);
for (i = 0; i < nActions; i++) {
       NXP_GETLISTELT(atom, NXP_AINFO_CACTIONS, i, &cActions);
       ...
}

/* returns the private If Change actions */
NXP_GETLISTLEN(atom, NXP_AINFO_CACTIONS|NXP_AINFO_PRIVATE, &nActions);
for (i = 0; i < nActions; i++) {
       NXP_GETLISTELT(atom, NXP_AINFO_CACTIONS|NXP_AINFO_PRIVATE,
                      &cActions);
       ...
}
/* Use NXP_AINFO_NAME to get the text of the methods, see the example in
NXP_AINFO_LHS */
```

See Also

NXP_AINFO_LHS	Get the list of conditions of a rule or a method.
NXP_AINFO_RHS	Get the list of actions of a rule or a method.
NXP_AINFO_EHS	Get the list of Else actions of a rule or a method.
NXP_AINFO_SOURCES	Get the list of Order of Sources methods.

NXP_GetAtomInfo / NXP_AINFO_CACTIONSON

Purpose

This returns whether or not the If Change actions are enabled (Enable If Change actions option of the Strategy dialog window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CACTIONSON***,** *optAtom***,***optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

```
AtomId  theAtom;  /* ignored */
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_CACTIONSON to get the default strategy. It must be NXP_AINFO_CACTIONSON | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation sets the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the If Change actions are enabled, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO((AtomId)0, NXP_AINFO_CACTIONSON, thePtr).
```

Examples

The following code gives a simple example.

```
int   cactionson;

/* returns in cactionson the default strategy.
 * cactionson = 1 if If Change actions are enabled,
 *                        0 if they are disabled
 */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_CACTIONSON, &cactionson);

/* returns in cactionson the current strategy */
NXP_GETINTINFO((AtomId)0,
        NXP_AINFO_CACTIONSON | NXP_AINFO_CURSTRAT, &cactionson);
```

See Also

NXP_Strategy	Change the default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_CACTIONSUNKNOWN

Purpose

This returns whether or not the If Change actions are triggered also when theAtom is set to UNKNOWN (ON / UNKNOWNIf Change actions option of the Strategy dialog window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CACTIONSUNKNOWN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

```
AtomId  theAtom;  /* ignored */
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_CACTIONSUNKNOWN to get the default strategy. It must be NXP_AINFO_CACTIONSUNKNOWN | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation sets the "current" bit).

desc must equal NXP_DESC_INT.

*thePtr m*ust point to an integer which will be set to 1 if the If Change actions are triggered also when theAtom is set to UNKNOWN, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO((AtomId)0, NXP_AINFO_CACTIONSUNKNOWN, thePtr)
```

Examples

The following code gives a simple example.

```
int   cactionsunknown;

/* returns in cactionsunknown the default strategy.
 * cactionson = 1 if If Change actions are enabled,
 *                  0 if they are disabled
 */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_CACTIONSUNKNOWN,
                           &cactionsunknown);

/* returns in cactionsunknown the current strategy */
NXP_GETINTINFO((AtomId)0,
                     NXP_AINFO_CACTIONSUNKNOWN|
NXP_AINFO_CURSTRAT,
                           &cactionsunknown);
```

See Also

`NXP_Strategy`	Change the default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_CHILDCLASS

Purpose

This returns information about the children classes of a class.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CHILDCLASS***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_CHILDCLASS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid class id.

code is NXP_AINFO_CHILDCLASS.

optInt is an integer between -1 and N-1. optInt = -1 will return in thePtr the number of children classes which are numbered from 0 to N-1.

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

The mechanism used to retrieve this list uses the following sequence of calls:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number N of atoms in the list (thePtr must be a pointer to an integer).
- Then you call NXP_GetAtomInfo with optInt set to any value between 0 and N-1. The id of the (optInt+1)th atom in the list will be returned in thePtr (thePtr must be a pointer to an AtomId).

Notes

The ordering of the list is not defined and can change during a session (i.e. links to children classes are created or deleted dynamically).

This call will return permanent links that have been temporarily deleted. To check the type of link, call NXP_GetAtomInfo with NXP_AINFO_LINKED.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theAtom is NULL or is not a valid class id.
`NXP_ERR_INVARG4`	optInt is not equal to -1 or is not a valid child class index.
`NXP_ERR_INVARG5`	optInt is equal to -1 and desc is not equal to NXP_DESC_INT. `optInt is a valid child class index, but desc is not equal to NXP_DESC_ATOM.`

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_CHILDCLASS, &N).
```

Then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_CHILDCLASS, i, thePtr);
```

Examples

The following code gives a simple example.

```
AtomId   class, childclass;
int      i, N;

/* Get the number of children classes */
NXP_GETLISTLEN(class, NXP_AINFO_CHILDCLASS, &N);

/* loop to get each child class */
for (i = 0; i < N; i++) {
       NXP_GETLISTELT(class, NXP_AINFO_CHILDCLASS, i, &childclass);
       /* ...more code ...  */
}
```

NXP_GetAtomInfo / NXP_AINFO_CHILDOBJECT

Purpose

This returns information about the children objects of an object or a class.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CHILDOBJECT***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_CHILDOBJECT */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid object or class id.

code is NXP_AINFO_CHILDOBJECT.

optInt is an integer between -1 and N-1. optInt = -1 will return in thePtr the number of children objects which are numbered from 0 to N-1.

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

The mechanism used to retrieve this list uses the following sequence of calls:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number N of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with optInt set to any value between 0 and N-1. The id of the (optInt+1)th atom in the list will be returned in thePtr.

Notes

The ordering of the list is not defined and can change during a session (i.e. links to children objects are created or deleted dynamically).

This call returns permanent links of any deleted objects (ones that have been unlinked) as well as intact permanent links. To check the type of link, call NXP_GetAtomInfo with NXP_AINFO_LINKED.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid object or class id.
NXP_ERR_INVARG4	optInt is not equal to -1 or is not a valid child object index.
NXP_ERR_INVARG5	optInt is equal to -1 and desc is not equal to NXP_DESC_INT. optInt is a valid child object index, but desc is not equal to NXP_DESC_ATOM.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_CHILDOBJECT, &N)
```

Then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_CHILDOBJECT, i, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   atom, childobject;
int      i, N;

NXP_GETLISTLEN(atom, NXP_AINFO_CHILDOBJECT, &N);

/* loop to get each child object */
for (i = 0; i < N; i++) {
       NXP_GETLISTELT(atom, NXP_AINFO_CHILDOBJECT, i,
&childobject);
       /* ...  more code ...  */
}
```

NXP_GetAtomInfo / NXP_AINFO_CHOICE

Purpose

This returns the choice of values for a given slot (as displayed in the session control window during a question for that slot).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CHOICE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_CHOICE */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code is equal to NXP_AINFO_CHOICE.

If *optInt* is -1, desc should be equal to NXP_DESC_INT and the number n of choices will be returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this last case, desc should be equal to NXP_DESC_STR and the choice with the index optInt will be returned in thePtr.

thePtr is a pointer to an integer (first call) or to a buffer of length len.

If optInt is -1, *len* is ignored. Otherwise, len is the maximum number of characters which can be written in buffer thePtr.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG4	optInt is not a valid index.
NXP_ERR_INVARG5	optInt equals -1 but desc does not equal NXP_DESC_INT, or optInt does not equal -1 and desc does not equal NXP_DESC_STR.

Macros

For the first call use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_CHOICE, &N)
```

then use the NXP_GETLISTELTSTR macro to get individual elements:

```
NXP_GETLISTELTSTR(theAtom, NXP_AINFO_CHOICE, i, thePtr, len)
```

Examples

The following example shows how to get the choice list for a slot. It could be used for instance in a question handler to present the possible values in your custom interface.

```
AtomId   slot;
int      i, N;
Char     str[255];

/* Get number of choices */
NXP_GETLISTLEN(slot, NXP_AINFO_CHOICE, &N);

/* loop to get the list of string choices */
for (i = 0; i < N; i++) {
      NXP_GETLISTELTSTR(slot, NXP_AINFO_CHOICE, i, str, 255);
      ...
      /* code to display the value str */
}
```

NXP_GetAtomInfo / NXP_AINFO_CLIENTDATA

Purpose

This returns the "client" information attached to an atom, as it was set by NXP_SetClientData.

NXP_SetClientData lets you associate a longword of information with any atom (i.e. a pointer to some private data structure of your program). You can retrieve this information later with NXP_AINFO_CLIENTDATA.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CLIENTDATA***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_CLIENTDATA */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_LONG */
Str     thePtr;
int     len;      /* ignored */
```

theAtom can be any class, object, property, slot, rule, method, condition, RHS action, EHS action.

code is NXP_AINFO_CLIENTDATA.

desc must be NXP_DESC_LONG (32 bits information).

thePtr must be a valid longword pointer which will receive the longword of information.

Notes

Associating client or user information with atoms is very useful in building interface programs. For instance, you can associate with the objects or slots the address of graphic objects which will represent them on the screen. Then, you can use generic execute routines to control your display.

The client data longword is initialized with 0 when the atom is created.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or not equal to a class, object, property, slot, rule, method, condition, RHS action, EHS action.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_LONG.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

None.

Examples

The following code gives a simple example.

```
AtomId         atom;
unsigned long  info;

NXP_GetAtomInfo(atom, NXP_AINFO_CLIENTDATA, (AtomId)0, 0,
                           NXP_DESC_LONG, (Str)&info, 0);
```

See Also

`NXP_SetClientData`	Set the client data longword.

NXP_GetAtomInfo / NXP_AINFO_COMMENTS

Purpose

This returns the Comments string attached to a slot, method, or rule (string entered in the Comments field of the meta-slot editor, method editor, or rule editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_COMMENTS***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_COMMENTS */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot, method or rule id.

code is equal to NXP_AINFO_COMMENTS.

desc must be NXP_DESC_STR.

thePtr must point to a buffer of size len where the comment string will be returned.

len is the maximum number of characters that can be written to thePtr.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or not a valid rule, method or slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_STR.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_COMMENTS, thePtr, len)
```

Examples

The following code gives a simple example.

```
AtomId   atom;
Char     str[255];

NXP_GETSTRINFO(atom, NXP_AINFO_COMMENTS, str, 255);
```

NXP_GetAtomInfo / NXP_AINFO_CONTEXT

Purpose

This returns the hypotheses that are in the context of a given hypothesis (list displayed in the Context editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CONTEXT***,** *optAtom***,** *optInt***,***desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_CONTEXT */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid hypothesis slot id.

code is equal to NXP_AINFO_CONTEXT.

If *optInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number N of contexts will be returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and N-1. In this last case, desc should be equal to NXP_DESC_ATOM and the context with the index optInt will be returned in thePtr.

The *contexts* are listed in the order they appear in the context editor.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid hypothesis id.
NXP_ERR_INVARG4	optInt is not -1 nor a valid context index.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT. optInt is a valid context index but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_CONTEXT, &N).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_CONTEXT, optInt, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   hypo, context;
int      N, i;

/* number of hypotheses in the context of hypo */
NXP_GETLISTLEN(hypo, NXP_AINFO_CONTEXT, &N);

/* loop to get each hypothesis id */
                    for (i = 0; i < len; i++) {
       NXP_GETLISTELT(hypo, NXP_AINFO_CONTEXT, i, &context);
       ...
}
```

NXP_GetAtomInfo / NXP_AINFO_CURRENT

Purpose

This returns the current atom ids in the inference engine. This code is useful in the middle of a session only.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CURRENT***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_CURRENT */
```

```
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

code is equal to NXP_AINFO_CURRENT.

*optIn*t defines which atom is wanted. It can be one of the following:

Code	Description
NXP_ATYPE_EHS	The current else actions of a rule or a method.
NXP_ATYPE_KB	The current knowledge base atom.
NXP_ATYPE_LHS	The current conditions of a rule or a method are returned.
NXP_ATYPE_METHOD	The current method atom.
NXP_ATYPE_RHS	The current right hand side actions of a rule or a method.
NXP_ATYPE_RULE	The current rule id is returned in thePtr.
NXP_ATYPE_SLOT	The current slot is returned. If this call is done in the middle of a question, the id returned is the same as the question's slot id passed to the question handler.

desc must equal NXP_DESC_ATOM.

thePtr must be a pointer to an AtomId where the requested current atom will be returned.

You should always test if the value returned in thePtr is not NULL before using it! There are cases, even in while the engine is running, where the current atom of a certain type is not defined.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

None.

Examples

The following code gives a few simple examples.

```
AtomId   currentrule, currentcond, currentslot;

/* stores the current rule AtomId in currentrule */
NXP_GetAtomInfo((AtomId)0, NXP_AINFO_CURRENT, (AtomId)0,
                    NXP_ATYPE_RULE, NXP_DESC_ATOM, (Str)&currentrule, 0);

NXP_GetAtomInfo((AtomId)0, NXP_AINFO_CURRENT, (AtomId)0,
                    NXP_ATYPE_LHS, NXP_DESC_ATOM, (Str)&currentcond, 0);
```

```
NXP_GetAtomInfo((AtomId)0, NXP_AINFO_CURRENT, (AtomId)0,
                    NXP_ATYPE_SLOT, NXP_DESC_ATOM, (Str)&currentslot, 0);

/* Test for 0 before going further! */
if(currentslot != (AtomId)0) {
      ...
}
```

NXP_GetAtomInfo / NXP_AINFO_CURRENTKB

Purpose

This returns the atomid of the current knowledge base. Usually it is the last KB loaded, unless changed by a call to NXP_SetAtomInfo with NXP_AINFO_CURRENTKB.

Note: New permanent objects or rules created in the editors belong to the current Knowledge Base.

C Format

The C format is as follows:

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_CURRENTKB***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments:

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_CURRENTKB */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

desc must be NXP_DESC_ATOM.

thePtr must point to an AtomId where the id of the current knowledge base will be returned.

All the other arguments are ignored.

Note: Once you have the KBId you can use the code NXP_AINFO_KBNAME (or NXP_AINFO_NAME after type-casting the KBId into an AtomId) to get the name of the Knowledge Base.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not a NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is null.
NXP_ERR_NOERR	Call was successful.

Macros

You can use the NXP_GETATOMINFO macro:

```
NXP_GETATOMINFO((AtomId)0, NXP_AINFO_CURRENTKB, thePtr)
```

Examples

The following example shows how to save the current KB or get its name:

```
AtomId   curKBid;
int      err, ret;
char    theStr[255];

/* get the id of the current KB */
ret = NXP_GETATOMINFO((AtomId)0, NXP_AINFO_CURRENTKB,
&curKBid);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
/* save the KB into a file "foo.ckb" */
ret = NXP_SaveKB(curKBid, "foo.ckb", NXP_MODE_COMPILED);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
/* get the KB name (DO NOT USE NXP_AINFO_NAME !) */
ret = NXP_GETSTRINFO(curKBid, NXP_AINFO_KBNAME, theStr, 255);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
```

See Also

NXP_GetAtomInfo / NXP_AINFO_KBNAME	returns the name of a knowledge base.
NXP_SetAtomInfo / NXP_AINFO_CURRENTKB	sets the current knowledge base.
NXP_GetAtomInfo / NXP_AINFO_NAME	returns the name of an atom.

NXP_GetAtomInfo / NXP_AINFO_DEFAULTFIRST

Purpose

This returns whether or not the slot's behavior is to follow the current global inheritance path strategy (class vs object first, depth vs breadth first). This is true by default when the slot is created or if you have clicked in the root box of the inheritance tree, this is false if you have set a strategy for that slot.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_DEFAULTFIRST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_DEFAULTFIRST */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is NXP_AINFO_DEFAULTFIRST.

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the slot's inheritance follows the default strategy, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_DEFAULTFIRST, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   slot;
int      default;
NXP_GETINTINFO(slot, NXP_AINFO_DEFAULTFIRST, &default);
```

NXP_GetAtomInfo / NXP_AINFO_DEFVAL

Purpose

This returns the text of the init value of a slot , i.e. the initial value field in the meta-slot editor (DEFVAL stands for "default value").

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_DEFVAL***,** *optAtom***,** *optInt***,** *desc***,***thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code must be NXP_AINFO_DEFVAL to return the public init value , or it must be NXP_AINFO_DEFVAL | NXP_AINFO_PRIVATE to return the private initvalue.

desc must equal NXP_DESC_STR.

thePtr must point to a string buffer of length len.

len is the maximum number of characters that will be written to thePtr.

If *theAtom* does not have any init value defined for it, an empty string is returned in thePtr.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot or value id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_STR.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_DEFVAL, thePtr, len)
```

Examples

The following code gives a simple example.

```
AtomId   slot;
char     str[255];

/* returns the InitValue for slot in str */
NXP_GETSTRINFO(slot, NXP_AINFO_DEFVAL, str, 255);
if(str[0] == (char)'\0') {
       /* empty string means no initvalue for that slot */
}

/* returns the private InitValue for slot in str */
NXP_GETSTRINFO(slot, NXP_AINFO_DEFVAL|NXP_AINFO_PRIVATE, str,
255);
```

NXP_GetAtomInfo / NXP_AINFO_EHS

Purpose

This returns information about the Else actions of a rule or a method (the false right-hand side actions). Each action's atomId is returned in thePtr and then you can get the text of the action using NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_EHS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_EHS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid rule or method id (as returned by NXP_GetAtomId, for instance).

code is equal to NXP_AINFO_EHS.

If *optInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number of Else right hand side actions will be returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this last case, desc should be equal to NXP_DESC_ATOM and the Else right hand side actions with the index optInt will be returned in thePtr.

thePtr must be a pointer or an integer if optInt equals -1. Otherwise it must be a pointer to an atomId.

The Else right hand side actions are listed in the natural rule or method order (as they appear in the rule or method editor).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule or method id.
NXP_ERR_INVARG4	optInt is not -1 nor a valid Else action index.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT. optInt is a valid EHS action index but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_EHS, thePtr)
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_EHS, optInt, thePtr)
```

Examples

The following examples illustrate how to get the text of a rule's Else actions using NXP_AINFO_EHS and NXP_AINFO_NAME:

```
AtomId ruleId, actionId;
int          nActions, i;
char   *ruleName, col1str[20], col2str[1000], col3str[1000];

/* First get the rule's atomId in ruleId */
NXP_GetAtomId(ruleName, ruleId, NXP_ATYPE_RULE);
/* get the number of conditions in nCond */
NXP_GETLISTLEN(ruleId, NXP_AINFO_EHS, &nActions);

/* loop to get each condition's atomId and then its text */
for(i = 0; i < nActions, i++) {
       NXP_GETLISTELT(ruleId, NXP_AINFO_EHS, i, &actionId);

       /* get the text of the 1st column (operator) in col1str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL1, NXP_DESC_STR, col1str, 20);

       /* get the text of the 2nd column in col2str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL2, NXP_DESC_STR, col2str, 1000);

       /* get the text of the 3rd column in col3str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL3, NXP_DESC_STR, col3str, 1000);
}
```

See Also

NXP_AINFO_CACTIONS	Getting the If Change methods
NXP_AINFO_PARENT	Getting the rule or method id from the LHS, RHS or EHS
NXP_AINFO_LHS	Getting the left hand side actions.
NXP_AINFO_RHS	Getting the right hand side actions.
NXP_AINFO_SOURCES	Getting the Order of Sources methods.

NXP_GetAtomInfo / NXP_AINFO_EXHBWRD

Purpose

This returns whether or not exhaustive backward chaining is enabled (option "Exhaustive evaluation" in the Strategy window of the interface).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_EXHBWRD***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list identifies all valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;
AtomId  optAtom;  /* ignored */
```

```
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must ve NXP_AINFO_EXHBWRD to return the default strategy, or it must be NXP_AINFO_EXHBWRD | NXP_AINFO_CURSTRAT to return the current strategy ("Or" operation sets the "current" bit).

desc must be NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if exhaustive backward chaining is on, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_EXHBWRD, thePtr)
```

Examples

The following code gives a simple example.

```
int   exhbwrd;

/* returns in exhbwrd the global strategy default  */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_EXHBWRD, &exhbwrd);
....
/* returns in exhbwrd the current strategy default */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_EXHBWRD | NXP_AINFO_CURSTRAT,
               &exhbwrd);
```

See Also

NXP_AINFO_PFACTIONS	Forward chaining through actions.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_FOCUSPRIO

Purpose

This returns the priority of the hypothesis on the focus agenda (corresponding to the current list in the Agenda monitor window in which the hypothesis appears).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_FOCUSPRIO***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list identifies all valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_FOCUSPRIO */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str      thePtr;
int     len;      /* ignored */
```

theAtom must be a valid hypothesis slot.

code is equal to NXP_AINFO_FOCUSPRIO.

desc must equal NXP_DESC_INT.

thePtr must be a pointer to an integer which will be between 0 and 31. The lower the number, the higher the hypothesis' priority on the focus agenda. A value of 30 = No Focus (i.e. Hypothesis not used by the engine).

Advanced Users: The focus priority can be interpreted as follows: 0 = Suggest, 1-2 = Forward from Suggest Hypothesis, 3-6 = Subgoal Forward, 7-10 = Gates RHS/EHS, 11-15 = Context, 16-29 = Delayed Focus, 30-31 = Unscheduled or No Focus (in other words, the hypothesis is not evaluated by the rule engine).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid hypothesis.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_FOCUSPRIO, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId       hypo;
int          prio;

NXP_GETINTINFO(hypo, NXP_AINFO_FOCUSPRIO, &prio);
```

See Also

NXP_AINFO_SUGGEST	Whether or not a hypothesis is suggested.
NXP_BwrdAgenda	Forces Order of Sources of an atom.

NXP_GetAtomInfo / NXP_AINFO_FORMAT

Purpose

This returns the format information attached to a slot or a property (string edited in the format field of the meta-slot or property editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_FORMAT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_FORMAT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom is a valid slot or property id.

*code i*s NXP_AINFO_FORMAT.

desc must be NXP_DESC_STR.

thePtr must point to a buffer where the format string will be returned.

len is the maximum number of characters that can be written to thePtr.

See the Intelligent Rules Element Reference Manual for more information on formats.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid slot or property id. Also, theAtom cannot be the id for special property Value.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_STR.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_FORMAT, thePtr, len)
```

Examples

The following code gives a simple example.

```
AtomId   atom;
Char     str[255];

NXP_GETSTRINFO(atom, NXP_AINFO_FORMAT, str, 255);
```

NXP_GetAtomInfo / NXP_AINFO_FWRDLINKS

Purpose

This returns the forward links from a slot to its conditions (i.e. conditions where the slot is used and which will be put on the agenda by the forward chaining mechanism).

A list of condition atom ids is returned, the text can be obtained later with NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_FWRDLINKS***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_FWRDLINKS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
```

```
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid atom id.

code is equal to NXP_AINFO_FWRDLINKS.

If optInt is equal to -1, desc must be equal to NXP_DESC_INT and the number n of conditions is returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this case, desc must be equal to NXP_DESC_ATOM, thePtr must be a pointer to an atomId memory location, and the condition id with the index optInt in the list is returned.

Note: The list of conditions is returned in no special order. Once you have the atomId of a condition, you can get the text of the condition using NXP_AINFO_NAME as shown in the example below.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid atom id. 0 (not an error condition) is returned if theAtom is not a valid hypothesis.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT. optInt is between 0 and n-1 but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call use the NXP_GETLISTLEN(atom, code, ptr) macro to get the number nCond of conditions in the list:

```
NXP_GETLISTLEN(theSlot, NXP_AINFO_FWRDLINKS, &nCond)
```

Then use the NXP_GETLISTELT(atom, code, index, ptr) macro, with i between 0 and nCond-1, to get each element in the list:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_FWRDLINKS, i, thePtr)
```

Examples

The following examples show how to get the text of a rule's conditions using NXP_AINFO_FWRDLINKS and NXP_AINFO_NAME:

```
AtomId       slotId, condId;
int          nCond;
char         col1str[20], col2str[1000], col3str[1000];

/* Get the number of conditions in nCond */
NXP_GETLISTLEN(slotId, NXP_AINFO_FWRDLINKS, &nCond);

/* loop to get each condition's atomId and then its text */
for(i = 0; i < nCond, i++) {
       NXP_GetAtomInfo(slotId, NXP_AINFO_FWRDLINKS, (AtomId)0, i,
                     NXP_DESC_ATOM, (Str)&condId, 0);
```

```
        /* get the text of the 1st column (operator) in col1str */
        NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                    NXP_CELL_COL1, NXP_DESC_STR, col1str, 20);

        /* get the text of the 2nd column in col2str */
        NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                    NXP_CELL_COL2, NXP_DESC_STR, col2str, 1000);

        /* get the text of the 3rd column in col3str */
        NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                    NXP_CELL_COL3, NXP_DESC_STR, col3str, 1000);
        ...
}
```

See Also

NXP_AINFO_BWRDLINKS	Backward links from a hypo to its rules.
NXP_AINFO_LHS	Conditions of a rule or method (LHS).

NXP_GetAtomInfo / NXP_AINFO_HASMETA

Purpose

This returns whether or not a slot has meta-slots defined for it (i.e. if anything is defined in the meta-slot editor for that slot).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_HASMETA***,***optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_HASMETA */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_HASMETA.

desc must be NXP_DESC_INT.

thePtr is a pointer to an integer.

A value of 1 will be returned in thePtr if theAtom has meta-slots, and 0 if it doesn't.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_HASMETA, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   slot;
int      hasmeta;

NXP_GETINTINFO(slot, NXP_AINFO_HASMETA, &hasmeta);
if(hasmeta) {
       /* use other GetAtomInfo codes to get the meta-slots
...*/
}
```

NXP_GetAtomInfo / NXP_AINFO_HYPO

Purpose

This returns the hypothesis of a rule.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_HYPO***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_HYPO */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid rule id.

code is equal to NXP_AINFO_HYPO.

desc must be NXP_DESC_ATOM.

thePtr must be a pointer to an AtomId where the hypothesis id will be returned.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

```
NXP_GETATOMINFO(theAtom, NXP_AINFO_HYPO, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId       rule, hypo;
Char         hypoStr[255];

/* Get the atom id of the hypothesis */
NXP_GETATOMINFO(rule, NXP_AINFO_HYPO, &hypo);

/* Get its name */
NXP_GETNAME( hypo, hypoStr, 255 );
```

See Also

NXP_AINFO_BWRDLINKS	Backward links from a hypo to its rules.
NXP_AINFO_LHS	Conditions of a rule or a method.
NXP_AINFO_RHS	Actions of a rule or a method.
NXP_AINFO_EHS	Else actions of a rule or a method.

NXP_GetAtomInfo / NXP_AINFO_INFATOM

Purpose

This returns the inference priority atom attached to a rule or a slot (as edited in the Inf Priority Slot field of the rule or meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INFATOM***,** *optAtom***,** *optInt***,**
*desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INFATOM */
AtomId  optAtom;  /* ignored */
```

```
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid rule or slot id.

code is equal to NXP_AINFO_INFATOM.

desc must be NXP_DESC_ATOM.

thePtr must be a pointer to an AtomId which will receive the id of the inference priority atom of theAtom . It is a slot of type float or integer.

If no priority atom has been defined for theAtom thePtr is set to 0. In that case you can use NXP_AINFO_INFCAT to get the inference priority number.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule or slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

```
NXP_GETATOMINFO(theAtom, NXP_AINFO_INFATOM, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId       atom, infatom;
int          valType;
int          intVal;
double       doubleVal;

/* returns the Inference Priority slot of atom into the
 * variable infatom.  infatom is set to (AtomId)0 if no
 * Inference Priority slot is defined for atom
 */
NXP_GETATOMINFO(atom, NXP_AINFO_INFATOM, &infatom);

/* Get the type and value of the inference priority */
if(infatom != (AtomId)0) {
       NXP_GETINTINFO(infatom, NXP_AINFO_VALUETYPE, &valType);
       if(valType == NXP_VTYPE_LONG)
              NXP_GETINTVAL(infatom, &intVal);
       else if(valType == NXP_VTYPE_DOUBLE)
              NXP_GETDOUBLEVAL(infatom, &doubleVal);
       else  /* error ! */ ;
}
else { /* priority atom doesn't exist: get priority number */
       NXP_GETINTINFO(atom, NXP_AINFO_INFCAT, &intVal);
}
```

See Also

NXP_AINFO_INFCAT	Get the inference priority number.
NXP_AINFO_INHATOM	Get the inheritance priority atom.

NXP_GetAtomInfo / NXP_AINFO_INFBREAK

Purpose

This returns whether a specific rule, condition, method, slot, object, class, or property has an inference break point set on it. It does not return whether a method has a filtered break.

Inference break points can be set with the NXP_SetAtomInfo function or with the stop icon in the network windows in the interface. They stop the inference engine after the evaluation of an atom (whereas agenda break-points stop the engine when the focus of a hypothesis changes).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INFBREAK***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INFBREAK */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid atomId.

code is equal to NXP_AINFO_INFBREAK.

desc must equal NXP_DESC_INT.

thePtr must point to an integer. *thePtr will be set to 1 if a break point is set, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid id of an atom, rule, condition, method, slot, object, class, or property.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INFBREAK, thePtr)
```

Examples

See NXP_AINFO_AGDVBREAK for an example of how to unset all break points through an execute routine.

See Also

`NXP_SetAtomInfo / NXP_SAINFO_INFBREAK`	Setting / unsetting break points from a program.
`NXP_AINFO_AGDVBREAK`	Information on agenda break-points

NXP_GetAtomInfo / NXP_AINFO_INFCAT

Purpose

This returns the inference priority number attached to a rule or a slot (as edited in the Inf Priority Number of the rule or meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INFCAT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INFCAT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid rule or slot id.

code is equal to NXP_AINFO_INFCAT.

desc must be NXP_DESC_INT.

thePtr must be an integer pointer which will receive the inference priority of theAtom.

By default the Rules Element sets the priority number to 1. See the Reference manual for more information on the different priority ranges.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule or slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INFCAT, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   atom;
int      infcat;

/* Get the inference priority number.
 * Normally you should check first that there is no
 * inference priority atom defined for that atom
 * (see the example with NXP_AINFO_INFATOM)
 */
NXP_GETINTINFO(atom, NXP_AINFO_INFCAT, &infcat);
```

See Also

NXP_AINFO_INFATOM	Get the inference priority atom.
NXP_AINFO_INHCAT	Get the inheritance priority number.

NXP_GetAtomInfo / NXP_AINFO_INHATOM

Purpose

This returns the inheritance priority atom attached to a slot (as edited in the Inh Priority Slot field of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHATOM***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHATOM */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

*code i*s equal to NXP_AINFO_INHATOM.

*desc m*ust equal NXP_DESC_ATOM.

t*hePt*r must be a pointer to an AtomId which will receive the id of the inheritance priority atom of theAtom. It is a slot of type float or integer.

If no priority atom has been defined for theAtom thePtr is set to 0. In that case you can use NXP_AINFO_INHCAT to get the inheritance priority number.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

```
NXP_GETATOMINFO(theAtom, NXP_AINFO_INHATOM, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   slot, inhatom;
int          valType;
int          intVal;
double doubleVal;

/* returns the Inheritance priority slot of atom into the
 * variable inhatom.  inhatom is set to (AtomId)0 if no
 * Inheritance priority slot is defined for atom
 */
NXP_GETATOMINFO(slot, NXP_AINFO_INHATOM, &inhatom);

/* Get the type and value of the inference priority */
if(inhatom!= (AtomId)0) {
       NXP_GETINTINFO(inhatom, NXP_AINFO_VALUETYPE, &valType);
       if(valType == NXP_VTYPE_LONG)
              NXP_GETINTVAL(inhatom, &intVal);
       else if(valType == NXP_VTYPE_DOUBLE)
              NXP_GETDOUBLEVAL(inhatom, &doubleVal);
       else  /* error ! */ ;
}
else { /* priority atom doesn't exist: get priority number */
       NXP_GETINTINFO(atom, NXP_AINFO_INHCAT, &intVal);
}
```

See Also

NXP_AINFO_INHCAT	Get the inheritance priority number.
NXP_AINFO_INFATOM	Get the inference priority atom.

NXP_GetAtomInfo / NXP_AINFO_INHCAT

Purpose

This returns the inheritance priority attached to a slot (as edited in the Inh Priority Number of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHCAT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHCAT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_INHCAT.

desc must be NXP_DESC_INT.

thePtr must be an integer pointer which will receive the inheritance priority of theAtom.

By default the Rules Element sets the priority number to 1. See the Reference manual for more information on the different priority ranges.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHCAT, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId   atom;
int      inhcat;

/* Get the inheritance priority number.
```

```
 * Normally you should check first that there is no
 * inheritance priority atom defined for that atom
 * (see the example with NXP_AINFO_INFATOM)
 */
NXP_GETINTINFO(atom, NXP_AINFO_INHCAT, &inhcat);
```

See Also

NXP_AINFO_INHATOM	Get the inheritance priority atom.
NXP_AINFO_INFCAT	Get the inference priority number.

NXP_GetAtomInfo / NXP_AINFO_INHCLASSDOWN

Purpose

This returns whether or not class slots are inheritable downwards (down arrow selected or not underneath the class button of the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHCLASSDOWN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_INHCLASSDOWN */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_INHCLASSDOWN to get the default strategy. If code is NXP_AINFO_INHCLASSDOWN | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if class slots are inheritable downwards, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

`NXP_GETINTINFO(theAtom, NXP_AINFO_INHCLASSDOWN, thePtr).`

Examples

The following code gives a simple example

```
int    prio;

/* default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHCLASSDOWN, &prio);

/* current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHCLASSDOWN | NXP_AINFO_CURSTRAT, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHCLASSUP

Purpose

This returns whether or not class slots are inheritable upwards (up arrow selected or not above the class button of the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHCLASSUP***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_INHCLASSUP */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_INHCLASSUP to get the default strategy. If code is NXP_AINFO_INHCLASSUP | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the class slots are inheritable upwards, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHCLASSUP, thePtr)
```

Examples

The following code gives a simple example

```
int    prio;

/* default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHCLASSUP, &prio);

/* current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHCLASSUP| NXP_AINFO_CURSTRAT,
&prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHDEFAULT

Purpose

This returns whether or not the slot inheritability of a slot follows the default global strategy (Slot button set to default or not in the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHDEFAULT***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHDEFAULT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_INHDEFAULT.

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the slot inheritability follows the default, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHDEFAULT, thePtr)
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

NXP_GETINTINFO(atom, NXP_AINFO_INHDEFAULT, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHDOWN

Purpose

This returns whether or not a slot is downward inheritable (down arrow selected or not, underneath the Slot button of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHDOWN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHDOWN */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_INHDOWN.

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the slot is downward inheritable, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHDOWN, thePtr)
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

NXP_GETINTINFO(atom, NXP_AINFO_INHDOWN, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHOBJDOWN

Purpose

This returns whether or not object slots are inheritable downwards (down arrow selected or not underneath the object button of the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHOBJDOWN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***)**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_INHOBJDOWN */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_INHOBJDOWN to get the default strategy. If code is NXP_AINFO_INHOBJDOWN | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the strategy is on, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHOBJDOWN, thePtr)
```

Examples

The following code gives a simple example

```
int    prio;

/* default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHOBJDOWN, &prio);

/* current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHOBJDOWN | NXP_AINFO_CURSTRAT, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHOBJUP

Purpose

This returns whether or not the object slots are inheritable upwards (up arrow selected or not above the object button of the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHOBJUP***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***)**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_INHOBJUP */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_INHOBJUP get the default strategy. If code is NXP_AINFO_INHOBJUP | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the object slots are inheritable upwards, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHOBJUP, thePtr).
```

Examples

The following code gives a simple example

```
int    prio;

/* default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHOBJUP, &prio);

/* current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHOBJUP| NXP_AINFO_CURSTRAT, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHUP

Purpose

This returns whether or not a slot is upward inheritable (up arrow selected or not, above the Slot button of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHUP***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHUP */
```

```
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_INHUP.

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the slot is upward inheritable, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHUP, thePtr)
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

NXP_GETINTINFO(atom, NXP_AINFO_INHUP, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHVALDEFAULT

Purpose

This returns whether or not the inheritability of the value of a slot follows the default global strategy (Value button set to default or not in the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHVALDEFAULT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHVALDEFAULT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_INHVALDEFAULT.

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the value's inheritability follows the default, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHVALDEFAULT, thePtr).
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

NXP_GETINTINFO(atom, NXP_AINFO_INHVALDEFAULT, &prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHVALDOWN

Purpose

This returns whether or not the value of a slot is downward inheritable (down arrow selected or not underneath the Value button in the meta-slot editor or in the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHVALDOWN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHVALDOWN */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id. If theAtom is NULL, the call returns the global strategy setting for downward inheritance of values. If theAtom is not NULL, the call returns behavior for that atom (combined with the global default).

code must be NXP_AINFO_INHVALDOWN to get the default strategy. If code is NXP_AINFO_INHVALDOWN | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the value of the slot is inheritable, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHVALDOWN, thePtr)
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

/* get the strategy for atom */
NXP_GETINTINFO(atom, NXP_AINFO_INHVALDOWN, &prio);

/* get the global default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHVALDOWN, &prio);

/* get the global current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHVALDOWN| NXP_AINFO_CURSTRAT,
&prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_INHVALUP

Purpose

This returns whether or not the value of a slot is upward inheritable (up arrow selected or not above the Value button in the meta-slot editor or in the Strategy window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_INHVALUP***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_INHVALUP */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

*theAtom i*s a valid slot id. If theAtom is NULL, the call returns the global strategy setting for upward inheritance of values. If theAtom is not NULL, the call returns behavior for that atom (combined with the global default).

code must be NXP_AINFO_INHVALUP to get the default strategy. If code is NXP_AINFO_INHVALUP | NXP_AINFO_CURSTRAT the current strategy is returned ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the value of the slot is for upward inheritance, and set to 0 otherwise.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_INHVALUP, thePtr)
```

Examples

The following code gives a simple example

```
AtomId       atom;
int          prio;

/* get the strategy for atom */
NXP_GETINTINFO(atom, NXP_AINFO_INHVALUP, &prio);

/* get the global default strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHVALUP, &prio);


/* get the global current strategy */
NXP_GETINTINFO(0, NXP_AINFO_INHVALUP | NXP_AINFO_CURSTRAT,
&prio);
```

See Also

`NXP_Strategy`	Change the default or current strategy.
`NXP_AINFO_INHXXX`	Other inheritance codes

NXP_GetAtomInfo / NXP_AINFO_KBID

Purpose

This returns the identifier of the knowledge base to which the atom belongs.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_KBID***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_KBID */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
```

```
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom can be a class, an object, a property, a slot , a method or a rule.

code is equal to NXP_AINFO_KBID.

desc must equal NXP_DESC_ATOM.

thePtr must point to a valid memory location where the knowledge base identifier, which is an unsigned long integer, will be returned.

Notes

There are several other ways of getting the Id of a knowledge base:

- If the knowledge base was loaded with NXP_LoadKB, its kbId was returned by this function.
- If the knowledge base already exists and you know its name you can use NXP_GetAtomId.
- There are three special knowledge bases identified by an id of 0, 1 or 2:

Identifier	Description
0	Corresponds to undefined.kb. This knowledge base contains all the objects, classes and properties which are referenced but are not defined in any of the knowledge bases which are currently loaded. It also contains all the slots which do not have any meta-slots or contexts. Usually, undefined.kb should contain only slots. Nevertheless, objects, classes and properties may be attached to undefined.kb if the knowledge base is split in several files and one of the files has not been loaded by mistake (in that case, warning messages are displayed in the Transcript during the loading).
1	Corresponds to temporary.kb. This knowledge base contains the dynamic objects which are created at runtime. It can only contain objects and it is cleared at each restart session.
2	Corresponds to untitled.kb. This knowledge base contains all the atoms created before any other KB was loaded.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

```
NXP_GETATOMINFO(theAtom, NXP_AINFO_KBID, thePtr).
```

Examples

The following code gives a simple example

```
AtomId theRule, kbId;

/* Get the KB id where theRule is defined */
```

```
NXP_GETATOMINFO(theRule, NXP_AINFO_KBID, &kbId);

/* Save the KB in compiled format */
NXP_SaveKB(kbId, "toto.tkb", NXP_MODE_COMPILED)
```

See Also

NXP_SaveKB	Saves a knowledge base.
NXP_LoadKB	Loads a knowledge base.

NXP_GetAtomInfo / NXP_AINFO_KBNAME

Purpose

This returns the name of a knowledge base (KB) once you know its atomid. (You can now use NXP_AINFO_NAME with KB ids if you typecast the KBId to an atomId).

C Format

The C format is as follows:

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_KBNAME***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments:

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_KBNAME */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom is the atomid of the knowledge base.

optInt is a flag. If optInt is not null the full pathname is returned, otherwise only the short name of the file is returned.

desc must be NXP_DESC_STR.

thePtr is a pointer to a buffer of size len where the string will be returned.

len is the maximum number of characters that can be returned.

You can get a KB atomid from the following calls: NXP_GetAtomId(), NXP_LoadKB(), NXP_GetAtomInfo / NXP_AINFO_CURRENTKB, NXP_GetAtomInfo / NXP_AINFO_KBID.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling NXP_Error

immediately after the call which has failed. NXP_Error returns one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid knowledge base id.
NXP_ERR_INVARG5	desc is not a NXP_DESC_STR.
NXP_ERR_INVARG6	thePtr is null.
NXP_ERR_NOERR	Call was successful.

Macros

You can use the NXP_GETSTRINFO macro if you need only the short name of the knowledge base (optInt is set to 0):

```
NXP_GETSTRINFO(kbId, NXP_AINFO_KBNAME, thePtr, len)
```

Examples

The following example shows how to use NXP_AINFO_KBNAME:

```
AtomId   curKBid;
int      err, ret;
char   theStr[255];

/* get the id of the current KB */
ret = NXP_GETATOMINFO((AtomId)0, NXP_AINFO_CURRENTKB, &curKBid);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
/* get the short KB name */
ret = NXP_GETSTRINFO(curKBid, NXP_AINFO_KBNAME, theStr, 255);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
/* get the full pathname of the same KB */
ret = NXP_GetAtomInfo(curKBid, NXP_DESC_STR, (AtomId)0, 1,
                          NXP_AINFO_KBNAME, theStr, 255);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
```

See Also

NXP_GetAtomId /NXP_ATYPE_KB	Returns the atomid of a knowledge base.
NXP_GetAtomInfo / NXP_AINFO_CURRENTKB	Returns the atomid of the current knowledge base.
NXP_GetAtomInfo / NXP_AINFO_KBID	Returns the atomid of the KB to which an atom is attached.
NXP_LoadKB	Loads a knowledge base.
NXP_SaveKB	Saves a knowledge base.
NXP_GetAtomInfo / NXP_AINFO_KBNAME	Returns the name of a knowledge base.
NXP_GetAtomInfo / NXP_AINFO_NAME	Returns the name of an atom.

NXP_GetAtomInfo / NXP_AINFO_LHS

Purpose

This returns the conditions of a rule or a method (the left-hand side). Each condition's atomId is returned in thePtr, and then you can get the text of the condition using NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_LHS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_LHS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid rule or method id (as returned by NXP_GetAtomId for instance).

code is equal to NXP_AINFO_LHS.

If o*ptInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number of conditions will be returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this last case, desc should be equal to NXP_DESC_ATOM and the condition with the index optInt will be returned in thePtr.

thePtr must be a pointer or an integer if optInt equals -1. Otherwise it must be a pointer to an atomId.

The conditions are listed in the natural rule order (as they appear in the rule editor notebook).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule or method id.
NXP_ERR_INVARG4	optInt is not -1 nor a valid LHS condition index.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT.
	optInt is a valid LHS condition index but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_LHS, thePtr)
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_LHS, optInt, thePtr).
```

Examples

The following examples show how to get the text of a rule's conditions using NXP_AINFO_LHS and NXP_AINFO_NAME:

```
AtomId        ruleId, condId;
int           nCond, i;
char          *ruleName, col1str[20], col2str[1000], col3str[1000];

/* First get the rule's atomId in ruleId */
NXP_GetAtomId(ruleName, ruleId, NXP_ATYPE_RULE);
/* get the number of conditions in nCond */
NXP_GETLISTLEN(ruleId, NXP_AINFO_LHS, &nCond);

/* loop to get each condition's atomId and then its text */
for(i = 0; i < nCond, i++) {
       NXP_GETLISTELT(ruleId, NXP_AINFO_LHS, i, &condId);

       /* get the text of the 1st column (operator) in col1str */
       NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL1, NXP_DESC_STR, col1str, 20);

       /* get the text of the 2nd column in col2str */
       NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL2, NXP_DESC_STR, col2str, 1000);

       /* get the text of the 3rd column in col3str */
       NXP_GetAtomInfo(condId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL3, NXP_DESC_STR, col3str, 1000);
}
```

See Also

NXP_AINFO_CACTIONS	Get the If Change methods
NXP_AINFO_PARENT	Get the rule or method id from the LHS, EHS or RHS
NXP_AINFO_RHS	Get the right hand side actions.
NXP_AINFO_EHS	Get the Else right hand side actions.
NXP_AINFO_SOURCES	Get the Order of Sources methods

NXP_GetAtomInfo / NXP_AINFO_LINKED

Purpose

This returns information about the type of link between a class or an object and another class or object.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_LINKED***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_LINKED */
AtomId  optAtom;
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
```

```
Str     thePtr;
int     len;      /* ignored */
```

theAtom and *optAtom* should both be valid class or object ids, theAtom being the id of the parent and optAtom the id of the child.

code is equal to NXP_AINFO_LINKED.

desc must equal NXP_DESC_INT.

**thePtr i*s a pointer to an integer representing the type of link. It can take the following values:

Code	Description
NXP_LINK_NOLINK	no link.
NXP_LINK_PERMLINK	permanent link (i.e. kept in the knowledge base)
NXP_LINK_TEMPLINK	temporary link (dynamically created in rules, methods or by external calls).
NXP_LINK_TEMPUNLINK	temporarily deleted link (permanent links deleted in rules, methods or by external calls).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or not a valid class or object id.
NXP_ERR_INVARG3	optAtom is NULL or not a valid class or object id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.

Macros

None.

Examples

This tests the type of link between myClass and myObject (both are atomIds):

```
int   link;

NXP_GetAtomInfo(myClass, NXP_AINFO_LINKED, myAtom, 0, NXP_DESC_INT,
                (Str)&link, 0);

switch(link) {
       case NXP_LINK_NOLINK:
       ...;
       case NXP_LINK_PERMLINK:
       ...;
       case NXP_LINK_TEMPLINK:
       ...;
        case NXP_LINK_TEMPUNLINK:
       ...;
}
```

See Also

NXP_AINFO_CHILDOBJECT	Get the children objects of a class.
NXP_AINFO_PARENTCLASS	Get the parent class of an object.

NXP_GetAtomInfo / NXP_AINFO_METHODS

Purpose

This returns information about the methods attached to a slot, an object, a class or a property.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_METHODS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_METHODS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid object or class id.

code must equal NXP_AINFO_METHODS.

optInt is an integer between -1 and n-1.

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

Notes

The mechanism used to retrieve this list uses the following sequence of calls:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number n of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with the NXP_AINFO_METHODS code and optInt set to any value between 0 and n-1 where n is the value returned by the first call. The id of the (optInt+1)th atom in the list will be returned in thePtr (which must be a pointer to an AtomId).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid class, object, slot or property id.
NXP_ERR_INVARG4	optInt is not equal to -1 or is not a valid method index.
NXP_ERR_INVARG5	optInt is equal to -1 and desc is not equal to NXP_DESC_INT.optInt is a valid method index, but desc is not equal to NXP_DESC_ATOM.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_METHODS, &len).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_METHODS, i, thePtr)
```

Examples

The following example illustrates how to get the list of methods of theObj using the macros:

```
NXP_GETLISTLEN(theObj, NXP_AINFO_METHODS, &nMethods);
for(i = 0; i < nMethods; i++){
       NXP_GETLISTELT(theObj, NXP_AINFO_METHODS, i,
theMethod);
       ...
}
```

See Also

NXP_AINFO_CHILDCLASS	The children classes of a class.
NXP_AINFO_CHILDOBJECT	The children objects of a class.

NXP_GetAtomInfo / NXP_AINFO_MOTSTATE

Purpose

This returns information about the current state of the inference engine.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_MOTSTATE***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_MOTSTATE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
```

```
Str     thePtr;
int     len;      /* ignored */
```

code is equal to NXP_AINFO_MOTSTATE.

desc must equal NXP_DESC_INT.

thePtr must be a pointer to an integer where one of the following codes will be returned:

Code	Description
NXP_STATE_DONE	The state after the end of a session.
NXP_STATE_QUESTION	A question is pending. The engine is stopped, waiting for a value.
NXP_STATE_RUNNING	The inference engine is running (i.e. when executing an "Execute" routine).
NXP_STATE_STOPPED	The session has been interrupted (by clicking in the interrupt button or by calling NXP_Control with the NXP_CTRL_STOPSESSION code.

Notes

If you make this call within a Question handler, thePtr is set to NXP_STATE_QUESTION.

If you make this call within another handler (Execute, Polling, Apropos, etc.), thePtr is set to NXP_STATE_RUNNING.

If you call NXP_CTRL_STOPSESSION within your Question handler to make the question non-modal, the state becomes NXP_STATE_STOPPED. You must call NXP_CTRL_CONTINUE to resume the session, but you should first answer the current question with NXP_Volunteer. Otherwise, the same question will return when the engine restarts.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO((AtomId)0, NXP_AINFO_MOTSTATE, thePtr).
```

Examples

The following code gives a simple example

```
int    theState;

/* get the state of the engine */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_MOTSTATE, &theState);

/* restart the session after the end of session */
```

```
if(theState == NXP_STATE_DONE)
      NXP_Control(NXP_CTRL_RESTART);
```

See Also

NXP_SetHandler /NXP_PROC_ENDOFSESSION	Handler called at the end of a session.
NXP_PROC_POLLING	Handler called at each inference cycle while the engine is running.
NXP_PROC_QUESTION	Handler called during a question.
NXP_Control	Controls the engine.

NXP_GetAtomInfo / NXP_AINFO_NAME

Purpose

This returns the string name of an atom from its id (The opposite function is NXP_GetAtomId).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_NAME***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* NXP_AINFO_NAME */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom is the id of the atom which name will be returned. theAtom can be any of the following types:

Code	Description
NXP_ATYPE_CLASS	See NXP_ATYPE_SLOT below.
NXP_ATYPE_OBJECT	See NXP_ATYPE_SLOT below.
NXP_ATYPE_PROP	See NXP_ATYPE_SLOT below.
NXP_ATYPE_SLOT	The Object, Class, Property, or Slot Name as displayed in the notebooks will be returned. For a slot with a .value property, the .value will NOT be present in the returned string.
NXP_ATYPE_RULE	The rule name will be returned as a string.
NXP_ATYPE_KB	The knowledge base.
NXP_ATYPE_CACTIONS	See NXP_ATYPE_SOURCE below.
NXP_ATYPE_LHS	See NXP_ATYPE_SOURCE below.
NXP_ATYPE_RHS	See NXP_ATYPE_SOURCE below.
NXP_ATYPE_EHS	See NXP_ATYPE_SOURCE below.

Code	Description
NXP_ATYPE_SOURCE	These types mean that theAtom is the id of a condition of a rule or of a method. In this case optInt can have any of the following values: NXP_CELL_COL1, NXP_CELL_COL2, NXP_CELL_COL3. The string returned corresponds to the first, second, and third column as they are displayed in the rule editor or in the method editor (operator, first argument, second argument). The operator string returned is the one used in the network display.

code is equal to NXP_AINFO_NAME.

optInt is unused except when the atom type is NXP_ATYPE_CACTIONS, NXP_ATYPE_LHS, NXP_ATYPE_RHS, NXP_ATYPE_EHS or NXP_ATYPE_SOURCE (see example below).

desc must equal NXP_DESC_STR.

thePtr is a pointer to the string returned. thePtr must point to a buffer of at least len characters.

len is the maximum number of characters that can be returned.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom id.
NXP_ERR_INVARG4	optInt is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_STR.

Macros

You can use the NXP_GETNAME(atom, ptr, len) macro:

```
NXP_GETNAME(theAtom, thePtr, len).
```

Examples

The following examples illustrate two ways of getting the name of slot "object.property" using the atomIds of the object theObj and the property theProp.

The first solution is to get the names of theProp and theObj and then concatenate them with a dot in the middle:

```
AtomId theProp, theObj, theSlot;
char   objStr[255], propStr[255], slotStr[255];

NXP_GETNAME(theProp, propStr, 255);
NXP_GETNAME(theObj, objStr, 255);

/* build "object.property" */
sprintf(slotStr, "%s.%s", objStr, propStr);
```

Another solution is to get the atomId of the slot first and then get its name:

```
NXP_GetAtomInfo(theObj, NXP_AINFO_SLOT, theProp, 0,
                        NXP_DESC_ATOM, theSlot, 0);
NXP_GetAtomInfo(theSlot, NXP_AINFO_NAME, (AtomId)0, 0,
                        NXP_DESC_STR, slotStr, 255);
```

Note: In the above case, ".Value" is not added to the slot's name if theProp is the special property Value.

The following examples illustrate how to get the text of a rule's conditions using NXP_AINFO_LHS and NXP_AINFO_NAME:

```
AtomId       ruleId, theCond;
char         col1str[20], col2str[1000], col3str[1000];

/* First get the rule's atomId in ruleId */
NXP_GetAtomInfo(ruleName, ruleId, NXP_ATYPE_RULE);

/* get the number of conditions in nCond */
NXP_GETLISTLEN(ruleId, NXP_AINFO_LHS, &nCond);

/* loop to get each condition's atomId and then its text */
for(i = 0; i < nCond, i++){
       NXP_GETLISTELT(ruleId, NXP_AINFO_LHS, i, &theCond);

       /* get the text of the 1st column (operator) in col1str */
       NXP_GetAtomInfo(theCond, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL1, NXP_DESC_STR, col1str, 20);

       /* get the text of the 2nd column in col2str */
       NXP_GetAtomInfo(theCond, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL2, NXP_DESC_STR, col2str, 1000);
       /* get the text of the 3rd column in col3str */
       NXP_GetAtomInfo(theCond, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL3, NXP_DESC_STR, col3str, 1000);
}
```

See Also

`NXP_GetAtomId` Get the atomId from an atom name.

NXP_GetAtomInfo / NXP_AINFO_NEXT

Purpose

This code allows you to retrieve lists of atoms stored in the working memory.

You can get the list of all the classes, objects, properties, methods, data, hypotheses, rules or knowledge bases by successive calls using NXP_AINFO_NEXT. See also NXP_AINFO_PREV to go through the lists in the other direction. In each list, atoms are ordered as they appear in the editors or notebooks.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_NEXT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_NEXT */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

The general mechanism to retrieve these lists is the following:

- In your first call you pass NULL in theAtom and the first element in the list will be returned in thePtr.
- Then you pass the atom returned by the previous call in theAtom and the next element in the list will be returned in thePtr. When the end of the list is reached, NULL is returned in thePtr.

theAtom can be NULL in which case the first atom of the list will be returned. Otherwise theAtom should be a valid atom id belonging to the same list as specified in optInt. If theAtom is the last one in the list, NULL will be returned.

code is equal to NXP_AINFO_NEXT.

desc must be NXP_DESC_ATOM.

thePtr must be a valid pointer to a memory location where the id of the next atom will be returned.

optInt can take the following values:

Code	Description
NXP_ATYPE_CLASS	If theAtom is NULL, thePtr will hold the id of the first class (in alphabetical order). Otherwise, theAtom should be a valid class id, and thePtr will hold the class following theAtom in alphabetical order. The order is the same as displayed in the class notebook.
NXP_ATYPE_DATA	If theAtom is NULL, thePtr will hold the id of the first data (in alphabetical order). Otherwise, theAtom should be a valid data id, and thePtr will hold the data following theAtom in alphabetical order. The order is the same as displayed in the data notebook. In this case, theAtom should not only be a slot, but it should also belong to the list of data if it isn't NULL.
NXP_ATYPE_HYPO	If theAtom is NULL, thePtr will hold the id of the first hypothesis (in alphabetical order). Otherwise, theAtom should be a valid hypothesis id, and thePtr will hold the hypothesis following theAtom in alphabetical order. The order is the same as displayed in the hypotheses notebook.
NXP_ATYPE_KB	If theAtom is NULL, thePtr will hold the id of the first knowledge base (in alphabetical order). Otherwise, theAtom should be a valid knowledge base id, and thePtr will hold the knowledge base following theAtom in alphabetical order.
NXP_ATYPE_METHOD	If theAtom is NULL, thePtr will hold the id of the first method (in alphabetical order). Otherwise, theAtom should be a valid method id, and thePtr will hold the method following theAtom in alphabetical order. The order is the same as displayed in the method notebook. The methods are sorted alphabetically by method name.
NXP_ATYPE_OBJECT	If theAtom is NULL, thePtr will hold the id of the first object (in alphabetical order). Otherwise, theAtom should be a valid object id, and thePtr will hold the object following theAtom in alphabetical order. The order is the same as displayed in the object notebook.

Code	Description
NXP_ATYPE_PROP	If theAtom is NULL, thePtr will hold the id of the first property (in alphabetical order). Otherwise, theAtom should be a valid property id, and thePtr will hold the property following theAtom in alphabetical order. The order is the same as displayed in the property notebook.
NXP_ATYPE_RULE	If theAtom is NULL, thePtr will hold the id of the first rule (in alphabetical order). Otherwise, theAtom should be a valid rule id, and thePtr will hold the rule following theAtom in alphabetical order. The order is the same as displayed in the rule notebook. The rules are sorted alphabetically by rule names and then by hypothesis names if they do not have a rule name.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not of the right type (as specified by optInt).
NXP_ERR_INVARG4	optInt is not a valid code.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.

Macros

For the first call use the NXP_GETLISTFIRST(type, ptr) macro:

```
NXP_GETLISTFIRST(listType, nextAtom)
```

then use the NXP_GETLISTNEXT(atom, type, ptr) macro:

```
NXP_GETLISTNEXT(currentAtom, listType, nextAtom)
```

Examples

This illustrates how to get the list of all objects loaded in memory:

```
AtomId   theObj, nextObj, firstObj;

/* get the first object in alphabetical order */
NXP_GETLISTFIRST(NXP_ATYPE_OBJECT, &firstObj);

/* loop to get the following objects */
if(firstObj != NULL){
      theObj = firstObj;
      while(theObj != NULL) {
            NXP_GETLISTNEXT(theObj, NXP_ATYPE_OBJECT,
&nextObj);
            theObj = nextObj;
            ...
      }
}
```

See Also

NXP_AINFO_PREV Get the previous atom in a list.

NXP_GetAtomInfo / NXP_AINFO_PARENT

Purpose

This returns information about the parent of a slot, a condition, a right-hand side action, or an Else (right-hand side) action that appears in a rule or method. Can also return for the method name.

Note: Here PARENT does not refer to the class-object or object-object relation. See NXP_AINFO_PARENTCLASS and NXP_AINFO_PARENTOBJECT.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PARENT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_PARENT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must a valid slot, condition (LHS), action (RHS or EHS) of a rule or a method. If theAtom is a slot id the parent object or class is returned. If theAtom is a LHS, RHS or EHS the id returned is the rule or method id. If theAtom is a method, the AtomId of the atom to which the method is attached is returned.

code is equal to NXP_AINFO_PARENT.

desc must equal NXP_DESC_ATOM.

thePtr must be a pointer to an AtomId where the id of the parent atom will be returned.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid id or is NULL.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

`NXP_GETATOMINFO(theAtom, NXP_AINFO_PARENT, thePtr).`

Examples

The following examples illustrate two ways of getting the name of the parent object of a slot.

One way is to use NXP_AINFO_PARENT to get the id of the object, and then use NXP_AINFO_NAME to get its name:

```
NXP_GetAtomInfo(theSlot, NXP_AINFO_PARENT, (AtomId)0, 0,
                            NXP_DESC_ATOM, (Str)&theObj, 0);
NXP_GetAtomInfo(theObj, NXP_AINFO_NAME, (AtomId)0, 0,
                            NXP_DESC_STR, objName, 255);
```

Another way is to get the name of the slot and remove the ".property" part (if any):

```
NXP_GetAtomInfo(theSlot, NXP_AINFO_NAME, (AtomId)0, 0,
                    NXP_DESC_STR, slotName, 255);
strcpy(objName, slotName);
for(i = 0; objName[i] != (char)'\0'; i++){
      if(objName[i] == (char)'.'){
            objName[i] = (char)'\0'; /*cut string at the dot*/
            break;
      }
}
```

See Also

NXP_AINFO_CACTIONS	Get the If Change methods.
NXP_AINFO_EHS	Get the Else actions.
NXP_AINFO_LHS	Get the left hand side conditions.
NXP_AINFO_METHODS	Get the method's attached to atom.
NXP_AINFO_PARENTCLASS	Get the parent in the class structure.
NXP_AINFO_PARENTOBJECT	Get the parent in the object structure.
NXP_AINFO_RHS	Get the right hand side actions.
NXP_AINFO_SOURCES	Get the Order of Sources methods.

NXP_GetAtomInfo / NXP_AINFO_PARENTCLASS

Purpose

This returns information about the parent classes of a class or an object.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PARENTCLASS***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_PARENTCLASS */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid object or class id.

code must equal NXP_AINFO_PARENTCLASS.

optInt is an integer between -1 and n-1.

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

The mechanism used to retrieve this list uses the following sequence of calls:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number n of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with the NXP_AINFO_PARENTCLASS code and optInt set to any value between 0 and n-1 where n is the value returned by the first call. The id of the (optInt+1)th atom in the list will be returned in thePtr (which must be a pointer to an AtomId).

Notes

The ordering of the list is not defined and can change during a session (i.e. links to parent classes are created or deleted dynamically).

For the parent class of a slot use NXP_AINFO_PARENT (a slot has only one parent class or object).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theAtom is NULL or is not a valid class id.
`NXP_ERR_INVARG4`	optInt is not equal to -1 or is not a valid parent class index.
`NXP_ERR_INVARG5`	optInt is equal to -1 and desc is not equal to NXP_DESC_INT.
	optInt is a valid parent class index, but desc is not equal to NXP_DESC_ATOM.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_PARENTCLASS, &len).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_PARENTCLASS, i, thePtr)
```

Examples

The following example illustrates how to get the list of parent classes of theObj using the macros:

```
NXP_GETLISTLEN(theObj, NXP_AINFO_PARENTCLASS, &nClass);
for(i = 0; i < nClass; i++){
      NXP_GETLISTELT(theObj, NXP_AINFO_PARENTCLASS, i,
theClass);
      ...
}
```

See Also

NXP_AINFO_CHILDCLASS	The children classes of a class.
NXP_AINFO_CHILDOBJECT	The children objects of a class.
NXP_AINFO_LINKED	The type of link between a class or object and another class or object.
NXP_AINFO_PARENT	The parent class or object of a slot.
NXP_AINFO_PARENTOBJECT	The parent objects of an object.

NXP_GetAtomInfo / NXP_AINFO_PARENTFIRST

Purpose

This returns whether the inheritance search for a slot should begin by searching the parent objects of the slot or the classes to which the slot belongs (strategy displayed in the Strategy window or in the meta-slot editor for a slot).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PARENTFIRST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_PARENTFIRST */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid slot id. If theAtom is NULL, the call returns the global strategy setting for parent first versus class first inheritance. If theAtom is not NULL, the call returns behavior for that atom (combined with the global default).

code must be NXP_AINFO_PARENTFIRST to get the default strategy. it must be NXP_AINFO_PARENTFIRST | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the parent objects are searched first, and set to 0 if the parent classes are searched first.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PARENTFIRST, thePtr).
```

Examples

The following example illustrates the inheritance strategy for theSlot:

```
int          strat;

NXP_GETINTINFO(theSlot, NXP_AINFO_PARENTFIRST, &strat);
if(strat == 1)
  /* Parent objects are used first */
else
  /* Parent classes are used first */
```

See Also

NXP_AINFO_BREADTHFIRST	Breadth first or depth first strategy.
NXP_AINFO_DEFAULTFIRST	Default strategy
NXP_Strategy	Change default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_PARENTOBJECT

Purpose

This returns the parent objects of an object.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PARENTOBJECT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_PARENTOBJECT */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid object id.

code is equal to NXP_AINFO_PARENTOBJECT.

optInt is an integer between -1 and n-1.

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

The mechanism used to retrieve this list uses the following sequence of calls:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number n of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with the NXP_AINFO_PARENTOBJECT code and optInt set to any value between 0 and n-1 where n is the value returned by the first call. The id of the (optInt+1)th atom in the list will be returned in thePtr (which must be a pointer to an AtomId).

Notes

The ordering of the list is not defined and can change during a session (i.e. links to parent objects are created or deleted dynamically).

For the parent object of a slot use NXP_AINFO_PARENT (a slot has only one parent class or object).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theAtom is NULL or is not a valid object id.
`NXP_ERR_INVARG4`	optInt is not equal to -1 or is not a valid parent object index.
`NXP_ERR_INVARG5`	optInt is equal to -1 and desc is not equal to NXP_DESC_INT. optInt is a valid parent object index but desc is not equal to NXP_DESC_ATOM.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_PARENTOBJECT, &len).
```

Then use NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_PARENTOBJECT, i, thePtr)
```

Examples

The following example illustrates how to get the list of parent objects of theObj using the macros:

```
int          nObject;
AtomId       theObj;
NXP_GETLISTLEN(theObj, NXP_AINFO_PARENTCLASS, &nObject);
for(i = 0; i < nObject; i++){
      NXP_GETLISTELT(theObj, NXP_AINFO_PARENTCLASS, i,
theObj);
      ...
}
```

See Also

NXP_AINFO_CHILDCLASS	The children classes of a class.
NXP_AINFO_CHILDOBJECT	The children objects of a class.
NXP_AINFO_LINKED	The type of link between a class or object and another class or object.
NXP_AINFO_PARENTCLASS	The parent classes of an object.

NXP_GetAtomInfo / NXP_AINFO_PFACTIONS

Purpose

This returns whether or not the assignments done in the right-hand-side of rules or in methods are forwarded (PF = Propagate Forward).

It corresponds to the Forward Action Effects option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PFACTIONS***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PFACTIONS */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PFACTIONS to get the default strategy. It must be NXP_AINFO_PFACTIONS | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the action effects are forwarded, and set to 0 if they are not forwarded.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PFACTIONS, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PFELSEACTIONS

Purpose

This returns whether or not the assignments done in the Else actions of rules are forwarded (PF = Propagate Forward).

It corresponds to the Else Forward Action Effects option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PFELSEACTIONS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PFELSEACTIONS */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PFELSEACTIONS to get the default strategy. It must be NXP_AINFO_PFELSEACTIONS | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to NXP_FSTRAT_ON if the action effects are forwarded, NXP_FSTRAT_OFF if they are not forwarded, and NXP_FSTRAT_GLOBAL if they are set to global forward actions strategy.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PFELSEACTIONS, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from righthand-side and right-hand-side actions of a method.
NXP_AINFO_PFACTIONS	Forward chaining from lefthand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PFMETHODACTIONS

Purpose

This returns whether or not the assignments done in the left-hand-side and right-hand-side of methods are forwarded (PF = Propagate Forward).

It corresponds to the Methods Forward Action Effects option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PFMETHODACTIONS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PFMETHODSACTIONS */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PFMETHODACTIONS to get the default strategy. It must be NXP_AINFO_PFMETHODACTIONS | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to NXP_FSTRAT_ON if the action effects are forwarded, set to NXP_FSTRAT_OFF if they are not forwarded, and set to NXP_FSTRAT_GLOBAL if they are set to global forward actions strategy.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PFMETHODACTIONS, thePtr).
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule.
NXP_AINFO_PFACTIONS	Forward chaining from lefthand-side and righthand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PFMETHODELSEACTIONS

Purpose

This returns whether or not the assignments done in the Else actions of methods are forwarded (PF = Propagate Forward).

It corresponds to the Methods Else Forward Action Effects option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PFMETHODELSEACTIONS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PFMETHODSACTIONS */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str      thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PFMETHODELSEACTIONS to get the default strategy. It must be NXP_AINFO_PFMETHODELSEACTIONS | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to NXP_FSTRAT_ON if the action effects are forwarded, NXP_FSTRAT_OFF if they are not forwarded, and NXP_FSTRAT_GLOBAL if they are set to global forward actions strategy.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PFMETHODELSEACTIONS,
thePtr).
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule.
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PREV

Purpose

This code allows you to retrieve lists of atoms stored in the working memory.

You can get the list of all the classes, objects, properties, data, hypotheses, methods, rules or knowledge bases by successive calls using NXP_AINFO_PREV. See also NXP_AINFO_NEXT to go through the lists in the other direction. In each list, atoms are ordered as they appear in the editors or notebooks.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PREV***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId     theAtom;
int        code;    * = NXP_AINFO_PREV */
AtomId     optAtom;  /* ignored */
int        optInt;
int        desc;/   * = NXP_DESC_ATOM */
Str        thePtr;
int        len;      /* ignored */
```

The general mechanism to retrieve these lists is the following:

- In your first call, you pass NULL in theAtom and the last element in the list will be returned in thePtr.
- Then you pass the atom returned by the previous call in theAtom and the previous element in the list will be returned in thePtr. When the end of the list is reached, NULL is returned in the Ptr.

theAtom can be NULL in which case the last atom of the list will be returned. Otherwise, theAtom should be a valid atom id belonging to the same list as specified in optInt. If theAtom is the first one in the list, NULL will be returned.

code is equal to NXP_AINFO_PREV.

desc must be NXP_DESC_ATOM.

thePtr must be a valid pointer to a memory location where the id of the next atom will be returned.

optInt can take one of the following values:

Code	Description
NXP_ATYPE_CLASS	If theAtom is NULL, thePtr will hold the id of the last class (in alphabetical order). Otherwise, theAtom should be a valid class id, and thePtr will hold the class preceding theAtom in alphabetical order. The order is the same as displayed in the class notebook.
NXP_ATYPE_DATA	If theAtom is NULL, thePtr will hold the id of the last data (in alphabetical order). Otherwise, theAtom should be a valid data id, and thePtr will hold the data preceding theAtom in alphabetical order. The order is the same as displayed in the data notebook. In this case, theAtom should not only be a slot, but it should also belong to the list of data if it isn't NULL.
NXP_ATYPE_HYPO	If theAtom is NULL, thePtr will hold the id of the last hypothesis (in alphabetical order). Otherwise, theAtom should be a valid hypothesis id, and thePtr will hold the hypothesis preceding theAtom in alphabetical order. The order is the same as displayed in the hypotheses notebook.
NXP_ATYPE_KB	If theAtom is NULL, thePtr will hold the id of the last knowledge base (in alphabetical order). Otherwise, theAtom should be a valid knowledge base id, and thePtr will hold the knowledge base preceding theAtom in alphabetical order.
NXP_ATYPE_METHOD	If theAtom is NULL, thePtr will hold the id of the last method (in alphabetical order). Otherwise, theAtom should be a valid method id, and thePtr will hold the method preceding theAtom in alphabetical order. The order is the same as displayed in the method notebook. The methods are sorted alphabetically by method name.
NXP_ATYPE_OBJECT	If theAtom is NULL, thePtr will hold the id of the last object (in alphabetical order). Otherwise, theAtom should be a valid object id, and thePtr will hold the object preceding theAtom in alphabetical order. The order is the same as displayed in the object notebook.
NXP_ATYPE_PROP	If theAtom is NULL, thePtr will hold the id of the last property (in alphabetical order). Otherwise, theAtom should be a valid property id, and thePtr will hold the property preceding theAtom in alphabetical order. The order is the same as displayed in the property notebook.
NXP_ATYPE_RULE	If theAtom is NULL, thePtr will hold the id of the last rule (in alphabetical order). Otherwise, theAtom should be a valid rule id, and thePtr will hold the rule preceding theAtom in alphabetical order. The order is the same as displayed in the rule notebook. The rules are sorted alphabetically by rule names and then by hypothesis names if they do not have a rule name.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not of the right type (as specified by optInt).
NXP_ERR_INVARG4	optInt is not a valid code.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.

Macros

None are defined in nxpdef.h but you can add your own macros on the model of NXP_GETLISTFIRST and NXP_GETLISTNEXT.

Examples

This illustrates how to go through the list of all rules in reverse order:

```
AtomId   theRule, prevRule, lastRule;

/* get the last rule in alphabetical order */
NXP_GetAtomInfo((AtomId)0, NXP_AINFO_PREV, (AtomId)0,
            NXP_ATYPE_RULE, NXP_DESC_ATOM, (Str)&lastRule,
0);

/* loop to get the preceding objects */
if(lastRule != NULL){
      theRule = lastRule;
      while(theRule != NULL) {
            NXP_GetAtomInfo(theRule, NXP_AINFO_PREV,
(AtomId)0,
            NXP_ATYPE_RULE,NXP_DESC_ATOM, (Str)&prevRule, 0)
            theRule= prevRule;
            ...
      }
}
```

See Also

`NXP_AINFO_NEXT` Get the next atom in a list.

NXP_GetAtomInfo / NXP_AINFO_PROCEXECUTE

Purpose

This returns either the number of execute handlers installed or the name of a execute handler.

Execute handlers are returned in the order they were installed. For more information see NXP_SetHandler / NXP_PROC_EXECUTE.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PROCEXECUTE***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom; /* ignored */
int     code;    /* = NXP_AINFO_PROCEXECUTE */
AtomId  optAtom; /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;
```

code is equal to NXP_AINFO_PROCEXECUTE.

If o*ptInt* is equal to -1, the number n of execute routines installed is returned in the order they were installed. In this case, desc is equal to NXP_DESC_INT and thePtr should be a pointer to an integer.

Otherwise, optInt is the index of the Execute (a number between 0 and n-1) and the name of the Execute Handler is returned. In this case, *desc* is equal to NXP_DESC_STR and the string is returned in thePtr.

len is the maximum number of characters that can be returned in thePtr if desc is equal to NXP_DESC_STR.

Once you have the name of an Execute procedure, you can call NXP_GetHandler to get more information on this procedure (or call NXP_GetHandler2 if the procedure was installed with NXP_SetHandler2).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG4`	desc is equal to NXP_DESC_STR but the value of optInt is not between 0 and n-1.
`NXP_ERR_INVARG5`	optInt is equal to -1 but desc is not equal to NXP_DESC_INT.
`NXP_ERR_INVARG6`	thePtr is NULL.

Macros

For the first call use the NXP_GETLISTLEN(atom, code, ptr) macro to get the number of Execute handlers:

```
NXP_GETLISTLEN((AtomId)0, NXP_AINFO_PROCEXECUTE, thePtr)
```

Then use the NXP_GETLISTELTSTR(atom, code, index, ptr, len) macro to get the individual names:

```
NXP_GETLISTELTSTR((AtomId)0, NXP_AINFO_PROCEXECUTE, i, thePtr,
len)
```

See Also

`NXP_GetHandler, NXP_GetHandler2`	Getting Execute handlers.
`NXP_SetHandler, NXP_SetHandler2`	Setting Execute handlers.

NXP_GetAtomInfo / NXP_AINFO_PROMPTLINE

Purpose

This returns the prompt line string attached to a slot. This is the string used during a question about the value of this slot (or when you change the value with Volunteer / Modify in the interface). This attribute is edited in the Prompt Line text field of the meta-slot editor.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PROMPTLINE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* NXP_AINFO_PROMPTLINE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a slot.

code is equal to NXP_AINFO_PROMPTLINE.

desc must be NXP_DESC_STR.

thePtr must point to a buffer where the prompt line string will be returned. The buffer must contain at least len characters. thePtr is the empty string if no prompt line is defined in the slot.

len is the maximum number of characters that can be written to thePtr.

Notes

If there is no prompt line attribute defined in theAtom (thePtr set to the empty string), the Rules Element tries to inherit the prompt line from parent objects or classes or it uses the default prompt line ("What is the value of slot?").

The prompt line is also the second argument passed to the Question handler, you don't need to use NXP_AINFO_PROMPTLINE within your handler procedure in order to display the question about the current slot.

There is a small difference between a prompt line returned by NXP_GetAtomInfo and the one passed to the Question handler: in the first case it is non expanded, i.e. can contain @SELF and @V() interpretations, in the second case it is completely expanded as it would be displayed in the Session Control window.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a slot.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_STR.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_PROMPTLINE, thePtr, len)
```

Examples

The following code gives a simple example:

```
char   prompt[1000];

NXP_GETSTRINFO(theSlot, NXP_AINFO_PROMPTLINE, prompt, 1000);
if(prompt[0] == (char)'\0') {
       printf("No prompt line\n");
}
else {
       printf("Prompt line = %s\n", prompt);
}
```

See Also

`NXP_PROC_QUESTION` Question handler

NXP_GetAtomInfo / NXP_AINFO_PROP

Purpose

This returns the property of the slot .

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PROP***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_PROP */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_ATOM */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code is equal to NXP_AINFO_PROP.

*des*c must be NXP_DESC_ATOM.

t*hePtr* must be a pointer to an AtomId. The property id of the slot is returned in thePtr.

Note: It is more efficient to use this call for getting the property id and then get the property's name rather than getting the slot's name first and parsing it to extract the property's name.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM.

Macros

You can use the NXP_GETATOMINFO(atom, code, ptr) macro:

```
NXP_GETATOMINFO(theAtom, NXP_AINFO_PROP, thePtr)
```

See Also

NXP_AINFO_SLOT Get properties of a class or an object.

NXP_GetAtomInfo / NXP_AINFO_PTGATES

Purpose

This returns whether or not forward chaining through semantic gates is enabled (PT = propagate through).

This corresponds to the Forward Through Gates option in the strategy window. A gate, or strong link, is a connection between two rules that share the same data.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PTGATES***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PTGATES */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PTGATES to get the default strategy. It must be NXP_AINFO_PTGATES | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

t*hePtr* must point to an integer which will be set to 1 if forward chaining through gates is enabled, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PTGATES, thePtr).
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PFACTIONS	Forwarding of action effects.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PWFALSE

Purpose

This returns whether or not the context propagation is enabled on FALSE hypotheses (PWFALSE = Propagate When False).

It corresponds to the Forward Rejected Hypothesis option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PWFALSE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PWFALSE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PWFALSE to get the default strategy. It must be NXP_AINFO_PWFALSE | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must be NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the strategy is on, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PWFALSE, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFACTIONS	Forward action effects.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PWNOTKNOWN

Purpose

This returns whether or not the context propagation is enabled on NOTKNOWN hypotheses (PWNOTKNOWN = Propagate When Notknown).

It corresponds to the Forward Notknown Hypothesis option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PWNOTKNOWN***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PWNOTKNOWN */
```

```
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PWNOTKNOWN to get the default strategy. It must be NXP_AINFO_PWNOTKNOWN | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the strategy is on, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PWNOTKNOWN, thePtr).
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFACTIONS	Forward action effects.
NXP_AINFO_PWFALSE	Context propagation on False hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_PWTRUE

Purpose

This returns whether or not the context propagation is enabled on TRUE hypotheses (PWTRUE = Propagate When True).

It corresponds to the Forward Confirmed Hypothesis option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_PWTRUE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_PWTRUE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_PWTRUE to get the default strategy. It must be NXP_AINFO_PWTRUE | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the strategy is on, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_PWTRUE, thePtr).
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFACTIONS	Forward action effects.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_QUESTWIN

Purpose

This returns the NOIR question window name attached to a slot (string edited in the Question Win. field of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_QUESTWIN***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_QUESTWIN */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code is equal to NXP_AINFO_QUESTWIN.

desc must equal NXP_DESC_STR.

thePtr must point to a buffer where the NOIR Question Window name will be returned.

len is the maximum number of characters that can be written to thePtr.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_QUESTWIN, thePtr, len)
```

NXP_GetAtomInfo / NXP_AINFO_RHS

Purpose

This returns information about the actions of a rule or a method (the right-hand side). Each action's atomId is returned in thePtr and then you can get the text of the action using NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_RHS***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_RHS */
AtomId  optAtom;  /* ignored */
```

```
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid rule or method id (as returned by NXP_GetAtomId, for instance).

code is equal to NXP_AINFO_RHS.

If *optInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number of right hand side actions will be returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this last case, desc should be equal to NXP_DESC_ATOM and the right hand side actions with the index optInt will be returned in thePtr.

thePtr must be a pointer or an integer if optInt equals -1. Otherwise it must be a pointer to an atomId.

The right hand side actions are listed in the natural rule or method order (as they appear in the rule or method editor notebook).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid rule or method id.
NXP_ERR_INVARG4	optInt is not -1 nor a valid RHS action index.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT. optInt is a valid RHS action index but desc is not equal to NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_RHS, thePtr).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_RHS, optInt, thePtr)
```

Examples

The following examples illustrate how to get the text of a rule's actions using NXP_AINFO_RHS and NXP_AINFO_NAME:

```
AtomId       ruleId, actionId;
int          nActions, i;
char         *ruleName, col1str[20], col2str[1000], col3str[1000];

/* First get the rule's atomId in ruleId */
NXP_GetAtomId(ruleName, ruleId, NXP_ATYPE_RULE);
/* get the number of conditions in nCond */
NXP_GETLISTLEN(ruleId, NXP_AINFO_RHS, &nActions);
```

```
/* loop to get each condition's atomId and then its text */
for(i = 0; i < nActions, i++) {
       NXP_GETLISTELT(ruleId, NXP_AINFO_RHS, i, &actionId);

/* get the text of the 1st column (operator) in col1str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL1, NXP_DESC_STR, col1str, 20);

/* get the text of the 2nd column in col2str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL2, NXP_DESC_STR, col2str, 1000);

/* get the text of the 3rd column in col3str */
       NXP_GetAtomInfo(actionId, NXP_AINFO_NAME, (AtomId)0,
                     NXP_CELL_COL3, NXP_DESC_STR, col3str, 1000);
}
```

See Also

NXP_AINFO_CACTIONS	Getting the If Change methods
NXP_AINFO_PARENT	Getting the rule or method id from the LHS, RHS or EHS
NXP_AINFO_LHS	Getting the left hand side actions.
NXP_AINFO_EHS	Getting the Else right hand side actions.
NXP_AINFO_SOURCES	Getting the Order of Sources methods.

NXP_GetAtomInfo / NXP_AINFO_SELF

Purpose

This returns information (either the atom id or atom name) about the current SELF atom.

SELF only exists in a method (i.e. you can use NXP_AINFO_SELF in an execute routine called from a method). This function returns NULL in all other cases.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SELF***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_SELF */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;
```

code is equal to NXP_AINFO_SELF.

optInt describes which part of the atom will be returned. optInt can take one of the following values:

Code	Description
NXP_ATYPE_CLASS, NXP_ATYPE_OBJECT	Only the "obj" part is returned.
NXP_ATYPE_NONE, NXP_ATYPE_SLOT	The full "obj.prop" is returned.
NXP_ATYPE_PROP	Only the "prop" part is returned.

desc describes which format the information will be returned in. If desc equals NXP_DESC_STR, the atom name is returned in thePtr. If desc equals NXP_DESC_ATOM, the atomId is returned in thePtr.

thePtr must be a pointer to an atomId if desc is NXP_DESC_ATOM and a pointer to a buffer of characters if desc is NXP_DESC_STR.

len is the maximum number of characters that can be written to thePtr if desc is NXP_DESC_STR.

Note: The Rules Element does not check that NXP_ATYPE_OBJECT or NXP_ATYPE_CLASS matches the type (class or object) of SELF.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG4	The value of optInt is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_ATOM or NXP_DESC_STR.
NXP_ERR_INVARG6	thePtr is NULL.

If SELF does not exist, NXP_Error() will return NXP_ERR_NOERR. This is not an error condition.

Macros

None.

Examples

This call is useful if you want to write a generic FormInput or Show procedure called from a method; SELF information is not passed for these (for an Execute you can always pass SELF in the list of Atoms @ATOMID).

The following example illustrates how to get the full name of the SELF slot in the form "object.property":

```
char   slotName[100];

NXP_GetAtomInfo((AtomId)0, NXP_AINFO_SELF, (AtomId)0,
                    NXP_ATYPE_SLOT, NXP_DESC_STR, slotName,
100);
```

NXP_GetAtomInfo / NXP_AINFO_SLOT

Purpose

This returns information about the slots of a class or an object.

The slots atom ids are returned, not the property ids. Once you have a slot id you can get its property id with NXP_AINFO_PROP.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SLOT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_SLOT */
AtomId  optAtom;
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a valid class or object id.

code is equal to NXP_AINFO_SLOT.

If optAtom is NULL, it returns the list of slot ids attached to theAtom. You must first call with optInt = -1 to get the number n of slots in thePtr (desc = NXP_DESC_INT, or use the NXP_GETLISTLEN macro). Then you call with optInt = 0 to n-1 to get the atomId of the slot of index optInt (desc = NXP_DESC_ATOM, or use the NXP_GETLISTELT macro). Slots are not sorted.

If optAtom is not NULL, it should be a property id. NXP_GetAtomInfo will look for the slot belonging to the object or class theAtom with the property optAtom. In this case, desc must be equal to NXP_DESC_ATOM and thePtr must be pointing to an AtomId. If the slot is not found, thePtr will be set to NULL (this is not considered an error by the Rules Element).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid class or object id.
NXP_ERR_INVARG3	optAtom is not NULL but is not a valid slot id.
NXP_ERR_INVARG4	optInt is not equal to -1 or is not a valid parent object or class index and optAtom is NULL.

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	optInt is equal to -1 and desc is not equal to NXP_DESC_INT. optInt is a valid parent object index but desc is not equal to NXP_DESC_ATOM. optAtom is a valid slot id but desc is not equal to NXP_DESC_ATOM.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_SLOT, &len).
```

Then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_SLOT, i, thePtr)
```

Examples

The following example shows how to list the slot names of an object or a class:

```
AtomId theObj, theSlot, theProp;
int          nSlots, i;
char   propName[255];

/* get the number of slots */
NXP_GETLISTLEN(theObj, NXP_AINFO_SLOT, &len);
printf("This object has %d properties:\n", len);

for( i = 0; i < len; i++) {
       /* get the slot id */
       NXP_GETLISTELT(theObj, NXP_AINFO_SLOT, i, &theSlot);

       /* get the property id */
       NXP_GETATOMINFO(theSlot, NXP_AINFO_PROP, &theProp);

       /* get the property name */
       NXP_GETNAME(theProp, propName, 255);

       printf("%s\n", propName);
}
```

See Also

NXP_AINFO_PROP	Get the property id from the slot id.

NXP_GetAtomInfo / NXP_AINFO_SOURCES

Purpose

This returns the atom ids of the Order of Sources methods attached to a slot (methods specified as an Order of Sources type in the method editor).

The text can be obtained later with NXP_AINFO_NAME.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SOURCES***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid slot id.

code must be NXP_AINFO_SOURCES to get the public Order of Sources. It must be NXP_AINFO_SOURCES | NXP_AINFO_PRIVATE ("Or" operation sets the "private" bit) to get the private Order of Sources, i.e. methods that can't be inherited. It must be NXP_AINFO_SOURCES | NXP_AINFO_MLHS to get the conditions of the Order of Sources. It must be NXP_AINFO_SOURCES | NXP_AINFO_MRHS to get the right-hand side (Then) actions of the Order of Sources. And it must be NXP_AINFO_SOURCES | NXP_AINFO_MEHS to get the Else actions of the Order of Sources.

If *optInt* is equal to -1, desc should be equal to NXP_DESC_INT and the number n of Order of Sources is returned as an integer in thePtr. Otherwise, optInt should be a number between 0 and n-1. In this case, desc should be equal to NXP_DESC_ATOM and the Order of Sources with the index optInt is returned in thePtr.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is NULL or is not a valid slot id.
NXP_ERR_INVARG4	The value of optInt is not between -1 and n-1.
NXP_ERR_INVARG5	optInt equals -1 but desc does not equal NXP_DESC_INT, or optInt does not equal -1 and desc does not equal NXP_DESC_ATOM.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_SOURCES, thePtr).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_SOURCES, optInt, thePtr)
```

Examples

The following code gives a simple example.

```
int      i, nSources;
AtomId   atom, theSource;

/* returns the public OS methods of atom */
NXP_GETLISTLEN(atom, NXP_AINFO_SOURCES, &nSources);
for (i = 0; i < nSources; i++) {
       NXP_GETLISTELT(atom, NXP_AINFO_SOURCES, i, &theSource);
       ...
}

/* returns the private OS methods actions */
NXP_GETLISTLEN(atom, NXP_AINFO_SOURCES|NXP_AINFO_PRIVATE, &nSources);
for (i = 0; i < nSources; i++) {
       NXP_GETLISTELT(atom, NXP_AINFO_SOURCES|NXP_AINFO_PRIVATE, i,
                      &theSource);
       ...
}
       /* Use NXP_AINFO_NAME to get the text of the methods, see the
       example in NXP_AINFO_LHS */
```

See Also

NXP_AINFO_LHS	Get the list of conditions of a rule or a method.
NXP_AINFO_RHS	Get the list of actions of a rule or a method.
NXP_AINFO_EHS	Get the list of Else actions of a rule or a method.
NXP_AINFO_CACTIONS	Get the list of If Change actions.

NXP_GetAtomInfo / NXP_AINFO_SOURCESCONTINUE

Purpose

This returns whether or not the Order of Sources methods will be fully executed (Enable Order of Sources / Continue option of the Strategy dialog window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SOURCESCONTINUE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_SOURCESCONTINUE to get the default strategy. It must be NXP_AINFO_SOURCESCONTINUE |

NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation sets the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the Order of Sources methods are enabled and executed until the end of the actions list, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_SOURCESCONTINUE, thePtr).
```

Examples

The following code gives a simple example.

```
int   sourcescontinue;

/* returns in sourcescontinue the default strategy.
 * sourceson = 1 if Order of Sources are enabled on
 *  continue, 0 if they are disabled or enabled but no
 *  continue
 */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_SOURCESCONTINUE,
      &sourcescontinue);

/* returns in sourcescontinue the current strategy */
NXP_GETINTINFO((AtomId)0,
      NXP_AINFO_SOURCESCONTINUE | NXP_AINFO_CURSTRAT,
            &sourcescontinue);
```

See Also

NXP_Strategy	Change the default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_SOURCESON

Purpose

This returns whether or not Order of Sources methods are enabled (Enable Order of Sources option of the Strategy dialog window).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SOURCESON***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_SOURCESON to get the default strategy. It must be NXP_AINFO_SOURCESON | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation sets the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the Order of Sources methods are enabled, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_SOURCESON, thePtr).
```

Examples

The following code gives a simple example.

```
int   sourceson;

/* returns in sourceson the default strategy.
 * sourceson = 1 if Order of Sources are enabled, 0 if they are
disabled
 */
NXP_GETINTINFO((AtomId)0, NXP_AINFO_SOURCESON, &sourceson);

/* returns in sourceson the current strategy */
NXP_GETINTINFO((AtomId)0,
          NXP_AINFO_SOURCESON | NXP_AINFO_CURSTRAT,
&sourceson);
```

See Also

NXP_Strategy	Change the default or current strategy.

NXP_GetAtomInfo / NXP_AINFO_SUGGEST

Purpose

This returns whether or not a hypothesis is suggested.

(Use the code NXP_AINFO_FOCUSPRIO if you need more details on the focus priority of the hypothesis)

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SUGGEST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_SUGGEST */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom must be a hypothesis slot.

code is equal to NXP_AINFO_SUGGEST.

desc must equal NXP_DESC_INT.

thePtr must be a pointer to an integer which will be set to 1 if the hypothesis is suggested, and set to 0 otherwise.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a hypothesis atomId.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_SUGGEST, thePtr)
```

Examples

The following code gives a simple example.

```
AtomId       hypo;
int          suggested;

NXP_GETINTINFO(hypo, NXP_AINFO_SUGGEST, &suggested);
if(suggested) {
```

```
        ...
}
```

See Also

NXP_AINFO_FOCUSPRIO	Focus priority of a hypothesis.
NXP_BwrdAgenda	Forces Order of Sources of an atom.

NXP_GetAtomInfo / NXP_AINFO_SUGLIST

Purpose

This returns the list of hypotheses kept in the suggest selection of the knowledge base (list built with the Suggest / Volunteer command or by selecting hypotheses in the notebook).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_SUGLIST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* NXP_AINFO_SUGLIST */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

code is equal to NXP_AINFO_SUGLIST.

optInt is an integer between -1 and n-1.

The mechanism used to retrieve this list is the following:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number n of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with the NXP_AINFO_SUGLIST code and optInt set to any value between 0 and n-1 where n is the value returned by the first call. The id of the (optInt+1)th atom in the list will be returned in thePtr (which must be a pointer to an AtomId).

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN((AtomId)0, NXP_AINFO_SUGLIST, &len)
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT((AtomId)0, NXP_AINFO_SUGLIST, i, thePtr)
```

Examples

The following example shows how the Rules Element suggests all the hypotheses selected when you use the Suggest command in the interface.

```
AtomId       hypo;
int          len;

NXP_GETLISTLEN((AtomId)0, NXP_AINFO_SUGLIST, &len);

for( i = 0; i < len; i++) {
      NXP_GETLISTELT((AtomId)0, NXP_AINFO_SUGLIST, i, &hypo);
      NXP_Suggest(hypo, NXP_SPRIO_SUG);
}
```

See Also

NXP_AINFO_VOLLIST	List of slots selected for Volunteer.
NXP_Suggest	Suggests a hypothesis.

NXP_GetAtomInfo / NXP_AINFO_TYPE

Purpose

This returns the type of an atom (i.e. a class, an object, a rule, a method, etc.).

This is not the same thing as the data type of a value: use the code NXP_AINFO_VALUETYPE to get a data type (int, float, string, etc.).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_TYPE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_TYPE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid atom id.

code is equal to NXP_AINFO_TYPE.

desc must be NXP_DESC_INT.

thePtr is a pointer to an integer. The type of theAtom is returned as an integer in *thePtr. It can be any of the following values:

Code	Description
NXP_ATYPE_CACTIONS	If Change method
NXP_ATYPE_CLASS	Class

Code	Description
NXP_ATYPE_EHS	Else actions of a rule or a method.
NXP_ATYPE_KB	Knowledge base atom.
NXP_ATYPE_LHS	Conditions of a rule or of a method.
NXP_ATYPE_METHOD	Method atom.
NXP_ATYPE_OBJECT	Permanent object.
NXP_ATYPE_OBJECT|NXP_ATYPE_TEMP	Temporary object. (NXP_ATYPE_TEMP bit set)
NXP_ATYPE_PROP	Property.
NXP_ATYPE_RHS	Do actions (RHS) of a rule or a method.
NXP_ATYPE_RULE	Rule name.
NXP_ATYPE_SLOT	Slot (NXP_ATYPE_DATA and NXP_ATYPE_HYPO bits are set if the slot is a data or a hypothesis.)
NXP_ATYPE_SOURCES	Order of Sources method.

**thePtr* returns the same value for all slots (so data and hypotheses could be confused), and for permanent and temporary objects. To solve this problem, use the mask NXP_ATYPE_MASK to get the basic type (SLOT or OBJECT) and use AND operations with NXP_ATYPE_DATA, NXP_ATYPE_HYPO, NXP_ATYPE_TEMP to get more information. See the example below.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_TYPE, thePtr)
```

Examples

The following example shows how to handle types.

```
AtomId        atom;
int           theType;

NXP_GETINTINFO(atom, NXP_AINFO_TYPE, &theType);

/* Always mask type to get the low bits! */
switch( theType & NXP_ATYPE_MASK ) {

       case NXP_ATYPE_CLASS:  ...;/* class */
              break;
       case NXP_ATYPE_OBJECT:
              if(theType & NXP_ATYPE_TEMP)...  /* temporary object */
              else ...                    /* permanent object */
              break;
       case NXP_ATYPE_PROP:  ...;/* property */
```

```
            break;
      case NXP_ATYPE_SLOT:  ...;
            if(theType & NXP_ATYPE_DATA)...  /* data slot */
            else if(theType & NXP_ATYPE_HYPO)/* hypo slot */
            else ...            /* neither data nor hypo slot */
            break;
      case NXP_ATYPE_LHS:  ...;  /* condition */
            break;
      case NXP_ATYPE_RHS:  ...;  /* actions */
            break;
      case NXP_ATYPE_CACTIONS:  ...;/* if change */
            break;
      case NXP_ATYPE_SOURCES:  ...;/* order of sources */
            break;
      case NXP_ATYPE_KB:  ...;   /* id of a knowledge base */
            break;
      default:                   /* error! */
            break;
}
```

See Also

`NXP_AINFO_VOLLIST`	List of slots selected for Volunteer.
`NXP_Suggest`	Suggests a hypothesis.

NXP_GetAtomInfo / NXP_AINFO_VALIDENGINE_ACCEPT

Purpose

This returns whether or not the validation of values generated by the engine is turned on with systematic validation of data if the validation expression contains missing information.

- It corresponds to the Valid ENGINE ON / ACCEPT option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDENGINE_ACCEPT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDUENGINE_ACCEPT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDENGINE_ACCEPT to get the default strategy. It must be NXP_AINFO_VALIDENGINE_ACCEPT | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the engine values validation is ON / ACCEPT, set to 0 if the engine values validation is not ON / ACCEPT (that is either OFF or ON / REJECT).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDENGINE_ACCEPT, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATE	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side oof a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user.
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user is off and value accepted unchecked.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDENGINE_OFF

Purpose

This returns whether or not the validation of values generated by the system is turned off.

It corresponds to the VALID ENGINE OFF option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDENGINE_OFF***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDENGINE_OFF */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDENGINE_OFF to get the default strategy. It must be NXP_AINFO_VALIDENGINE_OFF | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the engine values validation is OFF, set to 0 if the engine values validation is ON.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDENGINE_OFF, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.

NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDENGINE_ON

Purpose

This returns whether or not the validation of values entered by the end user is turned on.

It corresponds to the Valid ENGINE ON / ACCEPT and ON / REJECT options in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDENGINE_ON***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDENGINE_ON */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDENGINE_ON to get the default strategy. It must be NXP_AINFO_VALIDENGINE_ON | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the engine values validation is ON (either ON / ACCEPT or ON / REJECT), set to 0 if the engine values validation is OFF.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDUSER_ON, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOW	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user off is off and value accepted unchecked.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected of validation expression contains missing information.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDENGINE_REJECT

Purpose

This returns whether or not the validation of values generated by the engine is turned on with systematic rejection of data if the validation expression contains missing information.

It corresponds to the Valid ENGINE ON / REJECT option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDENGINE_REJECT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDENGINE_REJECT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDENGINE_REJECT to get the default strategy. It must be NXP_AINFO_VALIDENGINE_REJECT | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the engine values validation is ON / REJECT, set to 0 if the engine values validation is not ON / REJECT (that is either OFF or ON / ACCEPT).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDENGINE_REJECT, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left- hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left- hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user.
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user is off and and value is accepted unchecked.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDEXEC

Purpose

This returns the user-defined data validation execute information attached to a slot (the name of the user-provided execute in the Data Validation Execute field of the meta-slot editor) returned as a string.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDEXEC***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALIDEXEC */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code is equal to NXP_AINFO_VALIDEXEC.

*des*c must equal NXP_DESC_STR.

*theP*tr must point to a buffer where the data validation execute name will be returned.

len is the maximum number of characters that can be written to thePtr.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_VALIDEXEC, thePtr, len)
```

NXP_GetAtomInfo / NXP_AINFO_VALIDFUNC

Purpose

This returns the boolean data validation expression information attached to a slot (the expression in the Data Validation Expression field of the meta-slot editor returned as a string).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDFUNC***,** *optAtom***,** *optInt***,** *desc***,**
*thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALIDFUNC */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code is equal to NXP_AINFO_VALIDFUNC.

desc must equal NXP_DESC_STR.

thePtr must point to a buffer where the boolean data validation expression string will be returned.

len is the maximum number of characters that can be written to thePtr.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_VALIDFUNC, thePtr, len)
```

NXP_GetAtomInfo / NXP_AINFO_VALIDHELP

Purpose

This returns the data validation error help information attached to a slot (string edited in the Data Validation Help field of the meta-slot editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDHELP***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALIDHELP */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot id.

code is equal to NXP_AINFO_VALIDHELP.

desc must equal NXP_DESC_STR.

thePtr must point to a buffer where the data validation error help string will be returned.

*len i*s the maximum number of characters that can be written to thePtr.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_VALIDHELP, thePtr, len)
```

NXP_GetAtomInfo / NXP_AINFO_VALIDUSER_ACCEPT

Purpose

This returns whether or not the validation of values entered by the end user is turned on with systematic validation of data if the validation expression contains missing information.

It corresponds to the Valid User ON / ACCEPT option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDUSER_ACCEPT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDUSER_ACCEPT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDUSER_ACCEPT to get the default strategy. It must be NXP_AINFO_VALIDUSER_ACCEPT | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the end-user values validation is ON / ACCEPT, set to 0 if the end-user values validation is not ON / ACCEPT (that is either OFF or ON / REJECT).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDUSER_ACCEPT, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user.
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user is off.

NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDUSER_OFF

Purpose

This returns whether or not the validation of values entered by the end user is turned off.

It corresponds to the Valid User OFF option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDUSER_OFF***,** *optAtom***,** *optInt***,**
*desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDUSER_OFF */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDUSER_OFF to get the default strategy. It must be NXP_AINFO_VALIDUSER_OFF | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

t*hePtr* must point to an integer which will be set to 1 if the end-user values validation is OFF, set to 0 if the end-user values validation is ON.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDUSER_OFF, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDUSER_ON

Purpose

This returns whether or not the validation of values entered by the end user is turned on.

It corresponds to the Valid User ON / ACCEPT and ON / REJECT options in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDUSER_ON***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDUSER_ON */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDUSER_ON to get the default strategy. It must be NXP_AINFO_VALIDUSER_ON | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

t*hePtr* must point to an integer which will be set to 1 if the end-user values validation is ON (either reject or accept in the case of missing information), set to 0 if the end-user values validation is OFF.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDUSER_ON, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.

NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user off.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_REJECT	Validation of data entered by the end user and value rejected if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine.
NXP_AINFO_VALIDENGINE_OFF	Validation of data generated by the engine is off.
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strategy	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALIDUSER_REJECT

Purpose

This returns whether or not the validation of values entered by the end user is turned on with systematic rejection of data if the validation expression contains missing information.

It corresponds to the Valid User ON / REJECT option in the strategy window.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALIDUSER_REJECT***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VALIDUSER_REJECT */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
```

```
Str     thePtr;
int     len;      /* ignored */
```

code must be NXP_AINFO_VALIDUSER_REJECT to get the default strategy. It must be NXP_AINFO_VALIDUSER_REJECT | NXP_AINFO_CURSTRAT to get the current strategy ("Or" operation with the "current" bit).

desc must equal NXP_DESC_INT.

thePtr must point to an integer which will be set to 1 if the end-user values validation is ON / REJECT, set to 0 if the end-user values validation is not ON / REJECT (that is either OFF or ON / ACCEPT).

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG2	code is invalid.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.
NXP_ERR_INVARG6	thePtr is NULL.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALIDUSER_REJECT, thePtr)
```

See Also

NXP_AINFO_EXHBWRD	Exhaustive backward chaining.
NXP_AINFO_PTGATES	Forward chaining through gates.
NXP_AINFO_PFMETHODELSEACTIONS	Forward chaining from else actions of a method.
NXP_AINFO_PFMETHODACTIONS	Forward chaining from left-hand-side and right-hand-side actions of a method.
NXP_AINFO_PFELSEACTIONS	Forward chaining from else actions of a rule
NXP_AINFO_PFACTIONS	Forward chaining from left-hand-side and right-hand-side of a rule.
NXP_AINFO_PWFALSE	Context propagation on False hypotheses.
NXP_AINFO_PWNOTKNOWN	Context propagation on Notknown hypos.
NXP_AINFO_PWTRUE	Context propagation on True hypotheses.
NXP_AINFO_VALIDUSER_ON	Validation of data entered by the end user
NXP_AINFO_VALIDUSER_OFF	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDUSER_ACCEPT	Validation of data entered by the end user and value accepted if validation expression contains missing information.
NXP_AINFO_VALIDENGINE_ON	Validation of data generated by the engine
NXP_AINFO_VALIDENGINE_ACCEPT	Validation of data generated by the engine and value accepted if validation expression contains missing information.

NXP_AINFO_VALIDENGINE_REJECT	Validation of data generated by the engine and value rejected if validation expression contains missing information.
NXP_Strateg	Change the inference strategy.

NXP_GetAtomInfo / NXP_AINFO_VALUE

Purpose

This returns the current value of an atom.

Note: Note: This call doesn't trigger any inheritance mechanism or Order of Source methods (i.e. if Obj.Prop is unknown when the call is made the engine won't process the OS methods of Obj.Prop or inherit its value from a parent class, it will simply return the value Unknown). See below for more information.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALUE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALUE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;
Str     thePtr;
int     len;
```

theAtom can be a slot, rule, RHS, EHS, LHS of a rule or a method. If it is a slot its value can take different data types, otherwise it is a boolean (i.e. it takes only the values Unknown, Notknown, True and False).

code is equal to NXP_AINFO_VALUE.

optAtom and *optIn*t are ignored.

desc specifies the data type in which the information should be returned. It is one of the NXP_DESC_XXX codes as described on the next page. The value of theAtom will be converted in this data type.

The information about the value of theAtom is returned in **thePtr*. thePtr must point to various types of variable depending on the value of desc.

len is used only if desc is NXP_DESC_STR. It represents the maximum number of characters that can be returned in the buffer *thePtr.

desc can be one of the following codes:

desc Code	Description
NXP_DESC_INT	thePtr should be a pointer to an integer. If theAtom refers to a boolean atom (LHS, RHS, EHS, boolean slot, . . .), and the value of theAtom is Unknown, *thePtr will receive the value NXP_BOOL_UNKNOWN (-2). NXP_BOOL_NOTKNOWN (-1) will be returned if the value is Notknown. NXP_BOOL_TRUE (1) or NXP_BOOL_FALSE (0) will be returned otherwise. If theAtom is not a boolean, NXP_GetAtomInfo will return an error if the value is Unknown or Notknown (use NXP_DESC_NOTKNOWN or NXP_DESC_UNKNOWN to test the value first). If not, the value will be converted to an integer and returned in thePtr. If the value is a floating point, this will be equivalent to rounding the value. If the value is a string, the Rules Element will try to interpret the string as being a number and will return its integer equivalent in thePtr.
NXP_DESC_DOUBLE	Same as NXP_DESC_INT except thePtr should be a pointer to a double floating point variable (64 bits, IIIE format on AT, DFloat on VAX, 64 bits on Mac - type `double` for MPW C and `short double` for THINK-C).
NXP_DESC_FLOAT	Same as NXP_DESC_INT except thePtr should be a pointer to a floating point variable (normal precision 32 bits). If theAtom is a boolean value, NXP_BOOL_UNKNOWN, NXP_BOOL_NOTKNOWN, TRUE, or FALSE will be returned in the float format into thePtr.
NXP_DESC_LONG	Same as NXP_DESC_INT except thePtr should be a pointer to a long integer. If theAtom is a boolean value, NXP_BOOL_UNKNOWN, NXP_BOOL_NOTKNOWN, TRUE, or FALSE will be returned in the long format into thePtr.
NXP_DESC_NOTKNOWN	This should be used to find out if a value is known but has the value Notknown. thePtr should be a pointer to an integer which will be set to TRUE if the value is Notknown, FALSE if it is not Notknown.
NXP_DESC_STR	In this case, thePtr should be a pointer to a string. A maximum of len characters will be returned in thePtr. If the value is Unknown, the string "UNKNOWN" will be returned, "NOTKNOWN" if the value is Notknown (or the format string defined for these special values). If theAtom is not a string, the value will be converted to a string according to the format associated with theAtom (applies to Date and Time as well).
NXP_DESC_UNKNOWN	This should be used to find out if a value is Unknown or not. thePtr should be a pointer to an integer which will be set to TRUE if the value is Unknown, FALSE if it is not Unknown.

desc Code	Description
NXP_DESC_VALUE	thePtr should be a pointer to a structure NXP_ValueRec defined in nxpdef.h. This structure will be filled with all the characteristics of the value of theAtom.

```
typedef struct NXP_ValueRec {
    int    Known
    int    Notknown
    int    Type
    union {
       int       Bool
       double    Numb
       double    Double
       long      Long
    } NVal;
    char   *_Str;
} NXP_ValueRec, *NXP_ValuePtr
```

If the value oftheAtom is known, thePtr->Known will be set to TRUE. If it isn't known, it will be set to FALSE.

If theAtom is notknown, thePtr->Notknown will be set to TRUE. If it isn't notknown, it will be set to FALSE.

thePtr->Type will be set to the type of value of theAtom (same as returned by NXP_GetAtomInfo with code =NXP_AINFO_VALUETYPE).

thePtr->NVal.Bool will be set if theAtom is boolean and known.

thePtr->NVal.Double will be set if theAtom is float and known.

thePtr->NVal.Long will be set if theAtom is integer and known (long or short integer).

If thePtr->Str is not NULL, it will point to a string where the string value will be returned. In this last case, for any atom type, the string equivalent is returned (same as desc = NXP_DESC_STR). A maximum of len characters will be copied.

Notes

To get more information about the Atom you are requesting the value from you can use NXP_AINFO_TYPE (type of atom) or NXP_AINFO_VALUETYPE (data type of its value).

int and long types are the same on most platforms (32 bits value) except on PC where int is 16 bits (i.e. if desc = NXP_DESC_INT a long value will be truncated to 16 bits).
Warning: On the Macintosh the Rules Element treats int as 32 bits (MPW C convention) so you must always use long integers if you are programming with THINK-C.

string values of a slot are returned using the format defined for that slot or for the corresponding property (as they are displayed in the interface).

float and double types can have different meaning depending on the platform and the programming environment. Check in your platform specific API manual.

date and time data types are not available in the API. You must set desc to the string descriptor NXP_DESC_STR. The date or time will be returned using the format defined for that slot, or the default format (for more information on formats see the Intelligent Rules Element Reference manual).

NXP_AINFO_VALUE doesn't trigger any inheritance mechanism in case the value of a slot is unknown when the call is made. The only way to force the engine to evaluate the Order of Sources of a slot (and thus use the inheritance if necessary) is to use the function NXP_BwrdAgenda and continue the session.

Warning: **T**his cannot work in "modal" routines such as Executes or a modal Question handler. You must return to the Rules Element to let the engine process the Order of Sources that NXP_BwrdAgenda pushed on top of the agenda.

See the appendix on NXP_AINFO_VALUE at the end of this manual for a discussion on common errors and more examples.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom id.
NXP_ERR_INVARG5	desc is not valid.
NXP_ERR_INVARG6	thePtr is NULL.
NXP_ERR_NOTKNOWN	desc is equal to NXP_DESC_INT, NXP_DESC_LONG, NXP_DESC_FLOAT or NXP_DESC_DOUBLE and theAtom is Notknown and is not boolean.
NXP_ERR_UNKNOWN	desc is equal to NXP_DESC_INT, NXP_DESC_LONG, NXP_DESC_FLOAT or NXP_DESC_DOUBLE and theAtom is Unknown and is not boolean.

Macros

You can use the NXP_GETXXXVAL macros:

```
NXP_GETINTVAL(atom, ptr)          /* Integer value */
NXP_GETDOUBLEVAL(atom, ptr)       /* Double value */
NXP_GETSTRVAL(atom, ptr, len)     /* String value */
NXP_GETUNKNOWNVAL(atom, ptr)      /* Unknown value */
NXP_GETNOTKNOWNVAL(atom, ptr)     /* Notknown value */
```

Examples

This example illustrates how to get the value of a slot in a generic way:

```
int           ret;
AtomId        theSlot;
int           intVal, boolVal, typeVal;
double        doubleVal;
Char          strVal[255];

/* get the type of value first */
ret = NXP_GETINTINFO(theSlot, NXP_AINFO_VALUETYPE, &typeVal);
if(ret == 0)  ...  /* error */

switch(typeVal) {
       case NXP_VTYPE_BOOL:
              ret = NXP_GETINTVAL(theSlot, &boolVal);
              if(boolVal == NXP_BOOL_UNKNOWN) ...
              if(boolVal == NXP_BOOL_NOTKNOWN) ...
```

```
            ...
            break;
      case NXP_VTYPE_LONG:
            /* For most platforms int and long are the same */
            /* Internally Rules Element only keeps a long value */
            ret = NXP_GETINTVAL(theSlot, &intVal);
            if (ret == 0) {  /* error */
                ret == NXP_Error( );
                if (ret == NXP_ERR_UNKNOWN) ...
                if (ret == NXP_ERR_UNKNOWN) ...
                ...
            }
            break;
      case NXP_VTYPE_DOUBLE:
            ret = NXP_GETDOUBLEVAL(theSlot, &doubleVal);
            if (ret == 0) {  /* error */
                ret == NXP_Error( );
                if (ret == NXP_ERR_UNKNOWN) ...
                if (ret == NXP_ERR_UNKNOWN) ...
                ...
            }
            break;
      case NXP_VTYPE_STR:
      case NXP_VTYPE_DATE:/* date retrieved as string */
      case NXP_VTYPE_TIME:/* time retrieved as string */
            /* If value is unknown or notknown the string will be
             * set using the format.  It may be better to check
             * first with NXP_DESC_UNKNOWN or NXP_DESC_NOTKNOWN
             */
            ret = NXP_GETSTRVAL(theSlot, strVal, 255);
            ...
            break;
      default:
            /* error! */
}
```

See Also

NXP_AINFO_VALUETYPE	Get the data type of an atom's value.
NXP_AINFO_TYPE	Get the type of atom.
NXP_Volunteer	Change the value of a slot.

NXP_GetAtomInfo / NXP_AINFO_VALUELENGTH

Purpose

This returns the length of a string slot value. It allows you to know how much memory space to allocate before retrieving the string with the code NXP_AINFO_VALUE.

C Format

The C format is as follows:

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALUELENGTH***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments:

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALUELENGTH */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is the string slot you want information on.

desc must be NXP_DESC_INT.

thePtr must point to an integer where the length will be returned.

All the other arguments are ignored.

The length is in bytes and includes the terminating NULL (C string). If the value is UNKNOWN or NOTKNOWN, it will return the length of the reserved string (e.g. "UNKNOWN" or "NOTKNOWN") plus the NULL byte. If a user-supplied format is available, that format will be applied in determining the string length. For user-specified formats, only string lengths up to 2K will be returned. For string slots with the default format (i.e. no format defined), there is no length restriction.

Notes

The Rules Element string slots are not limited in size.

You can test whether a slot has format information attached to it by calling `NXP_GetAtomInfo` with the code `NXP_AINFO_FORMAT`. It returns an empty string if no format is defined.

Return Codes

`NXP_GetAtomInfo` returns 1 on success and 0 on error. In case of error, more information about the error is obtained by calling `NXP_Error` immediately after the call which has failed. `NXP_Error` returns one of the following codes:

NXP_Error() Return Code	Explanation
`NXP_ERR_INVARG1`	theAtom is null or not a string.
`NXP_ERR_INVARG5`	desc is not a `NXP_DESC_ATOM`.
`NXP_ERR_INVARG6`	thePtr is null.
`NXP_ERR_NOERR`	Call was successful.

Macros

You can use the `NXP_GETINTINFO` macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALUELENGTH, thePtr)
```

Examples

The following example shows how to use `NXP_AINFO_VALUELENGTH` to allocate a buffer to retrieve the string value to:

```
AtomId  theSlot;
int     err, ret;
```

```
int      len, type;
Str      theStr;
/* Check that it is a string slot */
NXP_GETINTINFO(theSlot, NXP_AINFO_VALUETYPE, &type);
if (type != NXP_VTYPE_STRING) {
       ...
       /* not a string.  Exit */
}
/* Get the length */
ret = NXP_GETINTINFO(theSlot, NXP_AINFO_VALUELENGTH, &len);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
/* Allocate a buffer and get the value */
theStr = malloc(len);
ret = NXP_GETSTRVAL(theSlot, theStr, len);
if (ret == 0)
       {  err = NXP_Error(); ...  /* error code */ }
```

See Also

`NXP_GetAtomInfo / NXP_AINFO_VALUE`	returns the value of a slot.
`NXP_GetAtomInfo / NXP_AINFO_VALUETYPE`	returns the data type of a slot.

NXP_GetAtomInfo / NXP_AINFO_VALUETYPE

Purpose

This returns the data type of the value of an atom.

This is not the same thing as the type of an atom: use the code NXP_AINFO_TYPE to get the atom type (object, class, slot, rule, method, kb, etc.).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VALUETYPE***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_VALUETYPE */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_INT */
Str     thePtr;
int     len;      /* ignored */
```

theAtom is a valid atom id (either a property, a slot, a rule, a RHS, a EHS, a LHS or a method).

code is equal to NXP_AINFO_VALUETYPE.

desc must equal NXP_DESC_INT.

thePtr must be a pointer to a valid integer memory location where the data type code will be returned. The data type is returned as one of the following codes:

Code	Explanation
NXP_VTYPE_BOOL	Boolean (used for slots, rules, LHS, RHS, EHS, methods).
NXP_VTYPE_DATE	Date
NXP_VTYPE_DOUBLE	Floating point (same as type NXP_VTYPE_NUMB of version 1.0.)
NXP_VTYPE_LONG	Long integer (internally the Rules Element keeps integer values as long).
NXP_VTYPE_STR	String
NXP_VTYPE_TIME	Time
NXP_VTYPE_SPECIAL	Special type (returned only if theAtom is the Value property.)

If theAtom is a slot id its value can have any of the first six types.

If theAtom is a property id, the data type of the property will be returned. If theAtom is the id of the property Value (the default property), the type returned will be NXP_VTYPE_SPECIAL.

If theAtom is a rule, RHS, EHS, LHS, or method, it automatically has a boolean value.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error() immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atom id.
NXP_ERR_INVARG5	desc is not equal to NXP_DESC_INT.

Macros

You can use the NXP_GETINTINFO(atom, code, ptr) macro:

```
NXP_GETINTINFO(theAtom, NXP_AINFO_VALUETYPE, &type)
```

Examples

This example shows how to get the type of an atom (see a more detailed example in NXP_AINFO_VALUE):

```
int          ret, typeVal;
AtomId theAtom;

/* get the type of value first */
ret = NXP_GETINTINFO(theAtom, NXP_AINFO_VALUETYPE, &typeVal);
if(ret == 0)  ...  /* error */

switch(typeVal) {
       case NXP_VTYPE_BOOL: ...   break;
       case NXP_VTYPE_LONG: ...   break;
```

```
        case NXP_VTYPE_DOUBLE: ... break;
        case NXP_VTYPE_STR: ...    break;
        case NXP_VTYPE_DATE: ...   break;
        case NXP_VTYPE_TIME: ...   break;
        case NXP_VTYPE_SPECIAL: ...  break;
}
```

See Also

NXP_AINFO_VALUE	Get value of an atom.
NXP_AINFO_TYPE	Get the type of atom.
NXP_Volunteer	Change the value of a slot.

NXP_GetAtomInfo / NXP_AINFO_VERSION

Purpose

This returns the names and version numbers of the software components (the Rules Element, Client / Server, etc.) included in the package used. These are always returned as strings. For instance, the Development System adds a string such as “Intelligent Rules Element” at the top of Transcript by using this call.

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VERSION***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom; /* ignored */
int     code;    /* = NXP_AINFO_VERSION */
AtomId  optAtom; /* ignored */
int     optInt;  /* ignored */
int     desc;
Str     thePtr;
int     len;
```

*cod*e is equal to NXP_AINFO_VERSION.

n is always greater than or equal to 1, although optInt equal to 0 is a special case. The software component with index 0 is the full software package, and the full serial number is returned as it appears in the Transcript window. optInt equal to 1 signifies the first software component within the package, etc.

If *optInt* is -1, l*en* is ignored. Otherwise, len is the maximum number of characters that can be written to thePtr.

Return Codes

NXP_GetAtomInfo returns 1 on success and 0 on error. In case of error, you can obtain more information about the error by calling NXP_Error()

immediately after the failed call. NXP_Error() will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG4	The value of optInt is invalid.
NXP_ERR_INVARG5	optInt is equal to -1 but desc is not equal to NXP_DESC_INT, or optInt is between 0 and n-1 but desc is not equal to NXP_DESC_STR.

Macros

You can use the NXP_GETLISTLEN(atom, code, ptr) macro to get the number of software components:

```
NXP_GETLISTLEN((AtomId)0, NXP_AINFO_VERSION, thePtr)
```

Then use the NXP_GETLISTELTSTR(atom, code, index, ptr, len) macro to get individual versions:

```
NXP_GETLISTELTSTR((AtomId)0, NXP_AINFO_VERSION, i, thePtr,
len)
```

Examples

The following example illustrates how to display a Rules Element serial number in your application:

```
char   serialStr[100];

NXP_GetAtomInfo((AtomId)0, NXP_AINFO_VERSION, (AtomId)0,
                           0, NXP_DESC_STR, serialStr, 100);
```

or

```
NXP_GETLISTELTSTR((AtomId)0, NXP_AINFO_VERSION, 0, serialStr,
100);
```

NXP_GetAtomInfo / NXP_AINFO_VOLLIST

Purpose

This returns the list of slots kept in the volunteer section with the knowledge base (Suggest / Volunteer from the main menu or selection in the data notebook).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_VOLLIST***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;  /* ignored */
int     code;     /* = NXP_AINFO_VOLLIST */
AtomId  optAtom;  /* ignored */
int     optInt;
int     desc;
Str     thePtr;
int     len;      /* ignored */
```

code is equal to NXP_AINFO_VOLLIST.

optInt is an integer between -1 and n-1.

The mechanism to retrieve this list is the following:

- In your first call, you pass a value of -1 in the optInt argument. In return, thePtr is set to the number n of atoms in the list (thePtr must be a pointer to an integer).
- Then you can call NXP_GetAtomInfo with the NXP_AINFO_VOLLIST code and optInt set to any value between 0 and n-1 where n is the value returned by the first call. The id of the (optInt+1)th atom in the list will be returned in thePtr (which must be a pointer to an AtomId).

desc must be NXP_DESC_INT when optInt is -1, and NXP_DESC_ATOM otherwise.

thePtr must be a pointer to an integer when optInt is -1, and a pointer to an AtomId otherwise.

Macros

For the first call, use the NXP_GETLISTLEN(atom, code, ptr) macro:

```
NXP_GETLISTLEN(theAtom, NXP_AINFO_VOLLIST, thePtr).
```

then use the NXP_GETLISTELT(atom, code, index, ptr) macro:

```
NXP_GETLISTELT(theAtom, NXP_AINFO_VOLLIST, optInt, thePtr)
```

NXP_GetAtomInfo / NXP_AINFO_WHY

Purpose

This returns the why information attached to a slot, method, or rule (string edited in the Why field of the meta-slot editor, method editor, or the rule editor).

C Format

The C format is as follows.

int NXP_GetAtomInfo(*theAtom***,** *NXP_AINFO_WHY***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***,** *len***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;     /* = NXP_AINFO_WHY */
AtomId  optAtom;  /* ignored */
int     optInt;   /* ignored */
int     desc;     /* = NXP_DESC_STR */
Str     thePtr;
int     len;
```

theAtom must be a valid slot, method or rule id.

code is equal to NXP_AINFO_WHY.

desc must equal NXP_DESC_STR.

thePtr must point to a buffer where the why string will be returned.

len is the maximum number of characters that can be written to thePtr.

Macros

You can use the NXP_GETSTRINFO(atom, code, ptr, len) macro:

```
NXP_GETSTRINFO(theAtom, NXP_AINFO_WHY, thePtr, len)
```

NXP_GetAtomInfo / Examples

The following lines of C code show examples of calls to NXP_GetAtomInfo. This code could be improved by adding error checking and could be simplified by using the macros defined in nxpdef.h.

More examples are available with each NXP_AINFO_XXX code description.

In the following example we consider an object named "Object" with a property "Prop1"

```
AtomId        theObjId;
AtomId        theSlotId;
AtomId        thePropId;
AtomId        theRuleId;
AtomId        theAtom;
Char          theStr[255];
int           i;

/* Get the object id using NXP_GetAtomId */
NXP_GetAtomId("Object", &theObjId, NXP_ATYPE_OBJECT)

/* Get the name and the type from the id */
NXP_GetAtomInfo(theObjId, NXP_AINFO_NAME, (AtomId)NULL, 0,
                          NXP_DESC_STR, theStr, 255)
NXP_GetAtomInfo(theObjId, NXP_AINFO_TYPE, (AtomId)NULL, 0,
                          NXP_DESC_INT, (Str)&i, 0)
/* check that we get the same object back */
if (strcmp(theObjId, "Object") != 0 || i != NXP_ATYPE_OBJECT) {
  /* Problem! */
}

/* Now get the slot "Prop1" of this object */
NXP_GetAtomId("Prop1", &thePropId, NXP_ATYPE_PROP);
NXP_GetAtomInfo(theObjId, NXP_AINFO_SLOT, thePropId, 0,
                          NXP_DESC_ATOM, (Str)&theSlotId, 0)

/* another method is to use GetAtomId with "Object.Prop1"
 * This is not as fast because it has to parse the name */
NXP_GetAtomId("Object.Prop1", &theAtom, NXP_ATYPE_SLOT)
if (theAtom != theSlotId) {
     /* Problem! */
}

/* Now check some dependencies */
NXP_GetAtomInfo(theSlotId, NXP_AINFO_PROP, (AtomId)NULL, 0,
                          NXP_DESC_ATOM, (Str)&theAtom, 0)
if (theAtom != thePropId) {
    /* Problem! */
}
NXP_GetAtomInfo(theSlotId, NXP_AINFO_PARENT, (AtomId)NULL, 0,
                          NXP_DESC_ATOM, (Str)&theAtom, 0)
if (theAtom != theObjId) {
    /* Problem! */
```

```
}

/* Find theSlotId in the list of slot of the Objects */
NXP_GetAtomInfo(theObjId, NXP_AINFO_SLOT, (AtomId)NULL,
                          -1, NXP_DESC_INT, (Str)&i, 0)
while (--i >= 0) {
       NXP_GetAtomInfo(theObjId, NXP_AINFO_SLOT, (AtomId)NULL, i,
                                  NXP_DESC_ATOM, (Str)&theAtom, 0)
       if (theAtom == theSlotId) {
              /* found the Slot id in the list */
              goto noerr;
       }
}
/* Problem.if we come here! */

noerr:
/* returns the value of theSlotId in theStr */
NXP_GetAtomInfo(theSlotId, NXP_AINFO_VALUE, (AtomId)NULL, 0,
                           NXP_DESC_STR, theStr, 255)

/* get the choice list for theSlotId */
NXP_GetAtomInfo(theSlotId, NXP_AINFO_CHOICE, (AtomId)NULL, -1,
                           NXP_DESC_INT, (Str)&i, 0)
while (--i >= 0) {
       NXP_GetAtomInfo(theSlotId, NXP_AINFO_CHOICE, (AtomId)NULL, i,
                                  NXP_DESC_STR, theStr, 255)
       /* display theStr or do whatever */
       ...
}
/* get the current rule and display its contents */
NXP_GetAtomInfo((AtomId)NULL, NXP_AINFO_CURRENT, (AtomId)NULL,
                           NXP_ATYPE_RULE, NXP_DESC_ATOM, &theRuleId, 0);
/* get the hypothesis id */
NXP_GetAtomInfo(theRuleId, NXP_AINFO_HYPO, (AtomId)NULL, 0,
                           NXP_DESC_ATOM, (Str)&theAtom, 0);

/* get all the LHS strings in reverse order */
NXP_GetAtomInfo(theRuleId, NXP_AINFO_LHS, (AtomId)NULL,
                           -1, NXP_DESC_INT, (Str)&i, 0);
while (--i >= 0) {
       NXP_GetAtomInfo(theRuleId, NXP_AINFO_LHS, (AtomId)NULL, i,
                           NXP_DESC_ATOM, (Str)&theAtom, 0);
       /* get the operator string */
       NXP_GetAtomInfo(theAtom, NXP_AINFO_NAME, (AtomId)NULL,
                                        NXP_CELL_COL1, NXP_DESC_STR,
theStr, 255);
       /* get the first argument string */
       NXP_GetAtomInfo(theAtom, NXP_AINFO_NAME, (AtomId)NULL,
                                        NXP_CELL_COL2, NXP_DESC_STR,
theStr, 255);
       /* get the second argument string */
       NXP_GetAtomInfo(theAtom, NXP_AINFO_NAME, (AtomId)NULL,
                                        NXP_CELL_COL3, NXP_DESC_STR,
theStr, 255);
}
```

Chapter

5 NXP_SetAtomInfo Routine

This chapter describes the NXP_SetAtomInfo routine and the information codes associated with it.

NXP_SetAtomInfo

Purpose

NXP_SetAtomInfo provides some control over knowledge bases allowing to change information associated with individual atoms or entire knowledge bases. This function is the opposite of NXP_GetAtomInfo (although all possible codes are not implemented yet).

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *code***,** *optAtom***,** *optInt***,** *desc***,** *thePtr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;
AtomId  optAtom;
int     optInt;
int     desc;
Str     thePtr;
```

theAtom specifies the atom or knowledge base you want to modify. theAtom is an atomId obtained by a previous call to NXP_GetAtomId, NXP_GetAtomInfo or, in the case of knowledge bases, to NXP_LoadKB.

code specifies what is being changed. The different values for code are described in the following pages.

optAtom is an additional AtomId argument with different meanings depending on the value of code or is unused.

optInt is an additional integer argument with different meanings depending on the value of code or is unused.

desc is a code which describes the data type of the information or is unused.

thePtr points to the data associated with code or is unused.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed.

NXP_SetAtomInfo Codes List

Following are the NXP_SetAtomInfo codes in alphabetical order:

Code	Description
NXP_SAINFO_AGDVBREAK	Sets / unsets agenda break points on hypotheses.
NXP_SAINFO_CURRENTKB	Sets the current (or "default") knowledge base.
NXP_SAINFO_DISABLESAVEKB	Disables the saving of knowledge bases from the API.
NXP_SAINFO_INFBREAK	Sets / unsets inference break points on atoms.
NXP_SAINFO_INKB	Sets the knowledge base that an atom belongs to.
NXP_SAINFO_MERGEKB	Merges two knowledge bases into one.
NXP_SAINFO_PERMLINK	Changes the links of an atom to permanent.
NXP_SAINFO_PERMLINKKB	Changes all links in a knowledge base to permanent.

NXP_SetAtomInfo Codes By Categories

Following are the NXP_SetAtomInfo codes by categories.

Controlling the knowledge bases:

NXP_SAINFO_CURRENTKB	Sets the current (or "default") knowledge base.
NXP_SAINFO_DISABLESAVEKB	Disables the saving of knowledge bases from the API.
NXP_SAINFO_INKB	Sets the knowledge base that an atom belongs to.
NXP_SAINFO_MERGEKB	Merges two knowledge bases into one.

Setting/unsetting break points:

NXP_SAINFO_AGDVBREAK	Sets / unsets agenda break points on hypotheses.
NXP_SAINFO_INFBREAK	Sets / unsets inference break points on atoms.

Changing permanent/temporary links:

NXP_SAINFO_PERMLINK	Changes the links of an atom to permanent.
NXP_SAINFO_PERMLINKKB	Changes all links in a knowledge base to permanent.

These codes are described in detail in the following sections.

NXP_SetAtomInfo / NXP_SAINFO_AGDVBREAK

Purpose

This allows your program to set or unset agenda break points on a specific hypothesis. It is similar to clicking on the hypothesis name in the Agenda window (it puts ">" in front of the name).

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_AGDVBREAK***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;       /* = NXP_SAINFO_AGDVBREAK */
AtomId  optAtom; /* ignored */
int     optInt;
int     desc;    /* ignored */
Str     thePtr; /* ignored */
```

theAtom must be a valid hypothesis id.

code is equal to NXP_SAINFO_AGDVBREAK.

If *optInt* equals 1, the break point is set. If optInt equals 0, the break point is unset.

Notes

If the break point is set, the session will stop the next time the hypothesis' state changes and the Session Control window will show the break point. If you are using the runtime library, you will get the break point message through a SetData handler with winId = NXP_WIN_QUESTION.

Agenda break-points are different from inference break-points (which are set in a network window or with the code NXP_SAINFO_INFBREAK in the API). Agenda break-points give better control on hypotheses since they allow monitoring the changes of state rather than the changes of value.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a hypothesis.

Examples

Instead of interrupting the inference engine manually and setting / unsetting break points in the Agenda window, you can write an Execute routine that does it for you. You would call this routine from a rule or a method, passing the list of hypotheses for which the break point must be set or unset :

```
int           setBreakPoints(theStr, nAtoms, theAtoms)
Str           theStr;      /* flag "SET" or "UNSET" */
int           nAtoms;      /* number of hypos passed to Execute */
AtomId        *theAtoms;/* pointer to the list of hypotheses ids */
{
       int   set, i;

       if(strcmp("SET", theStr) == 0)
              set = 1;
       else
              set = 0;
              /* loop to set a break-point on each hypothesis */
       for( i = 0; i < nAtoms; i++) {
              NXP_SetAtomInfo(theAtoms[i], NXP_SAINFO_AGDVBREAK,
```

```
                                    (AtomId)0, set, 0, (Str)0);
        }
}
```

See also NXP_GetAtomInfo / NXP_AINFO_AGDVBREAK for an example of how to unset all break points on hypotheses in the knowledge base.

See Also

`NXP_GetAtomInfo / NXP_AINFO_AGDVBREAK`	Information on Agenda breakpoints.
`NXP_SetAtomInfo / NXP_SAINFO_INFBREAK`	Set inference breakpoints.

NXP_SetAtomInfo / NXP_SAINFO_CURRENTKB

Purpose

This sets the current (or "default") knowledge base to atom. The current knowledge base is the one used for every new creation (permanent objects or rules). This call is similar to using the "Set Knowledge Base..." command in the interface.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_CURRENTKB***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;       /* = NXP_SAINFO_CURRENTKB */
AtomId  optAtom;    /* ignored */
int     optInt;     /* ignored */
int     desc;       /* ignored */
Str     thePtr;     /* ignored */
```

theAtom must be a valid knowledge base Id.

code is equal to NXP_SAINFO_CURRENTKB. All other arguments are ignored.

Notes

By default, the current knowledge base is the last one loaded. If no KB has been loaded yet it is the special knowledge base untitled.kb (id = 2).

Temporary objects created during a Retrieve, for instance, belong to the knowledge base temporary.kb and not the current knowledge base. Also, undefined.kb (id = 0) and temporary.kb (id = 1) cannot be set to the current knowledge base.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error

immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a knowledge base Id.

Examples

The following example illustrates how to specify which knowledge base permanent objects should belong to (if you are creating permanent objects using NXP_Compile for instance):

```
KbId  theKB;

/* Get the KB id from the KB name */
NXP_GetAtomId("myKB.tkb", &theKB, NXP_ATYPE_KB);

/* Change the current KB */
NXP_SetAtomInfo(theKb, NXP_SAINFO_CURRENTKB, (AtomId)0, 0, 0, (Str)0);
```

See Also

NXP_SAINFO_INKB	Set the KB that an atom belongs to.
NXP_SAINFO_MERGEKB	Merge two knowledge bases into one.
NXP_GetAtomId,	Get a knowledge base Id.
NXP_GetAtomInfo / NXP_AINFO_KBID	Get a knowledge base Id.

NXP_SetAtomInfo / NXP_SAINFO_DISABLESAVEKB

Purpose

This allows you to disable the saving of knowledge bases from the API. You cannot enable it after you disable it. This call is useful if you are delivering a protected knowledge base and don't want it saved after your application decrypts it.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_DISABLESAVEKB***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;    /* ignored */
int     code;       /* = NXP_SAINFO_DISABLESAVEKB */
AtomId  optAtom;    /* ignored */
int     optInt;     /* ignored */
int     desc;       /* ignored */
Str      thePtr;    /* ignored */
```

code is equal to NXP_SAINFO_DISABLESAVE.

All other arguments are ignored.

Return Codes

There are no possible errors for this information code.

Examples

The following example illustrates how your application loads an encrypted knowledge base, supplies the password (with a password handler, for instance), and immediately uses NXP_SAINFO_DISABLESAVEKB so that nobody can save the decrypted knowledge base.

```
/* Loads the knowledge base
 * (see NXP_PROC_PASSWORD for an example of a password handler)
 */
NXP_LoadKB("myKB", &theKbId);

/* Disable the Save knowledge base */
NXP_SetAtomInfo( (AtomId)0, NXP_SAINFO_DISABLESAVEKB,
                    (AtomId)0, 0, 0, (Str)0);
```

NXP_SetAtomInfo / NXP_SAINFO_INFBREAK

Purpose

This allows your program to set / unset inference break points on any rule, condition, RHS, EHS, method (except filtered breaks), slot, object, class, or property. It is similar to clicking on the Atom name in the Network window with the Stop cursor. If a break point is set, the inference engine will stop the next time the value of the atom changes.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_INFBREAK***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;       /* = NXP_SAINFO_INFBREAK */
AtomId  optAtom;    /* ignored */
int     optInt;
int     desc;       /* ignored */
Str      thePtr;    /* ignored */
```

theAtom must be a valid atomId.

*code i*s equal to NXP_SAINFO_INFBREAK.

If *optInt* equals 1, the break point is set. If optInt equals 0, the break point is unset.

Notes

If you are using the Rules Element's interface, the inference break points stop the session and display an explanation in the Session Control window. If you are using the runtime library, you can get the break point message through a SetData handler with winId = NXP_WIN_QUESTION.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid atomId.

Examples

Instead of interrupting the inference engine manually and setting / unsetting break points in the Rules or Objects network window, you can write an Execute routine that does it for you. You would call this routine from a rule or a method, passing the list of atoms for which the break point must be set or unset :

```
int setBreakPoints(theStr, nAtoms, theAtoms)
Str      theStr;    /* flag "SET" or "UNSET" */
int     nAtoms;/* number of atoms passed to Execute: */
AtomId *theAtoms;/* pointer to the list of atoms */
{
       int   set, i;

       if(strcmp("SET", theStr) == 0
              set = 1;
       else
              set = 0;

       for( i = 0; i < nAtoms; i++){
              NXP_SetAtomInfo(theAtoms[i],
NXP_SAINFO_INFBREAK,
                           (AtomId)0, set, 0, (Str)0);
       }
}
```

See Also

NXP_GetAtomInfo / NXP_AINFO_INFBREAK	Information on Inference breakpoints
NXP_SetAtomInfo / NXP_SAINFO_AGDVBREAK	Set agenda breakpoints.

NXP_SetAtomInfo / NXP_SAINFO_INKB

Purpose

This sets the knowledge base that an atom belongs to. This is equivalent to using the command "Change KB" in the editors.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_INKB***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;       /* = NXP_SAINFO_INKB */
AtomId  optAtom;
int     optInt;     /* ignored */
int     desc;       /* ignored */
Str     thePtr;     /* ignored */
```

theAtom must be a valid object id, class id , rule id, etc.

code is equal to NXP_SAINFO_INKB.

optAtom must be a valid knowledge base Id.

Notes

Use the opposite function NXP_GetAtomInfo / NXP_AINFO_KBID to get the knowledge base that an atom belongs to.

You can use NXP_SAINFO_INKB to move temporary atoms out of the special kb temporary.kb, and make them permanent. This way they will not be deleted at the next restart session. If you also wish to keep the temporary links, you will need to use NXP_SAINFO_PERMLINK.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG3	optAtom is not a valid knowledge base Id.

Examples

The following example illustrates how to change the knowledge base of an object :

```
AtomId   theObj;
KBId     newKB;

/* sets the knowledge base of theObj to newKB */
NXP_SetAtomInfo(theObj, NXP_SAINFO_INKB, newKB, 0, 0, (Str)0);
```

If theObj was created as a temporary object (during a Retrieve operation, for instance), you would also need to change its links to class(es) to permanent with:

```
NXP_SetAtomInfo(theObj, NXP_SAINFO_PERMLINK, (AtomId)0,
                    0, 0, (Str)0);
```

See Also

NXP_SAINFO_CURRENTKB	Set the current knowledge base
NXP_SAINFO_MERGEKB	Merge two knowledge bases into one.
NXP_GetAtomId, NXP_AINFO_KBID	Get a knowledge base Id.
NXP_GetAtomInfo / NXP_AINFO_KBID	Get a knowledge base Id.

NXP_SetAtomInfo / NXP_SAINFO_MERGEKB

Purpose

This merges two knowledge bases into one by setting all the rules, objects and slots, etc. of the second one so they belong to the first one.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_MERGEKB***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;    /* = NXP_SAINFO_MERGEKB */
AtomId  optAtom;
int     optInt;  /* ignored */
int     desc;    /* ignored */
Str      thePtr; /* ignored */
```

theAtom must reference a valid knowledge base Id.

code is equal to NXP_SAINFO_MERGEKB.

optAtom must also reference a valid knowledge base Id. The knowledge base optAtom is merged into knowledge base theAtom, and optAtom is deleted from the list of knowledge bases.

All other arguments are ignored.

Notes

It is not possible to merge a knowledge base into the reserved knowledge bases undefined.kb (id = 0) and temporary.kb (id = 1), but is possible to merge undefined.kb and temporary.kb into another knowledge base.

NXP_SAINFO_MERGEKB is very useful if you want your temporary objects to become permanent. In this case, you would merge temporary.kb into your knowledge base.
Warning: As this call only affects the objects and classes and not the links, you will need to use NXP_SAINFO_PERMLINKKB to change the links from the objects and classes of the knowledge base to permanent.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid knowledge base Id.
NXP_ERR_INVARG3	optAtom is not a valid knowledge base Id.

Examples

The following code gives a simple example.

```
KBId   into, from;/* two knowledge base ids */

/* Merge KB "from" into KB "into" */
NXP_SetAtomInfo(into, NXP_SAINFO_MERGEKB, from, 0, 0, (Str)0);
```

See Also

`NXP_SAINFO_CURRENTKB`	Set the current knowledge base.
`NXP_SAINFO_INKB`	Set the knowledge base that an atom belongs to.
`NXP_GetAtomId, NXP_AINFO_KBID`	Get a knowledge base Id.
`NXP_GetAtomInfo / NXP_AINFO_KBID`	Get a knowledge base Id.

NXP_SetAtomInfo / NXP_SAINFO_PERMLINK

Purpose

This changes an object's temporary link(s) to permanent link(s); in other words, links are not deleted after a restart session. This has no effect if a link is already permanent.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_PERMLINK***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;    /* = NXP_SAINFO_PERMLINK */
AtomId  optAtom;
int     optInt;  /* ignored */
int     desc;    /* ignored */
Str      thePtr; /* ignored */
```

theAtom must be a valid object or class Id.

code is equal to NXP_SAINFO_PERMLINK.

optAtom can be NULL, or a valid object or class Id.

If optAtom is not NULL, this changes the temporary link between theAtom and optAtom to permanent. If optAtom is NULL, this changes all temporary links from theAtom to permanent.

Notes

This call does not create any links; use NXP_CreateObject to do so.

If the temporary link from or to a temporary object is changed to permanent, the object and the link will be deleted anyway when the session is restarted. You need to move the object from temporary.kb to a permanent knowledge base using NXP_SAINFO_INKB, or move all objects out of temporary.kb using NXP_SAINFO_MERGEKB.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid object or class Id.
NXP_ERR_INVARG3	optAtom is not NULL or not a valid object or class Id.

Examples

The following example illustrates how to change the knowledge base and links of a temporary object (created, for instance, during a Retrieve operation):

```
AtomId   theObj;
KBId     newKB;

/* Sets the knowledge base of theObj to newKB */
NXP_SetAtomInfo(theObj, NXP_SAINFO_INKB, newKB, 0, 0, (Str)0);

/* Changes all its links to permanent */
NXP_SetAtomInfo(theObj, NXP_SAINFO_PERMLINK, (AtomId)0, 0, 0,
                    (Str)0);
```

See Also

NXP_SAINFO_PERMLINKKB	Changes all the links in a knowledge base to permanent.

NXP_SetAtomInfo / NXP_SAINFO_PERMLINKKB

Purpose

This changes all links of objects / classes belonging to a knowledge base to permanent (it only affects temporary links).

This call is interesting after using NXP_SAINFO_MERGEKB to move all the temporary objects out of temporary.kb.

C Format

The C format is as follows.

int NXP_SetAtomInfo(*theAtom***,** *NXP_SAINFO_PERMLINKKB***,** *optAtom***,** *optInt***,** *desc***,** *ptr***);**

Arguments

The following list shows the valid arguments.

```
AtomId  theAtom;
int     code;    /* = NXP_SAINFO_PERMLINKKB */
AtomId  optAtom; /* ignored */
int     optInt;  /* ignored */
int     desc;    /* ignored */
Str      thePtr; /* ignored */
```

theAtom must be a valid knowledge base Id.

code is equal to NXP_SAINFO_PERMLINKKB.

Use NXP_SAINFO_PERMLINK if you just want to change the links of one atom.

Return Codes

NXP_SetAtomInfo returns 1 on success and 0 on error. In case of error, more information about the error can be obtained by calling NXP_Error immediately after the call which has failed. NXP_Error will return one of the following codes:

NXP_Error() Return Code	Explanation
NXP_ERR_INVARG1	theAtom is not a valid knowledge base Id.

Examples

The following example shows how to change all the links in theKB.

```
KBId   theKB;

NXP_SetAtomInfo(theKB, NXP_SAINFO_PERMLINKKB, (AtomId)0, 0, 0,
                    (Str)0);
```

See Also

NXP_SAINFO_PERMLINK	Changes the links of an individual atom.

Chapter

6 NXP_Edit Functions

This chapter describes the API you use to access and edit any atom in the Rules Element memory.

Introduction

The Rules Element allows developers to edit knowledge base atoms (classes, objects, rules, etc.) from the API via a mechanism that is improved over the old NXP_Edit() function. The new Edit API makes editing from the API much easier.

The new Edit API provides several levels of functionality:

- Functions which provide general structure handling to allow creation and deletion, as well as a reset of the basic data structure.
- Functions which fill and / or access the structure with relative safety.
- Editing functions that allow data to be fetched and atoms deleted, modified, or created.
- Dependencies information related to modifications can be retrieved before the modification is made.

Compatibility with Previous Releases

NXP_Compile() remains unchanged. It takes a string for input and submits it to the same compiler that the Rules Element uses for compiling TKB's. This approach has a limitation in that you cannot delete atoms. This can still allow you to modify classes, objects, and properties, but you can only add additional rules, for example, since the old ones will not be deleted using NXP_Compile(). It also does not provide much flexibility in dealing with issues like dependencies, etc. It simply takes a string / buffer and passes it to the Rules Element compiler. In terms of what it provides:

Capability	Class/Object/Property	MetaSlot	Rule/Method
`New`	Yes	Yes	Yes
`Edit`	Yes	Yes	No
`Copy`	No	No	No
`Delete`	No	No	No

The old NXP_Edit() simulates how the user would use the development interface to edit, for example, a rule. Through NXP_Edit, the user of this API is expected to have already installed a GetData handler in order to provide information for the "editor", as it is requested by the Rules Element. In some cases the user also needs to install a SetData and a Notify handler.

Technical Overview

The Edit API provides a logical view of the atoms that you can edit, which varies from the view seen in the Rules Element editors. The Edit API lets you define the following atom types within the knowledge base.

Atom Type

- Class
- Object
- Rule
- Method
- Property
- Hypo
- Slot

In all cases except Hypo, the usage is unambiguous. In the case of the Hypothesis it is necessary to specify whether its is defined in a Context or a Meta-Slot.

NxpEditRec Structure

One structure will be used for all editing of atoms via the API:

```
typedef structure _NxpEditRec {
      Int                AtomType;
      ArrayPtr           Id;
      ArrayPtr           Text;
      NxpEditInfoPtr     Errors;
      NxpEditInfoPtr     Dependencies;
} NxpEditRec, C_FAR * NxpEditPtr;
```

There will be a minimalist API for power users who wish to manipulate this structure directly (more efficiently), and a super-set of this which will provide more assistance for the more novice programmer.

Briefly, the members of the structure are as follows:

AtomType	describes the type of the atom.
Id	is an array of integers (element size = sizeof(ClientPtr)) that indicates what "logical" field is being described. Examples of such fields are some, but not all, of the NXP_AINFO_XXX fields.
Text	is an array of VStr's that contain the text (actually a VSTR) of what is described by the "Id".

Errors and *Dependencies* are pointers to an NxpEditInfoRec structure, where either error or dependency information will be reported.

```
typedef structure _NxpEditInfoRec {
      ArrayPtr     Codes;
      ArrayPtr     Strs;
      ArrayPtr     Atoms;
} NxpEditInfoRec, *NxpEditInfoPtr;
```

For instance, for dependency information:

Codes	This array would contain a code describing the type of dependency. This code is currently not public, it is the code used in the XREF modules (i.e.: XREF_V_HYPO).
Strs	This array would contain a VSTR which describes the dependency.
Atoms	This array would contain the AtomId of the dependent atom.

The use of the Arrays in this structure is optional. If any of the three ArrayPtr's is NULL, then that type of information will not be reported. In other words, in the above example, if the ArrayPtr Codes is NULL, then an error code would not be reported.

In all cases the Nth entries of each array will correspond to each other. In the case where an Nth entry is either not applicable or available, a place holder (NULL) entry will be inserted.

If the NxpEditInfoPtr in the NxpEditRec is NULL, then either dependencies or error information will not be reported.

There is a minimalist API to manipulate this structure as desribed in this chapter. See the Setting Up the Edit API section for details.

AtomType

In more detail, AtomType can specifically be one of the following:

`NXP_ATYPE_CLASS`	--> class edition
`NXP_ATYPE_OBJECT`	--> object edition
`NXP_ATYPE_PROP`	--> property edition
`NXP_ATYPE_SLOT`	--> meta-slot edition
`NXP_ATYPE_RULE`	--> rule edition
`NXP_ATYPE_METHOD`	--> method edition
`NXP_ATYPE_CONTEXT`	--> context edition

To determine the AtomType, the atom->Itself field is used, except in the case of a slot, where the user will have to set the AtomType using the NXP_EditSetAtomType() function. This is because for both NXP_ATYPE_CONTEXT and NXP_ATYPE_SLOT atoms, the Id field is a slot. In all cases, prior to calling NXP_EditCreate(), the AtomType will have to be set explicitly.

Id

The Id list for each atom is the set of values that can be edited for the specified AtomType. The Ids are identical to the TKB keywords used to define atoms in the editors. Note that items that are displayed in the graphical editors which are NOT editable do not have valid Ids (for example, all editors show the KB name; class, object, and property editors show attached methods, etc.).

The valid Id list for each AtomType includes the following:

Rule

The following atom ids can be defined for rules. In the case of the _LHS, _RHS, and _EHS ids, a single id defines one condition or action. When more than one condition or action is required to define the rule, multiple _LHS, _RHS, and _EHS ids need to be added to the atom edit structure. It is important to note that the system processes the conditions and actions in the order in which they have been defined in the atom edit structure. The order in which all other ids are defined in the atom edit structure is not important.

```
NXP_AINFO_NAME"myRule"
NXP_AINFO_LHS"Assign (|"hello world|") (message)"
NXP_AINFO_RHS"Assign (n+1) (n)"
NXP_AINFO_EHS"Reset (myhypo)"
NXP_AINFO_HYPO"myhypo"
NXP_AINFO_COMMENTS  "some comments"
NXP_AINFO_WHY       "a why string"
NXP_AINFO_INFATOM   "infslot.prop"
NXP_AINFO_INFCAT    "10"
```

Context

The following atom ids can be defined for contexts. The order in which these are defined in the atom edit structure is not important.

```
NXP_AINFO_NAME      "hyponame"
NXP_AINFO_CONTEXT   "contexthyponame"
```

Object

The following atom ids can be defined for objects. Note that multiple _CHILDOBJECT ids can be defined when the parent object contains more than one child object. The order in which these are defined in the atom edit structure is not important.

```
NXP_AINFO_NAME      "myObject"
NXP_AINFO_PARENTCLASS"ParentClass1"  allows one entry
NXP_AINFO_CHILDOBJECT"ChildObject1"  allows one entry
NXP_AINFO_PROPPUBLIC"publicProp1"  allows one entry
NXP_AINFO_PROPPRIVATE"privateProp1"  allows one entry
```

Note that _PARENTCLASS, _CHILDOBJECT, _PROPPUBLIC, and _PROPPRIVATE each allow one entry per definition.

Class

The following atom ids can be defined for classes. Note that multiple _CHILDCLASS ids can be defined when the parent class contains more than one child class. The order in which these are defined in the atom edit structure is not important.

```
NXP_AINFO_NAME      "myClass"
NXP_AINFO_CHILDCLASS"childClass1" allows one entry
NXP_AINFO_PROPPUBLIC"publicProp1" allows one entry
NXP_AINFO_PROPPRIVATE"privateProp1" allows one entry
```

Note that _CHILDCLASS, _PROPPUBLIC, and _PROPPRIVATE each allow one entry per definition.

Property

The following atom ids can be defined for properties. The order in which these are defined in the atom edit structure is not important.

```
NXP_AINFO_NAME      "myProp"
NXP_AINFO_TYPE      "Integer", "String", ...
```

Slot

The following atom ids can be defined for slots. The order in which these are defined in the atom edit structure is not important. Note, the graphical editor which allows you to define the slot is the Meta-Slot editor.

```
NXP_AINFO_NAME              "obj.prop"
NXP_AINFO_INFCAT            "10"
NXP_AINFO_INHCAT            "20"
NXP_AINFO_INFATOM           "infslot.prop"
NXP_AINFO_INHATOM           "inhslot.prop"
NXP_AINFO_INHUP             "TRUE" or "FALSE"
NXP_AINFO_INHDOWN           "TRUE" or "FALSE"
NXP_AINFO_INHVALUP          "TRUE" or "FALSE"
NXP_AINFO_INHVALDOWN        "TRUE" or "FALSE"
NXP_AINFO_PARENTFIRST       "TRUE" or "FALSE"
NXP_AINFO_BREADTHFIRST      "TRUE" or "FALSE"
NXP_AINFO_PROMPTLINE        "question prompt"
NXP_AINFO_QUESTWIN          "module.classname"
NXP_AINFO_COMMENTS          "some comments"
NXP_AINFO_WHY               "a why string"
NXP_AINFO_FORMAT            "mmm ddd yyyy" ...
NXP_AINFO_VALIDFUNC         "SELF.prop>0"
NXP_AINFO_VALIDEXEC         "extfunctionname"
NXP_AINFO_VALIDHELP         "brief help string"
NXP_AINFO_PRIVATEINITVAL    "1" or "red" or "TRUE"...
NXP_AINFO_PUBLICINITVAL     "1" or "red" or "TRUE"...
NXP_AINFO_PROPPUBLIC        "TRUE" or "FALSE"
```

Method

The following atom ids can be defined for methods. In the case of the _LHS, _RHS, and _EHS ids, a single id defines one condition or action. When more than one condition or action is required to define the method, multiple _LHS, _RHS, and _EHS ids need to be added to the atom edit structure. It is important to note that the system processes the conditions and actions in the order in which they have been defined in the atom edit structure. The order in which all other ids are defined in the atom edit structure is not important.

Note the _ATTACHEDTO and _TYPE Ids together specify the atom to which the method is attached. The _ATTACHEDTO Id defines the name of the specific object, class, property, or slot to which the method is attached. The _TYPE Id defines the AtomType of the object, class, property, or slot to which the method is attached. The _METHODFLAGS Id takes a value of either "PUBLIC" or "PRIVATE" and defines the privacy status of the method.

```
NXP_AINFO_NAME      "methodname"
NXP_AINFO_ATTACHEDTO"atomName_attached_to"
NXP_AINFO_TYPE      "SLOT" or "OBJECT", ...
NXP_AINFO_LHS       "Assign (|"hello world|") (message)"
NXP_AINFO_RHS       "Assign (n+1) (n)"
NXP_AINFO_EHS       "Reset (myhypo)"
NXP_AINFO_METHODFLAGS"PRIVATE" or "PUBLIC"
```

```
NXP_AINFO_ARGLIST   "@ARG1=_arg1; @NATURE=Slot..."
                    see TKB format
NXP_AINFO_COMMENTS  "some comments"
NXP_AINFO_WHY       "a why string"
```

Text

The Text member of the structure is an array of VSTRs that contains the text representation of what appears in the Id field. The text representation defines the atom for use in the knowledge base. To avoid redundant definitions, we reuse the TKB syntax for the Edit API. In the case of a complex atom such as a rule condition with a Retrieve, Write, or Execute operator, we recommend that you output the TKB from the Rules Element editors. Using this approach lets you rely on the interface to prompt you for the required definition and prevents typing errors from being introduced into the text definition.

In the case of conditions and actions, the text will be a simplified version of the TKB (missing outer parentheses). All other atoms are defined by string representations identical to the TKB syntax.

For example given that you want to define a method, you could generate the following simple TKB description from the Method Editor:

```
(@METHOD testmeth
 (@ATOMID=a.b;@TYPE=SLOT;)
 (@ARG1=_d;@NATURE=Slot;@TYPE=Boolean;@LIST;@DEFVAL=TRUE;)
 (@ARG2=_d1;@NATURE=Slot;@TYPE=Boolean;@DEFVAL=TRUE;)
 (@FLAGS=PRIVATE)
 (@LHS (Assign("lhs")(a.string))
)
```

The following example shows the Ids and strings for the above method when defined in the atom edit structure:

```
NXP_AINFO_NAME      "testmeth"
NXP_AINFO_ATTACHEDTO"a.b"
NXP_AINFO_TYPE      "SLOT"
NXP_AINFO_ARGLIST   "@ARG1=_d;@NATURE=Slot; @TYPE=Boolean;
@LIST;
                          @DEFVAL=TRUE;"
NXP_AINFO_ARGLIST"        @ARG2=_d1;@NATURE=Slot;
@TYPE=Boolean;
                                @DEFVAL=TRUE;"
NXP_AINFO_METHODFLAGS     "PRIVATE"
NXP_AINFO_LHS             "Assign("lhs")(a.string)"
```

Values supplied for the Ids must be quoted strings. Keyword definitions must appear in all caps.

Error Handling

During compilation, in case of an error, NXP_CPL_Error is set to one of the following values:

```
NXP_CPLERR_NOERR
NXP_CPLERR_NOMEM
NXP_CPLERR_CANCEL
NXP_CPLERR_INVSYNTAXLOW
NXP_CPLERR_INVSYNTAXUP
NXP_CPLERR_NEVER
```

These codes are used to build error information (strings) which supply the error description. This second set of codes (referred to in this document as the IDS_CPL_xxx codes) can be used in conjunction with CPL_Error. The IDS_CPL_XXX codes are currently found in ccompres.h. They are used to load string resources from ccompres.rc. The string list resource in ccompres.rc is called Messages. Messages currently contains both error and warning strings.

Setting up the Edit API

The following functions provide general structure handling to allow creation and deletion, as well as reset of the basic atom edit data structure.

NXP_EditDispose

Deallocates an existing atom edit structure and releases any memory.

void NXP_EditDispose (NxpEditPtr *atomdef***);**

Once the atom's edit fields are deallocated by EditDispose(), the structure cannot be reused and a new structure must be allocated. To reset the atom edit structure without disposing, use EditReset().

Note that this must be a structure allocated with EditNew(), and must not be stack based. Arrays will be reset to a size of 0, but the allocation will not be shrunk. Any VStr's will be deallocated. In particular, any pointers in the Text array will be disposed by VSTR_Dispose().

NXP_EditNew

Allocates memory for an atom with the standard edit fields.

NxpEditPtr NXP_EditNew (void);

EditNew() creates memory for an atom whose fields you wish to modify and returns a pointer to the new structure. The new atom edit structure contains the following fields:

AtomType	A constant which defines the type of the knowledge base atom: class, object, slot, property, method, rule, or context.
Id	An array that constants that specify the field of the knowledge base atom to be edited.
Text	An array of strings that contain the text definition of the field to be edited.
Errors	Pointer to the atom info structure which reports error information. See EditInfoNew().
Dependencies	Pointer to the atom info structure which reports dependency information. See EditInfoNew().

Once allocated, the fields can be immediately filled through the EditSetStr() function. The structure allocated must be disposed using EditDispose() to avoid memory leaks.

NXP_EditReset

Resets the fields of the atom edit structure for reuse.

void NXP_EditReset (NxpEditPtr *atomdef***);**

EditReset() lets you reuse the memory already allocated when more than one atom is to be modified. You can also create a new structure with EditNew(), but must remember to free memory associated with each structure you create using EditDispose().

Arrays will be reset to a size of 0, but the allocation will not be shrunk. Any VStr's will be deallocated. In particular, any pointers in the Text array will be disposed by VSTR_Dispose().

Receiving Error and Dependency Information

The following functions provide general structure handling to allow creation and resetting of the atom information data structure. The structure must be created and the Errors and Dependencies fields directly accessed in the atom edit structure before error and dependency information can be received.

NXP_EditInfoNew

Allocates memory for reporting errors and dependencies.

NxpEditInfoPtr NXP_EditInfoNew (void);

EditInfoNew() create memory for an atom that you want error and dependency information reported for. The new atom info structure contains the following fields:

Codes	An array that contains the codes describing the type of dependency resulting from a modify or delete edit operation.
Strs	An array that contains the string descriptions of the dependency resulting from a modify or delete edit operation.
Atoms	An array that contains the AtomIds of the atoms which are dependent on the atom being modified or deleted.

Initially these fields are defined as NULL. To begin receiving error and dependency information for a particular atom that you want to modify, you must first obtain a pointer to the Error and Dependencies fields of the atom edit structure:

```
NxpEditPtr NXP_Edit(void);
.
.
.
ptr->Dependencies=NXP_EditInfoNew();
ptr->Dependencies->Atoms=ARRAY_New();
NXP_EditDelete(ptr, ... );
```

NXP_EditInfoDispose

Deallocates an existing atom info structure and releases any memory.

void NXP_EditInfoDispose (NxpEditPtr *atomdef***);**

Once the atom's info fields are deallocated by EditInfoDispose(), the structure cannot be reused and a new structure must be allocated. To reset the atom edit structure without disposing, use EditInfoReset().

Note that this must be a structure allocated with EditInfoNew(), and must not be stack based. Arrays will be reset to a size of 0, but the allocation will not be shrunk. Any VStr's will be deallocated. In particular, any pointers in the Text array will be disposed by VSTR_Dispose().

NXP_EditInfoReset

Resets the fields of the atom info structure for reuse.

void NXP_EditInfoReset (NxpEditInfoPtr *atomerrinfo***);**

EditInfoReset() lets you reuse the memory already allocated when more than one atom is to be modified. You must remember to free memory associated with each structure you create using EditDispose().

Editing Capabilities

The following functions allow data to be fetched and atoms to be deleted, modified, or created. These functions may generate dependency information which you can have reported before completing the edit action. In order to have dependency information reported, you must allocate the atom info structure and access the Errors and Dependencies fields directly.

NXP_EditCreate

Creates a new atom in the knowledge base unless error information is specifically requested.

Int NXP_EditCreate (NxpEditPtr *atomdef***, AtomId C_FAR** * *atom***);**

EditCreate() will either immediately create the atom specified by AtomId within the knowledge base or it will generate a list of compilation errors for you to receive before completing the action. It is important to note that unless error information is specifically requested, EditCreate() automatically creates that atom. If the create operation is allowed, the atom whose definition you want to create will be based on the specified atom edit structure.

Note that *atom* is a pointer to an AtomId, and a value will be returned in *atom*, if the create request succeeds.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for a NULL atom pointer

Code	Description
NXP_ERR_COMPILEPB	There was a problem modifying the atom. If not NULL, the Errors ArrayPtr is provided in the atom edit structure and error information will be reported.

For EditCreate() if the atom edit structure passed is invalid, NXPError can be set to the following errors:

Code	Description
NXP_ERR_NULLEDPTR	The edPtr is NULL.
NXP_ERR_NULLDATA	The Text or Id array in the edPtr is 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 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.

NXP_EditDelete

Deletes the specified atom from the knowledge base unless dependency information is specifically requested.

Int NXP_EditDelete (NxpEditPtr *atomdef***, AtomId** *atom***);**

EditDelete() will either immediately delete the atom specified by AtomId from the knowledge base or it will generate a list of dependencies for you to receive before completing the action. It is important to note that unless dependency information is specifically requested, EditDelete() automatically deletes the atom.

If dependency information is requested, EditDelete() will return the list of dependency problems in the Dependencies field of the atom edit structure. The existence of any dependencies will result in the atom not being deleted. This gives you a chance to review the list of atoms which will be affected by deleting the specified atom. You might want to modify the knowledge base to remove the dependencies before completing the delete operation. If no dependencies were found to exist for the specified atom, the system completes the delete operation immediately.

You can force the system to delete an atom that has dependencies by resetting the Dependencies field of the atom edit structure to NULL. If a NULL NxpEditPtr or Dependencies field is provided, then it will be understood that dependencies are to be ignored, and the specified atom will be deleted.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for an unsupported or NULL atom
NXP_ERR_INVARG2	for a Dependencies ArrayPtr field that already contains something. The Dependencies field must be reset to NULL before dependencies can be received.
NXP_ERR_DEPENDENCIES	in the case of dependencies and a non-NULL "Dependencies". If a NULL Dependencies field or NxpEditPtr is provided, this error will never be returned.

NXP_EditFill

Fills the atom edit structure with an existing atom's definition.

Int NXP_EditFill (NxpEditPtr *atomdef***, AtomId** *atom***);**

EditFill() lets you specify the AtomId of an atom such as a class, object, rule, method that already exists in the knowledge base to define the fields of the atom edit structure. The atom edit structure passed must be valid and empty before it will be filled with the various fields corresponding to the atom. The atom edit structure which contains the existing atom definition can then be used for display or modification purposes.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for an unsupported or NULL atom
NXP_ERR_INVARG2	for a NULL atom edit structure, or an atom edit structure that already contains something.

In case of error, the structure passed in will remain unchanged.

If you want to define the atom edit structure with an existing slot, which can include either context or meta-slot information, the AtomType field must be filled in prior to calling EditFill(). This differentiates between context and meta-slot information. It will be filled with NXP_ATYPE_CONTEXT or NXP_ATYPE_SLOT respectively.

NXP_EditModify

Modifies the definition information of an existing atom unless dependency information is specifically requested.

Int NXP_EditModify (NxpEditPtr *atomdef***, AtomId** *atom***, AtomId** **newAtom***);*

EditModify() will either immediately modify the atom specified by AtomId within the knowledge base or it will generate a list of dependencies for you to receive before completing the action. It is important to note that unless dependency information is specifically requested, EditModify() automatically modifies that atom. If the modification is allowed, the atom

whose definition you want to modify will be based on the specified atom edit structure.

If dependency information is requested, EditModify() will return the list of dependency problems in the Dependencies field of the atom edit structure. The existence of any dependencies will result in the atom not being modified. This gives you a chance to review the list of atoms which will be affected by modifying the specified atom. You might want to modify the knowledge base to remove the dependencies before completing the modify operation. If no dependencies were found to exist for the specified atom, the system completes the modify operation immediately.

You can force the system to modify an atom that has dependencies by resetting the Dependencies field of the atom edit structure to NULL. If a NULL NxpEditPtr or Dependencies field is provided, then it will be understood that dependencies are to be ignored, and the specified atom will be modified.

Note that *newAtom* is a pointer to an AtomId, and a value will be returned in *newAtom*, if the create request succeeds. Normally EditModify() returns the same atomId specified when the modification is completed. In the case of rules, however, the system first deletes the old atom and creates a new atom for the modified rule. When modifying a rule, be sure to note the *newAtom* returned will be different from the atom specified for modification.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for an unsupported or NULL atom
NXP_ERR_INVARG2	for a NULL newAtom
NXP_ERR_INVARG3	for a Dependencies ArrayPtr field that already contains something. The Dependencies field must be reset to NULL before dependencies can be received.
NXP_ERR_NOTFOUND	for an atom that wasn't found
NXP_ERR_DEPENDENCIES	in the case of dependencies and a non-NULL "Dependencies". If a NULL "Dependencies" is provided, this error will never be returned.
NXP_ERR_COMPILEPB	There was a problem modifying the atom. If not NULL "Errors" ArrayPtr is provided in "rec", then error information will be reported.

For the EditModify() if the atom edit structure passed is invalid, NXPError can be set to the following errors:

Code	Description
NXP_ERR_NULLEDPTR	The edPtr is NULL.
NXP_ERR_NULLDATA	The Text or Id array in the edPtr is 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 AtomType of the edPtr.

Code	Description
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.

Setting and Querying the Atom Definition

The following functions let you fill and/or access the atom edit structure with relative ease. Using these functions ensures that users will not introduce errors to the ArrayPtr fields which result from accessing the atom edit structure and its contents directly.

NXP_EditFindInstance

Retrieves the index number of the value that matches the specified Id and string.

Int NXP_EditFindInstance (NxpEditPtr *atomdef***, Int** *Id***, Str** *value***, Int** * *index***);**

EditFindInstance() retrieves the *index* of the specified atom *Id* and string *value* in the atom edit structure passed. This function is used to determine which index is assigned to a given Text field Id and value.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	the instance was located and returned successfully
NXP_ERR_NOTFOUND	a matching instance could not be found.
NXP_ERR_INVARG1	the edPtr is invalid
NXP_ERR_INVARG3	the value is NULL
NXP_ERR_INVARG4	the instance is NULL

NXP_EditGetNthStr

Retrieves the value of the atom Id which matches the index specified.

Int NXP_EditGetNthStr (NxpEditPtr *atomdef***, Int** *id***, Str*** *value***, Int** *index***);**

EditGetNthStr() retrieves the *value* of the specified atom *Id* which contains multiple definition statements in the atom edit structure passed. For example, a class or object atom might have several properties, or a rule can have more than one condition defined in the atom edit structure. The atom Ids which can have more than one input include:

For rules and methods:

```
NXP_AINFO_LHS
NXP_AINFO_RHS
NXP_AINFO_EHS
```

For classes:

NXP_AINFO_CHILDCLASS

For objects:

NXP_AINFO_CHILDOBJECT

Since multiple inputs can be specified in these cases, the particular instance of the Id to retrieve is determined by the *index* number specified. The index is 0-based, so the first id definition in the list of multiple Ids is 0. If an index is specified that is higher than exists or the contents of the field is empty, then a NULL pointer will be returned. Otherwise, if the index exists, the user will be returned a pointer to the string part of the Text field present in the atom edit structure. The user is NOT allowed to modify the contents of this string.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for a NULL rec pointer
NXP_ERR_INVARG2	for an invalid value for "id" (this will depend on AtomType).
NXP_ERR_INVARG3	for instance less than zero.

NXP_EditGetStr

Retrieves the value of the atom Id specified.

Int NXP_EditGetStr (NxpEditPtr *atomdef***, Int** *id***, Str*** *value***);**

EditGetStr() retrieves the *value* of the specified atom *Id* in the atom edit structure passed. As a special case, *value* is returned as a NULL pointer if the contents of the field had been empty (i.e.: no field existed in the current structure). The user will be returned a pointer to the string part of the Text field present in the atom edit structure. The user is NOT allowed to modify the contents of this string.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for a NULL rec pointer
NXP_ERR_INVARG2	for an invalid value for "id" (this will depend on AtomType).

NXP_EditRemoveNthStr

Removes the atom Id and its text string which matches the index specified.

Int NXP_EditRemoveNthStr (NxpEditPtr *atomdef***, Int** *Id***, Int** *index***);**

EditRemoveNthStr() removes the value of the specified atom Id which contains multiple definition statements in the atom edit structure passed. The *index* parameter determines which instance of the Id should be

removed. The matching is done on Id. The index is 0-based, so the first Id definition in the list of multiple Ids is 0.

Use EditFindInstance() to obtain the index value of a particular atom Id definition statement.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	The instance was located and removed successfully
NXP_ERR_NOTFOUND	A matching instance could not be found.
NXP_ERR_INVARG1	The edPtr is invalid
NXP_ERR_INVARG3	The instance is invalid.

NXP_EditRemoveStr

Removes the atom Id specified and its text string.

Int NXP_EditRemoveStr (NxpEditPtr *atomdef***, Int** *Id***);**

EditRemoveStr() removes the value of the specified atom Id and its associated string definition in the atom edit structure passed. The matching is done on Id. In the case where more than one Array entry contains the Id, only the first one will be removed.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	A matching entry was located and removed.
NXP_ERR_NOTFOUND	A matching entry could not be found.
NXP_ERR_INVARG1	The edPtr is invalid

NXP_EditSetAtomType

Sets the value of AtomType in the atom edit structure specified.

Int NXP_EditSetAtomType (NxpEditPtr *atomdef***, Int** *type***);**

EditSetAtomType() sets the value of the AtomType field to the constant *type* in the atom edit structure passed. The field will be set regardless of its current value. AtomType will only be set if the atom edit structure is a valid, and new or reset structure, and type is a valid value.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for a NULL rec pointer, or a "rec" that already contains something
NXP_ERR_INVARG2	for an invalid value for "type"

NXP_EditSetNthStr

Sets the value of the atom Id which matches the index specified.

Int NXP_EditSetNthStr (NxpEditPtr *atomdef***, Int** *id***, Str** *value***, Int** *index***);**

EditSetNthStr() sets the value of the specified atom *Id* which contains multiple definition statements to the quoted string *value* in the atom edit structure passed. For example, a class or object atom might have several properties, or a rule can have more than one condition defined in the atom edit structure. The atom Ids which can have more than one input include:

For rules and methods:

```
NXP_AINFO_LHS
NXP_AINFO_RHS
NXP_AINFO_EHS
```

For classes:

```
NXP_AINFO_CHILDCLASS
```

For objects:

```
NXP_AINFO_CHILDOBJECT
```

Since multiple inputs can be specified in these cases, the particular instance of the Id to set is determined by the *index* number specified. The index is 0-based, so the first id definition in the list of multiple Ids is 0. If an index is specified that is higher than exists, then value will be added to the end. Otherwise, if the index exists, it will be changed.

Upon successful return, the contents of the string will have been captured in the structure, and the user is free to re-use, de-allocate, or do whatever with the string that is passed in.

The return code will be 1 for success, 0 for failure. At completion of this call, NXPError will be set as follows:

Code	Description
NXP_ERR_NOERR	for success
NXP_ERR_INVARG1	for a NULL rec pointer
NXP_ERR_INVARG2	for an invalid value for "id" (this will depend on AtomType).
NXP_ERR_INVARG3	for instance less than zero.

NXP_EditSetStr

Sets the value of the specified atom Id.

Int NXP_EditSetStr (NxpEditPtr *atomdef***, Int** *id***, Str** *value***);**

EditSetStr() sets the value of the specified atom *Id* to the quoted string *value* in the atom edit structure passed. The field will be set regardless of its current value. Text will only be set if the atom edit stru

Chapter

7 NXP_Context Functions

This chapter describes the API you use to invoke independent Intelligent Rules Element sessions.

Introduction

Users can use the new context switching API to invoke an independent session of the Rules Element while already in a session, and not have the knowledge bases / name spaces collide. For example, an application may be in a question handler, and in order to answer the question, it may be necessary to run another KB to get the answer.

In previous releases it was possible, but usually required use of the Rules Element journaling capability (saving a session to a file, and subsequently restoring it to the state at the time of the save). This exacted a heavy penalty, both in terms of use and performance. Even more difficult to control is the case when someone is using the Rules Element in an application that is using a tool or capability that may also be using the Rules Element. In this case, there is again the real possibility of "overlap", and no control over the names or agenda may be possible. This scenario will usually be in a single-user-oriented system or application.

Making the Rules Element application "partitionable" to work on the server side of a client-server application though the context switching API offers many advantages. In previous releases it was easier to put them on the client side, because to put the Rules Element on the server side meant either using a separate connection / process per client, or extensive use of the Rules Element journaling capabilities. The new context switching API offers the control of client-server functionality while improving performance over the old solutions.

Audience

Users needs to be able to develop a rules and objects-based application without concern of collision with anyone else and/or be able to provide a server-based environment that can handle multiple clients from a single process.

Specific Features

The context switching API includes the following features:

Multiple Sessions

It is possible to run multiple independent sessions without the various sessions impacting one another.

User Compatibility

Backward compatibility at the user level is provided. Developers can continue to use their existing knowledge bases (KB's) and API's. Users who know they may exist in a multi-session environment can use the new without affecting applications that use the old API.

Performance

In terms of both memory and CPU usage, there is a minimal performance impact whether the feature is used or not.

HW/SW Requirements

Specialized hardware and / or software (e.g., a multi-threaded environment) is not necessary.

Debugging

The new API allows the user to debug their application.

Context Switching Overview

In a context switched environment, the contexts are user defined and user controlled / switched. The context can only be switched by the user (i.e., “manually”), and can only be switched at certain times. For example, there can be no context switch by the user during inference engine cycles, or during a load or save of a KB. Another important difference is that there is only one stack. A local / stack-based variable in a routine exists in only one place, shared by all contexts. Switching must be done at controlled times.

This implementation of the context switch does not include the GUI or Database portions of the ND code / libraries. It is not normally necessary to “thread” the GUI portion anyway since on a server, there is typically no GUI; on a client, you can keep the GUI enshrouded and spawn off threads.

Switching Limitations

Since the context can only be switched in user code, this removes a lot of issues with maintaining the integrity / consistency of the engine along with the context currently executing. One problem that can occur is in a user installed handler. For example, a user handler could be invoked during an inference session, and the user could switch the context inside and return in a different context than the one we had at entry. In a true multi-threaded environment, this would not be an issue, since each thread has its own stack, registers (or copy thereof), etc. In this context case, however, we may have entered with N items being processed, and return where fewer than N items even exist. If we were in a loop, this would cause fatal memory access problems.

The only way to avoid this is that the Rules Element will “remember” the context block in use when entering a handler (i.e.: calling back to user code), and require that context block be the one in use when the handler returns. During execution of the handler, the user can temporarily invoke a new context block if desired, but will have to restore the original before returning from their code. If a context is not properly restored, the Rules Element will

issue an error, and restore the proper context. If this context is not available (e.g.: the user may have destroyed it in their handler), an ERR_Fail() will be executed to anyone prepared to attempt to recover.

Users will have to be aware of this issue in other places where external access is allowed outside the Rules Element scope (e.g.: script verbs).

Context Scope

This context block is currently intended to be limited in scope to the “pure” runtime. This means, basically, that the NXP library will be context switchable, but no other library (such as GUI, database, and execute).

NDCORE and NDRES will not be able to have contexts using this mechanism. The statics and (few) externs would be problems in a threaded environment, but probably not in a context-based environment. For example, there are some static fields for the FILE module, and the error handling, but these are acceptable in a context environment where the switches occur at discrete times (plus, the Rules Element does not rely much on the OI error-handling mechanism). In terms of the resource manager, there is again a single common resource root, but for the Rules Element's purposes, it only contains the “static” string resources of the Rules Element, and this can be considered as a readonly access that is constant from context to context. One possible problem (from memory) is that there may be an NDCORE static variable to determine which the Rules Element format to use: database or user-display. This may have to be cached in the context block

We cannot allow any graphics (e.g.: NOIR, NXGFX, NDVGM, NDTKIT, ...) because any context that used graphic resources could collide with any other. It is possible, however, for the same interface to have multiple the Rules Element contexts underneath.

A similar quick look at NXDB (Database and Flat file access) indicates that this would probably be a big effort to include in the context mechanism. Part of this is due to the large number of statics used in some of the modules (specifically the flat files); part is due to the lack of a context mechanism already when database handlers are called; and part is due to uncertainty with the external linkage with things like Data Access, Oracle, etc.

For now, SCRPT and SCVRB (scripts in general) cannot be added since there is a single lookup for verbs, event definitions, etc.

For those libraries that are not “active” in the context mechanism, it simply means that they can use the NXP library in a context mode, but they themselves cannot be customized on a context by context basis. For example, an Oracle connection pointer would appear as common to all contexts (some database software won't even allow multiple connections from a single process to a local database).

Contents of the Context Block

The context block currently in the Rules Element contains a number of (previously) extern variables, now accessed through the context record / C structure.

Logic Flow When Manipulating Contexts

Creating a context uses the following program logic:

```
cntx = NXP_ContextNew();
NXP_Control(NXP_CTRL_INIT);  /* mandatory */
set some visual clues or start another session
       (suggest/volunteer/knowcess)
attach multiple clientdata to a context. If you attach only
       one, you still need to get an Id.
```

To achieve best performance results, it is recommended that you globally initialize your contexts up-front (for instance, at start-up) if you know how many contexts will be used.

One problem that can occur is in a user installed handler. For example, a user handler could be invoked during an inference session, and the user could switch the context inside and return in a different context than the one we had at entry. Users will have to be aware of this issue in other places where external access is allowed outside the Rules Element scope (for example: script verbs).

The best way to deal with this is to stop the session and set a flag. When back to the main loop, you check the flag to know what context operation to do. Switching contexts within an Execute even after a stopped session, will make the new context fail.

Example code to pop out and push down contexts through the Execute:

```
static Int S_Switch = 0
static Int S_Running = 0
static Int C_FAR MyHandler (...)
{
       NXP_Control(NXP_CTRL_STOPSESSION);
       S_Switch = OPERATION...;
       return 1;
}
...
main_loop() {
       ...
       while (S_Switch) {
              ...
              switch (S_Switch) {
                     case OPERATION1:
                            S_Switch = 0;
                            /( new context switch */
                            /* knowcess is safe here */
                            break;
                     case OPERATION2:
                            S_Switch = 0;
                            /* Delete/restore context */
                            break;
                     case NOOP:
                            break;
                     default:
                            /* Error */
              }
              ...
       }
}
main () {
       /* initialize NXP, set handlers, ... */
       ...
       S_Running = 1;
```

```
        while (S_Running) main_loop();
        ...
        /* exit clean-up */
}
```

To delete a context, use the following program logic:

```
switch to a new context or back to the default initial one
        delete all clientdata attached to the context.
        If you don't you will get memory leaks
NXP_Control(NXP_CTRL_EXIT)
NXP_DeleteContext()
```

Sharing Information Between Contexts

Sessions that you create do not automatically share Rules Element information. If you want to transfer objects or values between contexts, it is your responsibility to either save it into a file of objects / values or in variables which provide what you need.

Note that although you can cache the AtomId in a context before switching, the atom has a separate AtomId per context if multiple contexts use the atom. There is no API function which lets you clone a context. The best alternative is to use the NXP_Journal() function to save the state and then restore the state in the second context as the following code shows:

```
/* clone a context */
NXP_Journal(NXP_JRNL_STATESAVE, (AtomId)0, "state.nxp");
if (!(cntx = NXP_ContextNew())) return 0;
NXP_Control(NXP_CTRL_INIT);
oldCntx = NXP_ContextSetCur(cntx);
NXP_Journal(NXP_JRNL_STATERESTORE, (AtomId)0, "state.nxp");
```

Context API

The existing API will remain unchanged and function as before. In particular, NXP_Control with NXP_CTRL_INIT or NXP_CTRL_EXIT will assume no contexts ... only a single instance. It will, however, use the context mechanism, but refer to a "well-known" context. It is the responsibility of the developer / user of the contexts to maintain (if necessary) a list of active session contexts (though it would be possible to provide a list of allocated contexts).

The NXP contains these context switching API functions:

NXP_ContextNew

Creates a new, empty context.

NxpContextPtr NXP_ContextNew (void);

The size of the context record, to be presented later, is actually very small (<100 bytes, but with pointers to probably very large client context blocks). Creating a new context does not make it the current context. All necessary structures must be initialized.

NXP_ContextDispose

Disposes of a context block.

void NXP_ContextDispose (NxpContextPtr *context***);**

It is the responsibility of any clients to dispose of their portion of the context block (see NXP_ContextSetNfyProc() later). The disposer of the context must destroy all necessary structures.

NXP_ContextGetCur

Returns the current context.

NxpContextPtr NXP_ContextGetCur (void);

A NULL return means there was no previous established context.

NXP_ContextSetCur

Sets the provided context pointer to be the current context.

NxpContextPtr NXP_ContextSetCur (NxpContextPtr *context***);**

The previous context pointer will be returned, since a likely use will be to do something, and then restore the original context. This can be ignored if NXP_ContextGetCur() is used. A NULL return will mean there was no previous established context (i.e., this is the first and/or only one).

NXP_ContextSetNfyProc

Associates a notification procedure with the context mechanism (globally, not per context).

void NXP_ContextSetNfyProc (NxpContextProc *proc***);**

It is possible to have more than one notification procedure installed. The NxpContextProc's will be called (in the reverse order in which they were registered) with the NxpContextPtr, when the context is changed, or as other events become necessary to register. This can be used by a listener to set global variables for faster lookup, update windows, etc., as desired.

NXP_ContextUnsetNfyProc

Removes the specified NxpContextProc from the list to receive notifications.

void NXP_ContextUnsetNfyProc (NxpContextProc *proc***);**

NXP_ContextAllocateClientId

Allocates a client id so that multiple client pointers / data can be associated with the context block.

NxpContextClientId NXP_ContextAllocateClientId (void);

The client/customer should always use this id when accessing the clientdata via one of the next two calls. Note that this id can vary from session to session depending on if/whether others are also storing clientdata and what order things get done in.

NXP_ContextSetClientData

Allows a customer to store information along with the context block, for a given client id.

void NXP_ContextSetClientData (NxpContextPtr *context***, NxpContextClientId** *id***, ClientPtr** *data***);**

NXP_ContextGetClientData

Allows a customer to retrieve their client information from the context block, given the appropriate client id.

ClientPtr NXP_ContextGetClientData (NxpContextPtr *context***, NxpContextClientId** *id***);**

Debugging API

Additional APIs have been provided to check the validity of certain context related items. This is because for the above API, serious errors will usually be signaled with the Open Interface error handling mechanism. Serious errors are (for now): an illegal CntxtClientId, and a bad CntxtPtr. If this is not desired, then the following API should be used, but once an application is debugged, the use of the following API should not be required:

NXP_ContextIsValid

Performs validation of the context block pointer.

BoolEnum NXP_ContextIsValid (NxpContextPtr *context***);**

NXP_ContextIsClientIdValid

Performs validation of the client id.

BoolEnum NXP_ContextIsClientIdValid (NxpContextClientId *id***);**

Examples Description

This Rules Element API primer teaches you how to use the Rules Element "data context switching" API. It starts with a very simple example and progresses to more complex examples.

The context API is designed to allow the user to have several Rules Element sessions going without any overlap of name space or engine operations. This allows applications to safely make use of multiple Rules Element sessions.

The examples are explained in more detail within the C file itself. Each file contains an overview at the front, and the code itself contains numerous explanatory comments. In addition, the examples print out information along the way to illustrate what is happening (the user could also follow this with a debugger, if that is available). There is no graphical interface to these examples, since the context switching is purely non-graphical. On those machines (eg: Macintosh, PC Windows, ...) that do not have a command line interface / console such as is found on most workstations, the print out will appear in an oit.dbg file in the working directory.

The KB's are extremely simple in order to allow the point to be made. The KB's are also designed to have significant "name overlap" ("h" for hypo, etc),

so that the separation of the contexts is more obvious. The examples generally stick with 2 KB's, again for no other reason than keeping the example simple.

Example 1 (cntx1*.*): KB's in series.

This example runs two KBs, first one to completion, then the other. Represented as a "timing diagram" this would be:

```
KB1 ------------X
KB2             -------------X
```

Example 2 (cntx2.*): Using a KB to answer another KB question.

This example again involves two KBs. The first KB asks a question that requires a second KB to be run in order to provide the answer. A question handler traps the request to setup a context for the second KB. After the second KB completes, its answer is provided to the first. Represented as a "timing diagram" this would be:

```
KB1 ------------              ------------X
KB2             -------------X
```

Example 3 (cntx3.*): Crude task switching from KB to KB via POLLING.

This example, using two KBs, installs polling handlers to allow Each KB to run for "a while", before the context is switched to run the next KB for "a while". It also illustrates how to set up "per context" ClientData so the user can use context-specific data. Represented as a "timing diagram" this would be:

```
KB1 --  --  --  --  --  --  --X
KB2   --  --  --  --  --  --  --X
```

Each example contains the following files:

`cntx*.c`	source files for the context examples
`cntx*.tkb`	KB's for the context examples
`makefile`	makefiles to rebuild the examples

A Simple Example cntx1.c

Overview

This example provides an introduction to using data contexts in the Rules Element. This is a simple example using one context to load a knowledge base, get some atom and KB IDs, run the KB to completion, and then do almost exactly the same thing in a following context. The same KB is used in the context, but it is important to note that different atom and KB IDs are returned from the queries. Also, since the questions are answered differently, the "same" hypo has different values in the two contexts. It should also be noted that each context keeps its own set of handlers (eg: the question handler).

Execution flow / logic

1. load / initialize the Rules Element
2. get hypo, set question handler, and knowcess (check IDs and values)
3. create a new data context

4. repeat (1) and (2) for this context
5. destroy this context; restore previous context

verify original context still preserved

destroy original context and exit

cntx1.c listing

```
#define      ERR_LIB      the Rules Element

#include <errpub.h>
#include <strpub.h>
#if (MCH_WIN == MCH_WINMSW)
#      include <apppub.h>
#endif
#include <ndoptpub.h>
#include <nxppub.h>

ERR_DECLARE

#define      ND_RT
#define      ND_NXP
#include <ndopt.c>

/*************************************************************************
       Question handlers
*************************************************************************/

/* This one always answers TRUE */
static       Int    S_AnswerTrue L2(AtomId, atom, Str, str)
{
       Int    status;
       Int    value = NXP_BOOL_TRUE;

       STR_Printf("%s ... answering TRUE\n", str);
       status = NXP_Volunteer(atom, NXP_DESC_INT, (Str)&value,
                     NXP_VSTRAT_QFWRD);
       return 1;
}

/* This one always answers FALSE */
static       Int    S_AnswerFalse L2(AtomId, atom, Str, str)
{
       Int    status;
       Int    value = NXP_BOOL_FALSE;

       STR_Printf("%s ... answering FALSE\n", str);
       status = NXP_Volunteer(atom, NXP_DESC_INT, (Str)&value,
                     NXP_VSTRAT_QFWRD);
       return 1;
}

/*************************************************************************
       Context-based code: creates context, runs a KB, destroys context
*************************************************************************/

static void   S_DoContext L0()
{
       Int          status;
       KBId         kb;
       AtomId       atom;
       Int          boolValue;
       NxpContextPtr cntx;
```

```
        NxpContextPtr prevCntx;
        NxpIProc      fcn;

        /*
         * Create a new the Rules Element context; make it the "current"
         * context (save the previous one); and initialize the new context
         */
        cntx = NXP_NewContext();
        prevCntx = NXP_CurSetContext(cntx);
        status = NXP_Control(NXP_CTRL_INIT);

        /*
         * Load the same KB as before, but answer the question differently.
         * Verify (check printout) that KBIds, AtomIds, and hypo values
         * are indeed different from before.
         */
        status = NXP_LoadKB("cntx1.tkb", &kb);
        status = NXP_GetAtomId("h.Value", &atom, NXP_ATYPE_HYPO);
        status = NXP_SetHandler(NXP_PROC_QUESTION, S_AnswerFalse, 0);
        status = NXP_Suggest( atom, NXP_SPRIO_SUG);
        status = NXP_Control(NXP_CTRL_KNOWCESS);
        status = NXP_GetAtomInfo(atom, NXP_AINFO_VALUE, 0, 0, NXP_DESC_INT,
                    (Str)&boolValue, 0);
        status = NXP_GetHandler(NXP_PROC_QUESTION, &fcn, 0);
        STR_Printf("\n");
        STR_Printf("Context: KBId = %p\n", kb);
        STR_Printf("Context: hypo AtomId = %p\n", atom);
        STR_Printf("Context: hypo value  = %d\n", boolValue);
        STR_Printf("Context: question handler  = %p\n", fcn);
        STR_Printf("Context: S_AnswerFalse     = %p\n", S_AnswerFalse);

        /* Destroy this context */
        status = NXP_Control(NXP_CTRL_EXIT);
        NXP_DisposeContext(cntx);

        /* Restore the previous context */
        NXP_CurSetContext(prevCntx);
}

/*************************************************************************
        Main program
*************************************************************************/

Int     main_L2(Int, argc, Str*, argv)
{
        Int           status;
        KBId          kb;
        AtomId        atom;
        Int           boolValue;
        NxpIProc      fcn;

        ERR_MAININIT;

        APP_InitFirst();
        NDOPT_Install();
        APP_InitLast();

        /*
         * For demo purposes, load a simple KB, install and answer a
         * question, and check various IDs and values (to be contrasted
         * later with what is found in a "data context".
         */
        status = NXP_LoadKB("cntx1.tkb", &kb);
        status = NXP_GetAtomId("h.Value", &atom, NXP_ATYPE_HYPO);
```

```
        status = NXP_SetHandler(NXP_PROC_QUESTION, S_AnswerTrue, 0);
        status = NXP_Suggest(atom, NXP_SPRIO_SUG);
        status = NXP_Control(NXP_CTRL_KNOWCESS);
        status = NXP_GetAtomInfo(atom, NXP_AINFO_VALUE, 0, 0, NXP_DESC_INT,
                                 (Str)&boolValue, 0);
        STR_Printf("Main: KBId = %p\n", kb);
        STR_Printf("Main: hypo AtomId = %p\n", atom);
        STR_Printf("Main: hypo value  = %d\n", boolValue);

        /* Do same calls in a context */
        S_DoContext();

        /*
         * One last quick check to show that the hypo and question handlers
         * in this main context still has the same value as before (different
         * from that in the context).
         */
        status = NXP_GetAtomId("h.Value", &atom, NXP_ATYPE_HYPO);
        status = NXP_GetAtomInfo(atom, NXP_AINFO_VALUE, 0, 0, NXP_DESC_INT,
                                 (Str)&boolValue, 0);
        status = NXP_GetHandler(NXP_PROC_QUESTION, &fcn, 0);
        STR_Printf("\n");
        STR_Printf("Main: hypo AtomId = %p (still)\n", atom);
        STR_Printf("Main: hypo value  = %d (still)\n", boolValue);
        STR_Printf("Main: question handler  = %p (still)\n", fcn);
        STR_Printf("Main: S_AnswerTrue      = %p\n", S_AnswerTrue);

        /* This will destroy the main (current) "context" */
        NDOPT_Exit();

        return EXIT_OK;
}
```

cntx1.ms makefile listing

```
#-------------------------------------------------------------------------
# Makefile
#
# Make: Microsoft nmake
#
# @(#)cntx1.ms5.1 95/04/05
#-------------------------------------------------------------------------

BUILD = DEBUG
SYSTEM = CONSOLE
!include <$(ND_HOME)\include\makedef.inc>

#-------------------------------------------------------------------------
#      Files
#-------------------------------------------------------------------------

OBJS   = cntx1.obj
EXES   = cntx1.exe
RCOS   =

LIBS   = $(LIBS_OS) $(LIBS_CORE) $(LIBS_NX)


!include <$(ND_HOME)\include\makerule.inc>

#-------------------------------------------------------------------------
#      End
#-------------------------------------------------------------------------
```

```
cntx1.tkb listing
(@VERSION=035)
(@COMMENTS="@(#)cntx1.tkb5.1 95/04/05")
(@OBJECT=a    (@PUBLICPROPS=Value@TYPE=Boolean;))
(@OBJECT=h    (@PUBLICPROPS=Value@TYPE=Boolean;))
(@RULE=R_h
      (@LHS= (Yes   (a)))
      (@HYPO=h)
)
(@GLOBALS=@SUGLIST=h;)
```

Using a Question Handler: cntx2.c

Overview

This example continues an introduction to using data contexts in the Rules Element. This example uses two contexts: the original context loads a Knowledge Base and asks a simple question; the second context invokes a KB that provides the answer to the question from the first KB. As with the first example, the KBs are kept extremely simple to not overly complicate the example, and the atom names are deliberately re-used to illustrate the distinct non-overlapping name spaces. In this particular example, the second context is pre-setup to improve the performance (an alternative with poorer performance would have the each invokation of the question handler create the context, load the KB, destroy the context, etc).

Execution flow / logic

1. Set up context1 for main KB; install question handler
2. Set up context2 to answer question
3. Return to context1 and loop over KB in context1
4. In question handler, switch to context2, run and get answer
5. In question handler, restore context1, and volunteer answer

cntx2.c listing

```
#define      ERR_LIB      the Rules Element

#include <errpub.h>
#include <strpub.h>
#if (MCH_WIN == MCH_WINMSW)
#      include <apppub.h>
#endif
#include <ndoptpub.h>
#include <nxppub.h>

ERR_DECLARE

#define      ND_RT
#define      ND_NXP
#include <ndopt.c>

/* statics for easy reference to context #1 */
static       NxpContextPtrS_Cntx1 = NULL;
static       AtomId       S_Hypo1 = NULL;

/* statics for easy reference to context #2 */
static       NxpContextPtrS_Cntx2 = NULL;
```

```
static       AtomId       S_Hypo2 = NULL;

/************************************************************************
       Question handler ... use 2nd KB to randomly answer question from 1st
************************************************************************/

static Int    S_AnswerTrueOrFalse L2(AtomId, atom, Str, str)
{
       Int          status;
       Int          boolValue;
       NxpContextPtr prevCntx;

       STR_Printf("Asking: %s\n", str);

       /*
        * Re-activate 2nd context.  Remember this function returns the
        * current context (1st, in this case) so we can re-install it
        * before we we leave.  The test is just to prove this case (the
        * cast as ULong is for PC-WIN)
        */
       STR_Printf("Enter context2\n");
       prevCntx = NXP_CurSetContext(S_Cntx2);
       if ((ULong)prevCntx != (ULong)S_Cntx1) {
            STR_Printf("context error in question handler\n");
       }

       /*
        * Run the KB in the 2nd context, and use the boolean value
        * of the hypo (randomly TRUE or FALSE) as the answer for
        * the question being asked in the 1st KB.
        */
       status = NXP_Control(NXP_CTRL_RESTART);
       status = NXP_Suggest(S_Hypo2, NXP_SPRIO_SUG);
       status = NXP_Control(NXP_CTRL_KNOWCESS);
       status = NXP_GetAtomInfo(S_Hypo2, NXP_AINFO_VALUE, 0, 0,
                    NXP_DESC_INT, (Str)&boolValue, 0);
       STR_Printf("cntx2_b.tkb, hypo = %d\n", boolValue);

       /* Restore the previous KB */
       (void)NXP_CurSetContext(S_Cntx1);

       /* Volunteer the answer and return */
       STR_Printf("Back to main context, answer: %d\n", boolValue);
       status = NXP_Volunteer(atom, NXP_DESC_INT, (Str)&boolValue,
                    NXP_VSTRAT_QFWRD);

       return 1;
}

/************************************************************************
       Main program
************************************************************************/

Int    main L2(Int, argc, Str*, argv)
{
       Int          status;
       KBId         kb;
       AtomId       atom;
       Int          boolValue;
       Int          i;

       ERR_MAININIT;

       APP_InitFirst();
```

```
        NDOPT_Install();
        APP_InitLast();

        /* Set up 1st context ... established by the above NDOPT_Install */
        S_Cntx1 = NXP_CurGetContext();
        status = NXP_LoadKB("cntx2_a.tkb", &kb);
        status = NXP_GetAtomId("h.Value", &S_Hypo1, NXP_ATYPE_HYPO);
        status = NXP_SetHandler(NXP_PROC_QUESTION, S_AnswerTrueOrFalse, 0);

        /* Set up the 2nd context for use later */
        S_Cntx2 = NXP_NewContext();
        (void)NXP_CurSetContext(S_Cntx2);
        status = NXP_Control(NXP_CTRL_INIT);
        status = NXP_LoadKB("cntx2_b.tkb", &kb);
        status = NXP_GetAtomId("h.Value", &S_Hypo2, NXP_ATYPE_HYPO);

        /* Go back to 1st context and run it "N" (4) times */
        (void)NXP_CurSetContext(S_Cntx1);
        for (i=0; i<4; i++) {
                STR_Printf("\n");
                STR_Printf("Main: pass number = %d\n", i);
                /*
                 * Run the 1st KB and print out the hypo.  The KB's are
                 * set up so that the boolean values change randomly,
                 * but should always be opposite values.
                 */
                status = NXP_Control(NXP_CTRL_RESTART);
                status = NXP_Suggest(S_Hypo1, NXP_SPRIO_SUG);
                status = NXP_Control(NXP_CTRL_KNOWCESS);
                status = NXP_GetAtomInfo(S_Hypo1, NXP_AINFO_VALUE, 0, 0,
                                NXP_DESC_INT, (Str)&boolValue, 0);
                STR_Printf("cntx2_a.tkb, hypo = %d\n", boolValue);

                /*
                 * One last print out to verify that the h.Value from
                 * the 2nd context hasn't changed and is still distinct
                 * from the h.Value of the 1st context.
                 * Note that since we have not destroyed the 2nd context and
                 * know the atom ID, we can still do queries about the other
                 * context, even though it is not active.  Static queries
                 * of this nature are possible, as long as you are careful
                 * about the existence of the other context!
                 */
                status = NXP_GetAtomInfo(S_Hypo2, NXP_AINFO_VALUE, 0, 0,
                                NXP_DESC_INT, (Str)&boolValue, 0);
                STR_Printf("cntx2_b.tkb, hypo = %d (still)\n", boolValue);
        }

        /* Restore 2nd context to destroy it ... */
        (void)NXP_CurSetContext(S_Cntx2);
        status = NXP_Control(NXP_CTRL_EXIT);
        NXP_DisposeContext(S_Cntx2);

        /* Restore 1st context to destroy it ... */
        (void)NXP_CurSetContext(S_Cntx1);
        NDOPT_Exit();

        return EXIT_OK;
}
```

cntx2.ms makefile listing

```
#-------------------------------------------------------------------------
# Makefile
#
# Make: Microsoft nmake
#
# @(#)cntx2.ms5.1 95/04/05
#-------------------------------------------------------------------------

BUILD = DEBUG
SYSTEM = CONSOLE
!include <$(ND_HOME)\include\makedef.inc>

#-------------------------------------------------------------------------
#      Files
#-------------------------------------------------------------------------

OBJS   = cntx2.obj
EXES   = cntx2.exe
RCOS   =

LIBS   = $(LIBS_OS) $(LIBS_CORE) $(LIBS_NX)


!include <$(ND_HOME)\include\makerule.inc>

#-------------------------------------------------------------------------
#      End
#-------------------------------------------------------------------------
```

cntx2_a.tbk listing

```
(@VERSION=035)
(@COMMENTS="@(#)cntx2_a.tkb5.1 95/04/05")
(@OBJECT=a    (@PUBLICPROPS=Value@TYPE=Boolean;))
(@OBJECT=h    (@PUBLICPROPS=Value@TYPE=Boolean;))
(@RULE=R_h
       (@LHS= (No    (a)))
       (@HYPO=h)
)
(@GLOBALS=@SUGLIST=h;)
```

cntx2_b.tbk listing

```
(@VERSION=035)
(@COMMENTS="@(#)cntx2_b.tkb5.1 95/04/05")
(@OBJECT=a    (@PUBLICPROPS=Value@TYPE=Float;))
(@OBJECT=h    (@PUBLICPROPS=Value@TYPE=Boolean;))
(@RULE=R_h
       (@LHS= (Assign(RANDOM())(a))
       (<     (a)    (RANDOMMAX()/2))
       )
       (@HYPO=h)
)
(@GLOBALS=@SUGLIST=h;)
```

A Polling Example: cntx3.c

Overview

This example concludes the introduction to using data contexts in the Rules Element. This example uses two contexts, though the code is able to be easily extended to more. Each context is pre-loaded with a KB (in this case the same KB, but that is also not necessary), with their appropriate starting hypos suggested. Polling and EndOfSession handlers are setup for each context (the same handlers for each). These contexts are individually registered in an Array as needed to be started. As these contexts are started, they are moved into an active task Array. Every "so many" ticks of the polling handler we stop the current the Rules Element session and return from the knowcess to switch to another context. At the very end, we remove the task from the active list and put it on the idle Array for destruction at the end of the program.

This example also shows how to make use of context-specific clientdata (for information you may wish to associate on a "per-context" basis.

Execution flow / logic

1. Create tasks: suggest hypo, install POLLING, ENDOFSESSION handlers, associate context clientdata, etc.
2. Loop over tasks needing running (start-new / continue-existing)
3. In Polling handler, StopSession (main loop moves to "next task")
4. In EndOfSession handler, task is moved from Active to Idle queue.
5. At "wind down", display information (different per context) and destroy contexts

cntx3.c listing

```
#define      ERR_LIB      the Rules Element

#include <errpub.h>
#include <strpub.h>
#include <arraypub.h>
#if (MCH_WIN == MCH_WINMSW)
#      include <apppub.h>
#endif
#include <ndoptpub.h>
#include <nxppub.h>

ERR_DECLARE

#define      ND_RT
#define      ND_NXP
#include <ndopt.c>

static       ArrayPtr     S_StartTasks= NULL;
static       ArrayPtr     S_ActiveTasks= NULL;
static       ArrayPtr     S_IdleTasks= NULL;

static       NxpContextClientId  S_ClientId= 0;
```

```
/************************************************************************
       Polling handler
************************************************************************/

#defineS_SWITCHCOUNT1

static       Int    S_SwitchTask L0()
{
static       Int    count = 0;
             Int    status;

       /*
        * Check against a static counter in this Polling handler
        * (incremented each time through the loop).  Every one out of
        * (S_SWITCHCOUNT + 1) times, trigger a STOPSESSION.  This will
        * pause the current the Rules Element session and return control to
        * the main loop (which will activate the next context and do a
        * CONTINUE).
        */
       if (count++ >= S_SWITCHCOUNT) {
              status = NXP_Control(NXP_CTRL_STOPSESSION);
              count = 0;
       }

       return 1;
}

/************************************************************************
       End-of-Session handler
************************************************************************/

static Int    S_EndTask L0()
{
       Int           status;
       Int           index;
       Int           boolValue;
       AtomId        hypo;
       AtomId        slot;
       Double        slotValue;
       NxpContextPtr cntx;

       /*
        * EndOfSession encountered ... move this task (context) from
        * the Active list to the Idle list for later disposal.
        */
       cntx = NXP_CurGetContext();

       index = ARRAY_LookupElt(S_ActiveTasks, (ClientPtr)cntx);
       if (index == -1) {
              STR_Printf("context error in EndOfSession handler\n");
       }
       ARRAY_SetElt(S_ActiveTasks, index, (ClientPtr)NULL);
       ARRAY_AddElt(S_IdleTasks, (ClientPtr)cntx);

       return 1;
}

/************************************************************************
       Print out task summary and dispose of it
       (This assumes known names for hypos, atoms of interest)
************************************************************************/

static       void   S_DisposeTask L1(NxpContextPtr, cntx)
{
```

```
        Int             status;
        Int             boolValue;
        AtomId          hypo;
        AtomId          slot;
        Double          slotValue;
        Str             taskName;

        if (cntx == NULL) {
                return;
        }

        (void)NXP_CurSetContext(cntx);

        taskName = (Str)NXP_GetContextClientData(cntx, S_ClientId);
        status = NXP_GetAtomId("h.Value", &hypo, NXP_ATYPE_HYPO);
        status = NXP_GetAtomInfo(hypo, NXP_AINFO_VALUE, 0, 0, NXP_DESC_INT,
                    (Str)&boolValue, 0);
        status = NXP_GetAtomId("a.Value", &slot, NXP_ATYPE_SLOT);
        status = NXP_GetAtomInfo(slot, NXP_AINFO_VALUE, 0, 0,
                                NXP_DESC_DOUBLE, (Str)&slotValue, 0);

        STR_Printf("\n");
        STR_Printf("task = %s\n", taskName);
        STR_Printf("cntx = %p\n", cntx);
        STR_Printf("hypo h.Value = %d\n", boolValue);
        STR_Printf("slot a.Value = %.3f\n", slotValue);

        PTR_Dispose(taskName);
        status = NXP_Control(NXP_CTRL_EXIT);
        NXP_DisposeContext(cntx);
}

/*************************************************************************
        Resume a task
*************************************************************************/

static        BoolEnum      S_ResumeTask L1(NxpContextPtr, cntx)
{
        Int             status;

        if (cntx == NULL) {
                return BOOL_FALSE;
        }

        /*
         * Make this context current, and resume.  (In this simple
         * implementation, completed tasks are NULLed out to avoid
         * perturbing this loop)
         */
        (void)NXP_CurSetContext(cntx);
        STR_Printf(" %s", (Str)NXP_GetContextClientData(cntx, S_ClientId));
        status = NXP_Control(NXP_CTRL_CONTINUE);

        return BOOL_TRUE;
}

/*************************************************************************
        Start a task on the list
*************************************************************************/

static        BoolEnum      S_StartTask L2(NxpContextPtr, cntx, Int, index)
{
        Int             status;
```

```
	if (cntx == NULL) {
		return BOOL_FALSE;
	}

	/*
	 * Make this context current, move context from Start to Active
	 * array and start it.  (Elements are removed from the back to
	 * not affect the countdown loop)
	 */
	(void)NXP_CurSetContext(cntx);
	ARRAY_RemoveIndex(S_StartTasks, index);
	ARRAY_AddElt(S_ActiveTasks, cntx);
	STR_Printf(" %s", (Str)NXP_GetContextClientData(cntx, S_ClientId));
	status = NXP_Control(NXP_CTRL_KNOWCESS);

	return BOOL_TRUE;
}

/*************************************************************************
	Create a new task and add it to the list
*************************************************************************/

static	void	S_NewTask L4(NxpContextPtr,cntx,
				Str,		taskName,
				Str,		kbName,
				Str,		hypoName)
{
	KBId		kb;
	AtomId		hypo;
	Int		status;
	Str		str;

	STR_Printf("add task: name = %s, context = %p, kb = %s, hypo = %s\n",
		taskName, cntx, kbName, hypoName);

	/*
	 * Load KB passed in, suggest hypo (for subsequent KNOWCESS).
	 * Install POLLING and ENDOFSESSION handlers to "task switch"
	 * and "terminate task" respectively.
	 */
	status = NXP_LoadKB(kbName, &kb);
	status = NXP_GetAtomId(hypoName, &hypo, NXP_ATYPE_HYPO);
	status = NXP_SetHandler(NXP_PROC_POLLING, S_SwitchTask, 0);
	status = NXP_SetHandler(NXP_PROC_ENDOFSESSION, S_EndTask, 0);
	status = NXP_Suggest(hypo, NXP_SPRIO_SUG);

	/* Store our name, using ClientData field */
	str = PTR_New(STR_Len(taskName)+1);
	STR_Cpy(str, taskName);
	NXP_SetContextClientData(cntx, S_ClientId, (ClientPtr)str);

	ARRAY_AddElt(S_StartTasks, (ClientPtr)cntx);
}

/*************************************************************************
	Main program
*************************************************************************/

Int	main L2(Int, argc, Str*, argv)
{
	Int		status;
	AtomId		atom;
	NxpContextPtr	cntx;
	Int		index;
```

```
BoolEnum     something_active = BOOL_TRUE;

ERR_MAININIT;

APP_InitFirst();
NDOPT_Install();
APP_InitLast();

/* Initialize task arrays */
S_StartTasks = ARRAY_New();
S_ActiveTasks = ARRAY_New();
S_IdleTasks = ARRAY_New();

/* Get a ClientId for using ClientData */
S_ClientId = NXP_AllocateContextClientId();

/*
 * Setup 1st context ... KB is arbitrary.
 * Pre-suggest hypo.
 * Install POLLING and ENDOFSESSION handlers.
 * Add context to start array.
 */
cntx = NXP_CurGetContext();
S_NewTask(cntx, "a", "cntx3.tkb", "h.Value");

/* Setup 2nd context ... KB is arbitrary, and do as before */
cntx = NXP_NewContext();
(void)NXP_CurSetContext(cntx);
status = NXP_Control(NXP_CTRL_INIT);
S_NewTask(cntx, "b", "cntx3.tkb", "h.Value");

/* Loop as long as something is still runnable ... */
while (something_active) {
       something_active = BOOL_FALSE;

       /* Look for anything needing starting ... */
       index = ARRAY_GetLen(S_StartTasks) - 1;
       if (index >= 0) STR_Printf("\nStart");
       for ( ; index >= 0 ; index--) {
              cntx = ARRAY_GetElt(S_StartTasks, index);
              if (S_StartTask(cntx, index) == BOOL_TRUE) {
                     something_active = BOOL_TRUE;
              }
       }

       /* Look for anything in a runnable state ... */
       index = ARRAY_GetLen(S_ActiveTasks) - 1;
       if (index >= 0) STR_Printf("\nRun");
       for ( ; index >= 0 ; index--) {
              cntx = ARRAY_GetElt(S_ActiveTasks, index);
              if (S_ResumeTask(cntx) == BOOL_TRUE) {
                     something_active = BOOL_TRUE;
              }
       }
}

/*
 * Nothing left running ... destroy all used contexts
 */
STR_Printf("\n");
index = ARRAY_GetLen(S_IdleTasks) - 1;
for ( ; index >= 0 ; index--) {
       cntx = ARRAY_GetElt(S_IdleTasks, index);
       S_DisposeTask(cntx);
```

```
        }

        /*
         * Note that we DO NOT CALL NDOPT_Exit() !
         * We manually destroyed all contexts in the destruct loop above.
         */

        return EXIT_OK;
}
```

cntx3.ms makefile listing

```
#-------------------------------------------------------------------------
# Makefile
#
# Make: Microsoft nmake
#
# @(#)cntx3.ms5.1 95/04/05
#-------------------------------------------------------------------------

BUILD = DEBUG
SYSTEM = CONSOLE
!include <$(ND_HOME)\include\makedef.inc>

#-------------------------------------------------------------------------
#      Files
#-------------------------------------------------------------------------

OBJS   = cntx3.obj
EXES   = cntx3.exe
RCOS   =

LIBS   = $(LIBS_OS) $(LIBS_CORE) $(LIBS_NX)


!include <$(ND_HOME)\include\makerule.inc>

#---------------------------------------------
```

Appendix

A Retrieving Rules Element Information

This appendix addresses the issues involved in the retrieving of information from the Rules Element via the API. Because a lot of programming is done "by example", working examples of actual code will be provided.

Following the examples, there will also be a discussion of things to watch out for that may cause problems if you find your calls are not working. Throughout this note, a very terse atom naming convention will be used. Names like "c", "c1", etc. will refer to classes; "o", "o1", etc. will refer to objects; "p", "p1", etc. will refer to properties. For example: "o.p1" would be the property "p1" of object "o".

C Language

1. Introduction / Common Errors

The general syntax of the call is:

```
int  NXP_GetAtomInfo( theAtom, code, optAtom, optInt,
                      desc, thePtr, len)
```

with the arguments described by:

```
AtomId       theAtom;
Int          code;
AtomId       optAtom;
int          optInt;
int          desc;
Str          thePtr;
int          len;
```

Remember that the routine NXP_GetAtomInfo must always have 7 arguments! It is possible (and even recommended) to use C macros for shortcuts and to use the function prototypes defined in `nxppub.h`. If you are using NXP_GetAtomInfo directly and are having problems, the first thing you should do is count the number of arguments you have provided in the call. Any arguments documented as being "optional" must still be present, typically as a "0" or "NULL".

On 32-bit machines, people are sometimes careless and use `int` when `AtomId` is requested. This will work on 32-bit machines like Alpha OpenVMS and most Unix machines. This will NOT work in a PC Windows 16environment where `int` means a 16-bit quantity. If the C compiler on your platform supports C function prototyping, you might wish to consider using the portable data type of the Elements Environment such as Int32 for 32-bit integers or Int16 for 16 bit integer.

The 6th argument is used to return information to the caller. It must be a writable, allocated storage area. The calling sequence follows the "C-standard" and is documented by how it is seen by NXP_GetAtomInfo: for example, a pointer to a `char` (since these are generic pointers).

However, some people have mistaken this for the argument to be passed and do

```
IntPtr       myIntPtr;
NXP_GetAtomInfo( ...  myIntPtr ...);
```

This is wrong! The variable `myIntPtr` does not point to a known storage space, and the results from NXP_GetAtomInfo will be written to a random location. The correct method is:

```
int          myInt;
NXP_GetAtomInfo( ... (Str) &myInt ... );
```

A pointer to the storage location `myInt` is being passed, and that is where the results will end up. This statement applies for `char`s, `int`s, `float`s, etc. With C strings, it is allowable to use "`myString`" instead of "`&myString[0]`". This note will use the "&..[0]" mode to keep similarity with the non-`char` accesses and to emphasize passing the "address-of" a known location.

NXP_GetAtomInfo returns a value of "1" (type `int`) if it succeeds, or "0" if it fails. It is not correct to check the return of NXP_GetAtomInfo against the NXP_ERR_xxx codes! Only if the call to NXP_GetAtomInfo fails should the user call NXP_Error() to determine the reason for the failure (now specified by the NXP_ERR_xxx codes). That is, the correct use is:

```
if ( !(NXP_GetAtomInfo( ... ) ) {
      reason = NXP_Error();
      ... etc ...
}
```

Neglecting the typecast indicated may cause the NXP_GetAtomInfo call to fail.

2. Declarations

Under C, your variable declarations will typically look like:

```
#include     <nxppub.h>
#define      STRSIZE      255
AtomId       myAtom;
int          myBool;
int          myInt;
long         myLong;
float        myFloat;
double       myDouble;
Char         myString[STRSIZE];
int          myStringLen;
int          status;
```

where the "`my`" variables are those your program wishes to retrieve. `STRSIZE` is a C "define" that has been used to set the length of the C-string.

3. Retrieving an Integer Property Value

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                          0, NXP_DESC_INT, (Str) &myInt, 0 );
```

Alternatively, using the pre-defined macro (which does the typecasts):

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GETINTVAL( myAtom,  &myInt );
```

On many platforms, there is no difference between `long`s and `int`s. On some platforms, like Windows 3.1.x for example, there is a difference

between an int and a long, and you should be sure to use the correct method to retrieve into the proper datatype. It is possible to explicitly retrieve longs instead of ints with NXP_GetAtomInfo. (As an aside, the Rules Element stores all integer slots internally as longs.)

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                          0, NXP_DESC_LONG, (Str) &myLong, 0
);
```

There is no macro version for NXP_DESC_LONG in nxppub.h but you can easily add one yourself.

4. Retrieving a String Property Value

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                    0, NXP_DESC_STR, &myString[0], STRSIZE );
```

Alternatively, using the pre-defined macro (which does the typecasts):

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GETSTRVAL( myAtom,  &myString[0], STRSIZE );
```

5. Retrieving a Boolean Property Value

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                          0, NXP_DESC_INT, (Str) &myBool, 0 );
```

Alternatively, using the pre-defined macro (which does the typecasts):

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GETINTVAL( myAtom,  &myBool );
```

You will note that this is essentially the same as retrieving an integer. Since the Rules Element booleans can have 4 different values, you must check for TRUE, FALSE, NXP_BOOL_UNKNOWN, and NXP_BOOL_NOTKNOWN.

6. Retrieving a Real/Floating Property Value

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                    0, NXP_DESC_DOUBLE, (Str) &myDouble, 0 );
```

Alternatively, using the pre-defined macro (which does the typecasts):

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GETDOUBLEVAL( myAtom,  &myDouble );
```

Note that this is a "double" (e.g. 8 bytes of floating point information). It is also possible to use floats instead of doubles with NXP_GetAtomInfo. On most platforms, these are different, and you should be sure to use the correct method to retrieve into the proper datatype. As an aside, the Rules Element stores all floating point numbers internally as doubles.

```
status = NXP_GetAtomId( "o.p", &myAtom, NXP_ATYPE_SLOT );
status = NXP_GetAtomInfo( myAtom, NXP_AINFO_VALUE, (AtomId) 0,
                          0, NXP_DESC_FLOAT, (Str) &myFloat,
0 );
```

There is no macro version for NXP_DESC_FLOAT in nxpdef.h but you can easily add one yourself.

Index

Symbols

A

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

U

W

PostScript error (--nostringval--, --nostringval--)