NodeBrain Library

Release 0.8.17

NodeBrain Library
August 2014
NodeBrain Open Source Project

History

2014-02-16 Title: *NodeBrain Library*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

Release 0.8.17

- This document replaces *NodeBrain API Reference* first drafted in 2003 for release 0.6.3
- Converted to texinfo format
- Dropped several function to be added back later after more complete testing.

Preface

This document is for C programmers writing extensions to NodeBrain. It describes the C API provided by the NodeBrain Library. In most cases you can use the API to provide new functionality in the form of a module, a plugin to the rule engine. You also have the option of writing a skull (main program) as a replacement for the `nb` program included in the distribution.

We assume the reader of this document is familiar with the content of other documents listed below. If you are looking for information on distributed modules, they each have a separate manual you can access via the project website.

See www.nodebrain.org for more information and the latest update to this document.

Documents

NodeBrain Guide - Information on using `nb`
NodeBrain Tutorial - A gentle introduction to `nb` and the rule language
NodeBrain Language - Rule language syntax and semantics
NodeBrain Library - C API

Caboodle NodeBrain Kit - A framework for managing rules
System NodeBrain Kit - A small sample application

Document Conventions

Sample code and input/output examples are displayed in a monospace font, indented in HTML and Info, and enclosed in a box in PDF or printed copies. Bold text is used to bring the reader's attention to specific portions of an example. In the following example, the first and last line are associated with the host shell and the lines in between are input or output unique to NodeBrain. The `define` command is highlighted, indicating it is the focus of the example. Lines ending with a backslash \ indicate when a command is continued on the next displayed line. This is supported by the language within source files, but not for other methods of command input. If you copy an example of a command displayed over multiple lines, you must enter it as a single line when used outside the context of a source file.

```
$ nb
> define myFirstRule on(a=1 and b=2) mood="happy";
> assert mood="sad";
> show mood
mood = "sad"
> assert a=1,b=2,c=3,d="This is an example of a long single line that",\
    e="we depict on multiple lines to fit on the documnet page";
2008/06/05 12:09:08 NB000I Rule myFirstRule fired(mood="happy")
> show mood
mood = "happy"
> quit
$
```

Table of Contents

1 Concepts **1**

1.1 Interface Structure 1
1.2 Command Language 1
1.2.1 Interactive Session 2
1.2.2 Source Files 2
1.2.3 Piping Commands to NodeBrain 2
1.2.4 Piping Action Messages from NodeBrain 2
1.3 Servant Interface 3
1.3.1 Source and Action Servants 3
1.3.2 Servant Node Module 3
1.3.3 Using NodeBrain as a Servant 4
1.4 Interface Modules 4
1.5 Library Functions 4
1.5.1 Cerebrum 5
1.5.2 Medulla 6
1.5.3 Spine 6
1.6 Selecting an Interface 6
1.7 Methods 7

2 Skull Functions **9**

2.1 nbStart - Start rule engine 11
2.2 nbServe - Serve engine arguments 11
2.3 nbStop - Stop rule engine 11

3 Module Functions **13**

3.1 nbBind Method - Initialize module 14
3.2 nbSkillDeclare - Register a skill 15
3.3 nbSkillFacet - Register a skill facet 15
3.4 nbSkillMethod - Register a skill facet method 16
3.5 nbSkillSetMethod - Bind a skill method 17
3.6 nbVerbDeclare - Register a command verb 18
3.7 Module Command Method - Handle a registered command verb 18

4 Skill Methods **19**

4.1 Alarm - Handle node alarm 20
4.2 Alert - Handle node sentence alert 20
4.3 Assert - Handle node sentence assertion 21
4.4 Bind - Create a skill handle and bind methods 21
4.5 Command - Handle node command 22
4.6 Compute - Handle node sentence computation 22

4.7 Construct - Handle node definition 23
4.8 Enable - Handle node enable command 24
4.9 Evaluate - Handle node sentence evaluation 24
4.10 Destroy - Handle node destruction 24
4.11 Disable - Handle node disable command 25
4.12 Show - Handle request to display node 25
4.13 Solve - Resolve unknowns and compute node sentence 26

5 Node Functions 27

5.1 nbCmd - Interpret command 27
5.2 nbListenerEnableOnDaemon - Schedule enable when daemonized 27
5.3 nbLogMsg - Write a message to the log 28
5.4 nbLogPut - Write text to the log 28

6 Cell Functions 31

6.1 nbCellCompute - Compute value of disabled cell 31
6.2 nbCellCreate - Create cell from expression 31
6.3 nbCellCreateReal - Create number cell 32
6.4 nbCellCreateString - Create string cell 32
6.5 nbCellDrop - Release cell 32
6.6 nbCellEvaluate - Compute enabled cell value after change 33
6.7 nbCellGetReal - Get number from number cell 33
6.8 nbCellGetString - Get string from string cell 34
6.9 nbCellGetType - Get cell type code 34
6.10 nbCellGetValue - Get enabled cell value 34
6.11 nbCellGrab - Reserve cell 35
6.12 nbCellSolve - Solve for value of cell 35
6.13 nbListOpen - Open a cell list to iterate 36
6.14 nbListGetCell - Get next cell in list 36
6.15 nbListGetCellValue - Get value of next cell in list 37
6.16 nbSynapseOpen - Register an Alarm method 37
6.17 nbSynapseSetTimer - Set an alarm 37
6.18 nbTermCreate - Create a new term 37
6.19 nbTermLocate - Locate an existing term 37
6.20 nbTermSetDefinition - Assign new definition to term 38

7 Medulla Functions 39

8 Spine Functions 41

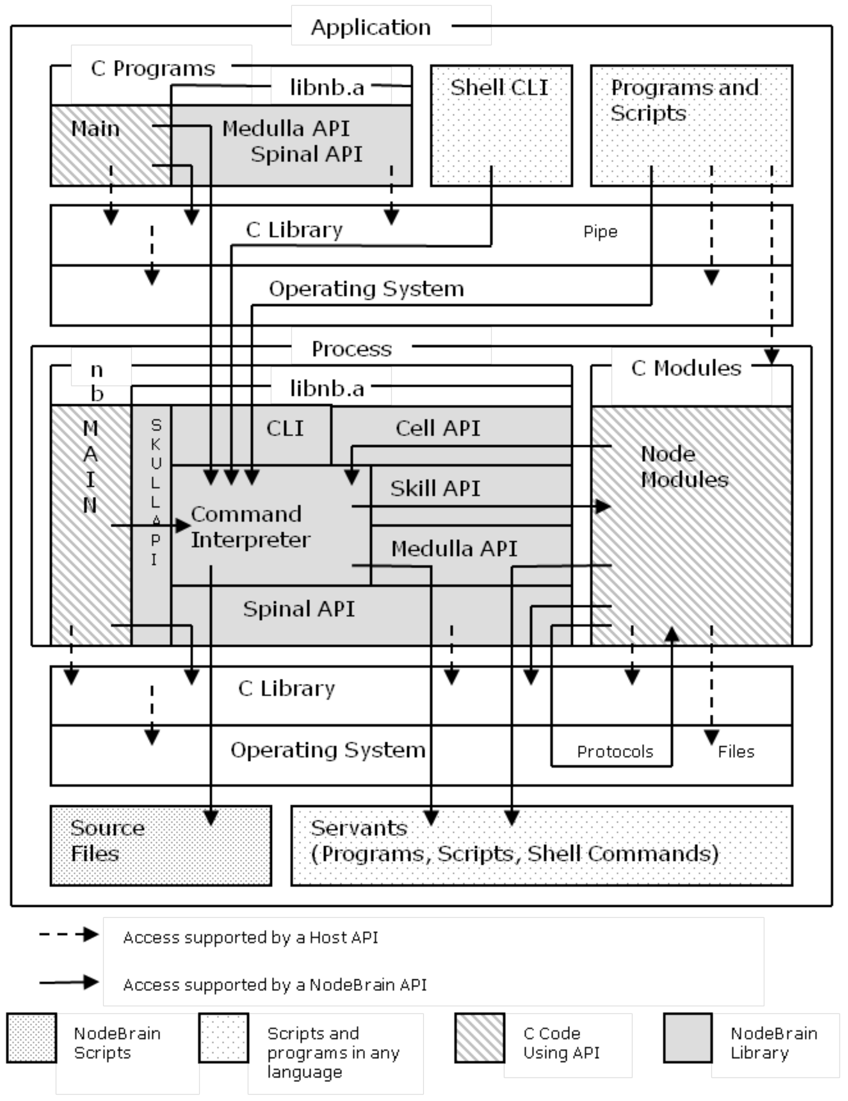
Index 43

1 Concepts

This chapter provides an outline of NodeBrain interfaces to help you choose the method most appropriate for you application. You may find that interfaces other than the C API provided by the NodeBrain Library are sufficient for your application.

1.1 Interface Structure

The figure below illustrates the available interface options and the relationship between NodeBrain and the components you may elect to write or use.



1.2 Command Language

The primary interface to NodeBrain is the command language. Use the command language to specify what you want NodeBrain to do. Familiarity with the language is required for using the interfaces described in this section. See the *NodeBrain Language* manual for more information.

1.2.1 Interactive Session

An interactive session may be launched by invoking the nb program from a shell command prompt as shown below.

```
$ nb
> command
```

NodeBrain commands are entered in response to the command prompt `">"`.

1.2.2 Source Files

The easiest way to provide commands to the interpreter is by directing it to process source files.

```
$ nb file
```

The `source` command may also be used to direct the interpreter to process a source file, provided you already have an interface to the interpreter. In interactive mode, the `source` command is entered at the prompt.

```
> source file
```

1.2.3 Piping Commands to NodeBrain

Commands may be piped to NodeBrain's `stdin` using a shell command as follows.

```
$ script | nb =
```

Pipes can also be used within a script as illustrated by the following Perl script.

```
#!/usr/bin/perl
open(NB,"| nb =")||die;
print(NB "command1");
print(NB "command2");
close(NB);
```

1.2.4 Piping Action Messages from NodeBrain

Output directed to NodeBrain's `stdout` using the ^ directive may be sent to another script to handle required actions.

```
$ nb source | script
```

This may also be done by invoking NodeBrain from within a script as illustrated by this Perl script.

```
#!/usr/bin/perl
open(NB,"nb source |")||die;
while(<NB>){
  ... take action ...
  }
close(NB);
```

1.3 Servant Interface

The servant interface is designed to enable programs and scripts written in any language to communicate easily with the NodeBrain interpreter by reading from `stdin` and writing to `stdout` and `stderr`. This is a convenient interface that you should consider as an alternative to the C API.

1.3.1 Source and Action Servants

A source servant is any command or script that writes NodeBrain commands to `stdout` to be interpreted by the invoking NodeBrain process. This is similar to CGI programs or scripts that write HTML on `stdout` to a web server.

```
> -:  command  Interpret stdout, log stderr and send to attached client
> -|: command  Interpret stdout, log stderr only
> =:  command  Interpret stdout, send stderr to generated output file
> =|: command  Interpret stdout, log stderr
```

An action servant is any program or script invoked by NodeBrain to perform an action—often a rule action.

```
> -  command    Output to log file and attached client
> -| command    Output to log file only
> =  command    Output to generated output file
> =| command    Output to log file
```

See the *NodeBrain Language* manual for a complete description of the servant commands, `-` and `=`.

1.3.2 Servant Node Module

The Servant module provides a way to define servants for repeated use, and enables the use of pipes for both `stdin` and `stdout`. This enables a program or script to persistently function as a source servant and/or action servant.

```
define term node servant:[|] servantCommand
```

This is a convenient way to add functionality to a rule set using languages best suited for implementing the needed functionality. Although a servant runs as a separate process, it is a logical extension of the parent NodeBrain process.

This interface also enables communication between servants. One servant may write a NodeBrain command to `stdout`, which when interpreted by the parent NodeBrain causes a message to be sent to `stdin` of another servant.

The servant module is described in the *Servant NodeBrain Module* manual.

1.3.3 Using NodeBrain as a Servant

You can use one NodeBrain script as a servant to another NodeBrain script since NodeBrain supports the trivial servant interface as both a parent and a child process. In such a case, the text messages exchanged in both directions must be valid NodeBrain commands. This is a way to split a NodeBrain agent into multiple processes, taking advantage of multiple CPU's and concurrent processing.

1.4 Interface Modules

Some of the modules distributed with NodeBrain support various ways of communicating with the rule engine. The Mail, Pipe, Snmptrap, Syslog, and Webster modules support communication via standard protocols. These can be used to communicate with non-NodeBrain components. For example, the Webster module enables you to send commands to a NodeBrain agent using a web browser.

In addition, the Peer and Message modules enable communication between NodeBrain agents using NodeBrain specific protocols. It is possible for servants written in your favorite language, running under two different NodeBrain agents, to communicate with one another via messages sent between agents using a module specific protocol without the servants having any knowledge of the protocol. For example, a servant script might write the following command to `stdout`.

```
foo:bar:hello
```

If `foo` is defined as a node within the parent agent that securely forwards commands to a peer agent, and `bar` is defined as a servant in the peer, then `hello` will be sent to the peer's servant on `stdin`. If the peer-to-peer nodes were defined as `foo` on both ends, and the servants were both defined as `bar`, then the peer servant could return a message with the same `foo:bar:` prefix. The Message module supports broadcasting messages to all agents within a cabal with assured one-time delivery.

1.5 Library Functions

The NodeBrain Library provides the C API, the topic of this manual. The C API is the most difficult method of interfacing with NodeBrain. Although this is the least efficient approach in terms of your time spent implementint a solution, it can sometimes provide the most efficient solution. So while most of this chapter is intended to discourage the use of the C API when unnecessary, you are encouraged to use it when you feel it provides the best option for your requirements.

Each function is part of one of three major logical components: Cerebrum, Medulla, and Spine. Within each, functions are further divided into logical sets. We refer to each individual set as an API, each larger component set as an API, and the full set provided by the library as the NodeBrain C API.

Component	**Set**	**Interface**
Cerebrum		Interface to interpreter
	Skull	Host programs interface to an embedded rule engine
	Module	Module registration of callback methods
	Node	Interface to nodes
	Cell	Interface to cells
	Skill	Rule engine calls back to modules
Medulla		Process and asynchronous IO management (breathing and heart rate)
	File	Functions to add/replace/remove file handlers to respond to files that become ready for read or write operations.
	Queue	Functions to add/replace/remove file handlers to respond to files that become ready for read or write operations.
	Process	Functions that spawn and terminate child processes and provide queued asynchronous communication between the parent and child processes. This function group makes use of the File and Queue functions, as well as functions in the Spine.
	Wait	This function group is based on the select() function on Unix/Linux and WaitForMultipleObjects() on Windows.
Spine		Interface to the host operating system
	Servant	Child process interface layered on top of the process function group. This interface is more restricted.
	Process	Child process interface
	Network	Network communications

1.5.1 Cerebrum

The Cerebrum is a subset of functions that provide the internal services of the interpreter. These functions are intended only for C programmers wanting to extend the functionality of NodeBrain by writing skulls or modules. A skull is a program that embeds the rule engine and must conform to the structure imposed by the Medulla. The nb program is itself a skull. A module is a plugin to the rule engine, providing additional capabilities.

1.5.2 Medulla

The Medulla is a subset of functions appropriate for a server application. These functions are intended for use by NodeBrain, but may be used by other C programs. Simply include the Medulla header (nb/nbmedulla.h) and link with the NodeBrain Library (-lnb). These functions impose a structure on the calling program that may not be easy to incorporate into an existing program. When the Medulla is operating in server mode it may conflict with a program's use of timers and IO not designed for use with the Medulla.

1.5.3 Spine

The Spine is a subset of functions that provide an interface to the host operating system, without dependence on a NodeBrain environment. These functions are primarily intended for use by other NodeBrain functions, but may be used by any C program. Simply include the spine header (nb/nbspine.h) and link with the NodeBrain Library (-lnb).

1.6 Selecting an Interface

It seems wise to start with simple interfaces and move to more complex interfaces only as necessary. If you follow this advice you will consider the interfaces in the order described here.

Start by writing rules and using existing components.

Frequently a NodeBrain application can be accomplished using existing programs and modules. All you need to do in many cases is write your NodeBrain rules and save them as source files. You can use most existing utilities and shell commands as action servants.

Write servants in your favorite language.

If you need to write servants, you should start by writing simple servants in your favorite programming language—shell scripts, other scripting languages (e.g. Perl or Python), or compiled languages (e.g. C/C++). Servants may be parent or child processes in relation to NodeBrain, and need only conform to NodeBrain's assumptions about `stdin`, `stdout`, `stderr`. Servants are often very simple programs. Even when you write a complex servant, you are working in a language familiar to you and the interface to NodeBrain never gets more complicated than reading from `stdin` and/or writing to `stdout`.

Write a module to extend NodeBrain.

If you are a C programmer, and want an integrated extension to NodeBrain, write a module. This enables you to enhance NodeBrain without modifying the interpreter. This is significantly more complicated than writing servants, and there is a greater (although hopefully not great) risk of encountering compatibility issues with future releases of NodeBrain. A module provides method functions called by the interpreter to handle assertions, cell expressions, and commands. This enables a module to extend the language in predefined ways.

Write a skull program to extend NodeBrain.

There may be cases where you want the NodeBrain interpreter embedded within your program. This is generally not recommended because it requires recompiling your program to upgrade to a new release of NodeBrain. However, if you think an integrated program is the right approach for your application, this option is available.

Modify NodeBrain.

If NodeBrain just doesn't work for you as released, but you see a way to modify it to make it work, this is a fine approach also. The downside of this approach is the burden of maintaining your own copy. However, if your change could benefit others, you may be able to escape the burden of maintenance by submitting your change as a proposal to the NodeBrain project.

1.7 Methods

In this document, a *method* is a function you write for NodeBrain to call. Most methods are passed as a callback function in a call to a NodeBrain API function. A couple are symbols in a dynamic module that follow a naming convension enabling NodeBrain to resolve them.

2 Skull Functions

This set of functions is used to embed the rule engine in a C program, a *skull* from the rule engine's perspective. This chapter covers only functions designed specifically for skull programs. After starting the rule engine you may use functions described in other chapters.

An example of a skull program follows. The `numstrdiff` skill implements an Evaluation method that computes the difference between the sum of all number arguments and the sum of the lengths of all string arguments. Other types of arguments are silently ignored.

```
#include <nb/nb.h>

static nbCELL numstrdiffEvaluate(
  nbCELL context,void *skillHandle,
  void *knowledgeHandle,nbCELL arglist){

  nbSET argSet;
  nbCELL cell;
  double length=0;
  int len=0;
  int type;

  argSet=nbListOpen(context,arglist);
  while((cell=nbListGetCellValue(context,&argSet))){
    type=nbCellGetType(context,cell);  // get cell type
    if(type==NB_TYPE_STRING){
      len+=strlen(nbCellGetString(context,cell)); // sum string lengths
      }
    else if(type==NB_TYPE_REAL){
      length+=nbCellGetReal(context,cell);        // sum numbers
      }
    nbCellDrop(context,cell);  // release each cell when done with it
    }
  cell=nbCellCreateReal(context,length-len);  // create cell with difference
  return(cell);  // return sum of numbers minus sum of string lengths
  }

static void *numstrdiffBind(
  nbCELL context,void *moduleHandle,
  nbCELL skill,nbCELL arglist,char *text){

  nbCELL facet=nbSkillFacet(context,skill,""); // get primary facet
  nbSkillMethod(context,facet,NB_NODE_EVALUATE,numstrdiffEvaluate);
  return(NULL);
  }

int main(int argc,char *argv[]){
  nbCELL context;

  context=nbStart(argc,argv);
  nbSkillDeclare(context,numstrdiffBind,NULL,"","numstrdiff",NULL,"");
  nbCmd(context,"define differ node numstrdiff;",NB_CMDOPT_ECHO);
  nbCmd(context,"show (differ(5,\"abcdefg\",2,\"abc\"));",NB_CMDOPT_HUSH);
  nbCmd(context,"show (differ(10,5,\"abcdefg\",2,\"abc\"));",NB_CMDOPT_HUSH);
  return(nbStop(context));
  }
```

The output from the three calls to nbCmd above should look like this.

```
> define differ node numstrdiff;
() = -3 == differ(5,"abcdefg",2,"abc")
() = 7 == differ(10,5,"abcdefg",2,"abc")
```

2.1 nbStart - Start rule engine

```
nbCELL nbStart(
  int argc,      // standard C main routine argument count
  char *argv[]); // standard C main routine arguments

Returns: Context handle for use in calls to other API functions
```

The nbStart function is used to initialize the rule engine as specified by the arguments. Only rule engine startup (`"++"`) arguments are processed.

2.2 nbServe - Serve engine arguments

```
nbCELL nbServe(
  int argc,      // standard C main routine argument count
  char *argv[]); // standard C main routine arguments

Returns: Context handle for use in calls to other API functions
```

The nbServe function is used to process a set of rule engine arguments, ignoring startup (`"++"`) arguments. This routine does not return until all arguments have been processed as described for the `nb` program. This means if the arguments put the engine in a server mode (daemon or servant), this function will not return until a `stop` command is issued in daemon mode, or an end-of-file on `stdin` is reached in servant mode.

2.3 nbStop - Stop rule engine

```
int nbStop(
  nbCELL context); // Context handle returned by nbStart

Returns: Rule engine exit code
```

The nbStop function closes files and releases memory allocated by NodeBrain.

Warning: This routine is not yet fully implemented. It does not clean up the environment. Repeated calls to nbStart and nbStop will create a major memory leak. So, you should only call nbStart and nbStop once in your program for now.

3 Module Functions

A module is a plugin to the rule engine. It provides one or more node skills. A node skill provides a set of skill methods that give a node functionality beyond that which the rule engine provides. Although modules are normally separate dynamically loaded modules, a skull program that embeds the rule engine can also call module functions to create new types of nodes.

The following is a sample module that implements a skill called `numstrdiff`. Notice the similarity to the skull example in the previous chapter. Here the nbBind method declares the skill instead of a main routine. As noted in the comments, the nbBind method is not always necessary.

```
#include <nb/nb.h>

static nbCELL numstrdiffEvaluate(
  nbCELL context,void *skillHandle,
  void *knowledgeHandle,nbCELL arglist){

  nbSET argSet;
  nbCELL cell;
  double length=0;
  int len=0;
  int type;

  argSet=nbListOpen(context,arglist);
  while((cell=nbListGetCellValue(context,&argSet))){
    type=nbCellGetType(context,cell);  // get cell type
    if(type==NB_TYPE_STRING){
      len+=strlen(nbCellGetString(context,cell)); // sum string lengths
      }
    else if(type==NB_TYPE_REAL){
      length+=nbCellGetReal(context,cell);        // sum numbers
      }
    nbCellDrop(context,cell);  // release each cell when done with it
    }
  cell=nbCellCreateReal(context,length-len);  // create cell with difference
  return(cell);  // return sum of numbers minus sum of string lengths
  }

static void *numstrdiffBind(
  nbCELL context,void *moduleHandle,
  nbCELL skill,nbCELL arglist,char *text){

  nbCELL facet=nbSkillFacet(context,skill,""); // get primary facet
  nbSkillMethod(context,facet,NB_NODE_EVALUATE,numstrdiffEvaluate);
  return(NULL);
  }

// if the module is named nb_numstrdiff.so, you can drop nbBind
// and change numstrdiffBind above from static to extern.
extern void *nbBind(nbCELL context,char *module,nbCELL argList,char *text){
  nbSkillDeclare(context,numstrdiffBind,NULL,"","numstrdiff",NULL,"");
  return(NULL);
  }
```

3.1 nbBind Method - Initialize module

```
extern void *nbBind(
  nbCELL context,  // Context handle returned by nbStart
  char *module,    // Name of module implicitly or explicitly declared
  nbCELL *argList, // Argument cells if explicitly declared - may be NULL
  char *text);     // Text options if explicitly declared - may be null string

Returns: Module handle - pointer to allocated structure or NULL
```

You may include an nbBind method in a module to perform initialization tasks. These tasks might include the allocation of a memory structure for use by any of the node skills provided by the module, and/or the declaration of skills. When initialization is not required, the nbBind method may be omitted from a module. This method does not apply to skull programs embedding the rule engine since a skull requires no special opportunity to perform initializaton tasks. For a module, the nbBind method is called once when the module is first loaded.

The `argList` and `text` will be null unless explicity declared.

```
declare module module dynamic_load_library[[(argList)]:text][;]
```

3.2 nbSkillDeclare - Register a skill

```
int nbSkillDeclare(
  nbCELL context,          // Context handle returned by nbStart
  void *(*bindFunction)(), //
  void *moduleHandle,      // Point to dynamic load module or NULL
  char *moduleName,        // Name of module or null string
  char *skillName,         // Name of skill
  nbCELL argList,          // Argument list or NULL
  char  *text);            // Option text or null string

Returns:
  0 - success
 -1 - error (see message)
```

This function is used to declare a new skill from within a program or module containing the functions to be used as skill methods. The `argList` and `text` arguments are the same as in a skill declaration command.

```
declare skill skill module.skill(arglist):text
```

With the nbSkillDeclare function you don't have an opportunity to alias the skill name, but do have an opportunity to alias the bind function. A common bind function may be used for multiple skills using different `argList` and `text` values.

3.3 nbSkillFacet - Register a skill facet

```
nbCELL nbSkillFacet(
  nbCELL context, // Context handle returned by nbStart
  nbCELL skill,   // Skill cell
  char  *name);   // Facet name - null string for primary facet

Returns: Facet cell for calls to nbSkillMethod
```

This function is used to add a facet to a skill and/or obtain a cell handle to the facet for use in calls to nbSkillMethod. Specify a null string for the name to get a handle to the primary facet. A multifaceted skill provides a node with multiple sets of skill methods—one set for each facet. Different facets can provide the same or different subsets of all possible methods.

When a node is referenced within a rule or command set, a facet name may be specified after the node name, separated by an underscore. In the example below, the `bob` node uses the `bobbie` skill of the `robert` module. The expression `bob_jack(10)` is handled by the Evaluate method of the `jack` facet of the `bobbie` skill. The expression `bob("abc")` is handled by the Assert method of the primary facet of the `bobbie` skill.

```
define bob node robert.bobbie;
define r1 on(bob_jack(10)>17) bob("abc");
```

Warning: This function is not available prior to version 0.9.00.

3.4 nbSkillMethod - Register a skill facet method

```
int nbSkillMethod(
  nbCELL context,  // Context handle returned by nbStart
  nbCELL facet,    // Facet cell
  int methodId,    // Method identifier code
  void *function); // Method callback function

Returns:
  0 - success
 -1 - error (see message)

Method Identifier Codes:

  NB_NODE_ALARM
  NB_NODE_ALERT
  NB_NODE_ASSERT
  NB_NODE_COMMAND
  NB_NODE_COMPUTE
  NB_NODE_CONSTRUCT
  NB_NODE_DESTROY
  NB_NODE_DISABLE
  NB_NODE_ENABLE
  NB_NODE_EVALUATE
  NB_NODE_SOLVE
  NB_NODE_SHOW
```

This function is used within a skill Bind method within a skull or module to bind a method to a skill facet. You must first obtain a facet cell using a call to nbSkillFacet with the skill cell provided as an argument to the Bind method.

Warning: This function is not available prior to version 0.9.00.

```
NB_NODE_ALARM
NB_NODE_ALERT
NB_NODE_ASSERT
NB_NODE_COMMAND
NB_NODE_COMPUTE
NB_NODE_CONSTRUCT
NB_NODE_DESTROY
NB_NODE_DISABLE
NB_NODE_ENABLE
NB_NODE_EVALUATE
NB_NODE_SOLVE
NB_NODE_SHOW
```

3.5 nbSkillSetMethod - Bind a skill method

This function is used within a skull or a Bind method within a module to bind a method to a skill. The skill cell is provided as an argument to a skill Bind method.

Warning: This function is deprecated but supported for compatibility with older modules. Starting with release 0.9.01, use the nbSkillFacet and nbSkillMethod functions instead.

3.6 nbVerbDeclare - Register a command verb

```
int nbVerbDeclare(
  nbCELL context,        // Context handle
  "module.verb", // Verb
  int auth,              // Authorization flags
  int flags,             // Reserved flags - always 0
  void *handle,          // Handle for module command method
  void *handler,         // Module command method
  char *helpText);       // A short string of help text

Returns:
  0 - successful
 -1 - error (see message)

Authorization Flags:
  NB_AUTH_CONNECT - connection permission
  NB_AUTH_ASSERT  - assert and alert permission
  NB_AUTH_DEFINE  - define, undefine, redefine permission
  NB_AUTH_DECLARE - declare permision (owner)
  NB_AUTH_SYSTEM  - shell command permission
  NB_AUTH_CONTROL - rank, set, stop (owner)
```

This function is used to create new commands. The verb is specified as *module.verb* to avoid conflict with commands provided by the rule engine.

3.7 Module Command Method - Handle a registered command verb

```
int moduleCmdVerb(
  nbCELL context, // Context cell
  void *handle,   // Handle specified in call to nbVerbDeclare
  char *verb,     // Verb specified in call to nbVerbDeclare
  char *text);    // Command body - text after verb

Returns:
  0 - successful
 -1 - error (see message)
```

The module command method is provided to process commands you define using nbVerb-Declare. This method is invoked each time a module command is specified as follows.

```
module.verb text
```

If your module has not been loaded, the rule engine loads the module first, if found. This will invoke your nbBind method, which calls nbVerbDeclare to register the command, and then the associated module command method is called. If the module is loaded by the verb is not defined, then an error message is displayed.

4 Skill Methods

A skill method is a function provided by a module to perform a particular task for nodes of a given type. The type of node is really defined by the set of skill methods associated with it. From this perspective, the notions of class and object in an object oriented language like C++ or Java are similar to our notions of skill and node. Skill methods are like class methods without the important object oriented notion of inheritance.

Method	**Purpose**
Alarm	Process an alarm
Alert	Process an alert command
Assert	Process an assert command
Bind	Associate methods with a skill
Construct	Allocate a node's knowledge structure and handle
Evaluate	Produce a value based on computed arguments
Compute	Produce a value by first computing argument values
Show	Display knowledge structure
Solve	Produce a value by first solving for unknowns
Enable	Start active participation
Disable	Stop active participation
Destroy	Free up knowledge structure
Command	Process a command

Some of the methods are passed a list of cells as an argument. The nbListOpen and nbList-GetCellValue functions may be used to iterate throught the cell list. The nbCellGetType, nbCellGetString, and nbCellGetReal functions are often used to convert the cell values into C data types.

```
type skill[_facet]Method(nbCELL context,...,nbCELL argList,...){
  nbCELL cell;
  nbSET set;
  int type;
  char *string;
  double real;

  set=nbListOpen(context,arglist);
  while((cell=nbListGetCellValue(context,set))){
    type=nbCellGetType(context,cell);
    if(type==NB_TYPE_STRING){
      string=nbCellGetString(context,cell);
      ...
      }
    else if(type==NB_TYPE_REAL){
      real=nbCellGetReal(context,cell);
      ...
      }
    else ...
    nbCellDrop(context,cell); // release cell fi done with it
    }
  }
```

If you want to access the cells in the `argList` instead of their values, then use the nbListGetCell function instead of nbListGetCellValue. This is necessary in cases where the cell provides a trigger condition for which you want to monitor the value instead of just obtaining the current value. It is also necessary to use this approach when the cells are not assumed to be enabled, as in the Compute method.

```
type skill[_facet]Method(nbCELL context,...,nbCELL argList,...){
  nbCELL cell;
  nbSET set;

  set=nbListOpen(context,arglist);
  while((cell=nbListGetCell(context,set))){
    ... process cell ...
    nbCellDrop(context,cell); // release cell if done with it
    }
  }
```

4.1 Alarm - Handle node alarm

```
void *skill[_facet]Alarm(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL cell);      // Optional cell to process on alarm

Returns: void
```

```
synapse=nbSynapseOpen(context,skillHandle,nodeHandle,cell,skill[_facet]Alarm);
nbSynapseSetTimer(context,synapse,seconds);
```

4.2 Alert - Handle node sentence alert

```
static oid *skill[_facet]Alert(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList,    // Argument cell list or NULL
  nbCELL value);     // Value cell

Returns:
  0 - success
 -1 - error (see message)
```

4.3 Assert - Handle node sentence assertion

```
static void *skill[_facet]Assert(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList,    // Argument cell list or NULL
  nbCELL value);     // Value cell

Returns:
  0 - success
 -1 - error (see message)
```

The Assert method is called to support assertions. It is intended to store knowledge in a node's memory for later reference. However, you can use it to take whatever action you like.

The general syntax for an assertion to a node includes an optional facet, arguments and a cell expression providing the value. There are shortcuts for three special values.

```
node[_facet](argList)=valueCellExpression
node[_facet](argList)    same as    node[_facet](argList)=1
!node[_facet](argList)   same as    node[_facet](argList)=!
?node[_facet](argList)   same as    node[_facet](argList)=?
```

This syntax may appear in an `assert` command, `alert` command, or in a rule action. When the node is specified in the command prefix, it is implied on an argument list. The last three lines in the example below show three ways the `treeAssert` method could get called.

```
define spruce node tree;
assert spruce("abc","def")=27;            # assert command
spruce. assert ("abc","def")=27;          # assert command with node as prefix
define r1 on(a=2) spruce("abc","def")=27; # rule with assertion
```

4.4 Bind - Create a skill handle and bind methods

```
void *skillBind(
  nbCELL context, // Context handle
  void *module,   // Module handle
  nbCELL skill,   // Skill cell for registering methods
  nbCELL argList, // Argument cell list or NULL
  char *text);    // Text options or null string

Returns:    Skill handle - pointer to an allocated structure, or NULL
```

The Bind method is invoked by nbSkillDeclare, which is called either by a skull program, a module's nbBind method, or by the rule engine for skills that are referenced without prior declaration. The Bind method is called once for a give skill. It is responsible for binding skill methods to the skill using the functions described in the *Node Module Functions* chapter.

The `argList` and `text` arguments are null unless explicitly declared as shown below, or specified in a call to nbSkillDeclare.

```
declare skillAlias skill module.skill[(argList)][:text|;]
```

4.5 Command - Handle node command

```
static int skill[_facet]Command(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList,    // Argument cell list or NULL
  char *text);       // Command text or null string

Returns:
  0 - successful
 -1 - error (see message)
```

The Command method is called to support extended node commands. This method may be used to implement your own command language for use within NodeBrain rules. It may also be used to implement commands to control specific nodes. We list some possibilities here, but you are free to use it as you like.

- Turn on and off a special trace option.
- Request a debugging dump of a node's knowledge.
- Initialize a node from an external file or database.
- Purge obsolete or expired knowledge.
- Inject knowledge in a unique syntax (an alternate form of assertion).

Node commands start with the node name which must be followed by a left parenthesis starting an argument cell list or a colon starting command text.

```
define node node definition
node[(argList)][:text][;]
```

4.6 Compute - Handle node sentence computation

```
static nbCELL skill[_facet]Compute(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList);   // Argument cell list or NULL

Returns: Value cell
```

The Compute method is similar to the Evaluate method, only it doesn't assume the argument cells are currently enabled. You need only implement this method if you want to achieve a performance improvement over NodeBrain's default handling for this method. By default, NodeBrain enables the argument list (causing all argument expressions to be enabled and evaluated), calls the Evaluate method, and then disables the argument list. For some types of nodes you can avoid unnecessary evaluation of disabled argument expressions by providing your own Compute method. For example, an implementation of AND or OR would only require computation of the second argument if the first argument were True for AND or not True for OR.

Use the nbCellCompute function to compute the value of arguments. The following example computes all argument cells for illustration only—this would make the Compute method unnecessary.

```
static nbCELL fooCompute(nbCELL context,void *skillHandle,void *nodeHandle,nbCELL argList){
  nbCELL cell,valueCell;
  nbSET set;

  set=nbListOpen(context,argList);
  while((cell=nbListGetCell(context,set))){
    valueCell=nbCellCompute(context,cell);
    ...
    }
  }
```

4.7 Construct - Handle node definition

```
static void *skillConstruct(
  nbCELL context,     // Context handle
  void *skillHandle,  // Pointer to skill configuration structure
  nbCELL argList,     // Argument cell list or NULL
  char *text);

Returns: Knowledge handle - pointer to an allocated structure
```

The Construct method is called when a node is defined. It is responsible for processing the argument cells and interpreting text provided in the definition. It may allocate a structure for knowledge representation. It returns a pointer to the allocated memory structure which is referenced as `nodeHandle` in the description of other methods.

The `argList` and `text` arguments are from the node definition as shown below.

```
define node node [module.]skill[(argList)][:text | ;]
```

4.8 Enable - Handle node enable command

```
static nbCELL skill[_facet]Enable(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle); // Pointer to node knowledge structure

Returns:
  0 - success
 -1 - error (see message)
```

The Enable method is called to support the `enable` command for a node. You should only implement this method for skills that requirement management of enabled and disabled states.

```
define node node definition
enable node;
```

Some skills require a node to be enabled when the rule engine daemonizes—switches to background agent mode. This can be scheduled using the nbListenerEnableOnDaemon function.

4.9 Evaluate - Handle node sentence evaluation

```
static nbCELL skill[_facet]Evaluate(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList);   // Argument cell list or NULL

Returns: Value cell
```

The Evaluate method is called to produce a value for a node condition. This method assumes the argument values have already been determined. It is normally invoked for a node condition when an argument value has changed.

4.10 Destroy - Handle node destruction

```
static nbCELL skill[_facet]Destroy(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle); // Pointer to node knowledge structure

Returns:
  0 - success
 -1 - error (see message)
```

The Destroy method is used to free up memory allocated for a node's knowledge representation. This is only be implemented for skills implementing the Construct method. While it is preferable to implement this method when Construct is implemented, it is not required. If it is not implemented, NodeBrain will be unable to undefine a node associated with the skill.

```
define node node definition
...
undefine node
```

4.11 Disable - Handle node disable command

```
static nbCELL skill[_facet]Disable(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle); // Pointer to node knowledge structure

Returns:
  0 - success
 -1 - error (see message)
```

The Disable method is called to support the `disable` command for a node. You will normally implement this method only when you also implement the Enable method.

```
define node node definition
...
disable node
```

4.12 Show - Handle request to display node

```
static nbCELL skill[_facet]Solve(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  int option);       // Option

Returns:
  0 - success
 -1 - error (see message)

Options:
  NB_SHOW_ITEM   - Show as single string without \n for glossary display
  NB_SHOW_REPORT - Show a multi-line report of skill and knowledge structures
```

The Show method is called to display a text representation of a node. When NB_SHOW_ITEM is specified, this method displays a single string for use when displaying a term glossary. This is done using the nbLogPut function with arguments like the `printf` function without including \n. When NB_SHOW_REPORT is specified, a multi-line report of a node's skill memory and node memory is produced using nbLogPut with \n characters included as needed to format the report.

In the example below the rule engine asks the spruce tree to display in NB_SHOW_REPORT format. While listing the terms in the spruce glossary, the fir tree is asked to display in NB_SHOW_ITEM format.

```
define spruce node tree;
spruce. define a cell 5;
spruce. define fir node tree;
show spruce
```

4.13 Solve - Resolve unknowns and compute node sentence

```
static nbCELL skill[_facet]Solve(
  nbCELL context,    // Context handle
  void *skillHandle, // Pointer to skill configuration structure
  void *nodeHandle,  // Pointer to node knowledge structure
  nbCELL argList);   // Argument cell list or NULL

Returns: Value cell
```

This Solve method is similar to the Compute method, only it uses the nbCellSolve function to resolve argument cells instead of nbCellCompute. The nbCellSolve function is more determined than the nbCellCompute function. When nbCellCompute is asked to compute an term with an Unknown definition, it returns the Unknown object. When nbCellSolve is asked to solve for an term with an Unknown definition, it attempts to obtain a definition and compute a value. A solve method should only solve for argument cells as necessary to produce a value.

You need only implement the Solve method if you want your node to work in diagnostic mode more efficiently than NodeBrain's default behavior. By default, NodeBrain will attempt to solve for all argument cell expressions and then invoke your Evaluate method.

5 Node Functions

Function described in this chapter interact with nodes and the first argument is a node context handle. You may use the context returned by nbStart and the first argument passed to a skill method as the node context handle. You may also use a node context handle returned by other API functions.

5.1 nbCmd - Interpret command

```
int nbCmd(
  nbCELL context, // Node context handle
  char  *command, // Pointer to skill configuration structure
  int    option); // Argument cell list or NULL

Returns:
 -1 - Syntax or semantic error. (See message)
  0 - Successfully processed
  n - Command specific return code

Options:
  0                - Use options defined for the context
  NB_CMDOPT_ECHO  - echo the command
  NB_CMDOPT_HUSH  - suppress echo
  NB_CMDOPT_TRACE - trace command execution
```

The nbCmd function gives a skull program or module the full range of functionality provided by the NodeBrain command language.

5.2 nbListenerEnableOnDaemon - Schedule enable when daemonized

```
int nbListenerEnableOnDaemon(
  nbCELL context); // Node context handle

Returns: 0
```

This function is normally used by a skill's Construct method when it needs to delay going into an active state until an agent deamonizes (goes into the background). At this time all of the rules required by the node have been loaded. The rules and other cells defined for a node are defined after the node is constructed. The Enable method may look at these cells to complete configuration before going into and enable state. If a node listens on a network socket for events to process against the rules defined for the node, this is another reason to wait until all the rules have been defined before listening. The nbListenerEnableOnDaemon function provides a call to the Enable method at the appropriate time. Many skills will not need a delayed enable, and some will not even implement an Enable method.

5.3 nbLogMsg - Write a message to the log

```
int nbLogMsg(
  nbCELL context,  // Node context handle
  int   msgNumber, // Message number - 0 to 999
  char  msgType,   // Message type
  char *format,    // Format string for printf function
  ...);            // Variable number of arguments

Returns:
 -1 - Error  (See message)
  0 - Successfully processed

Message Types:
  (T)race
  (I)nformation
  (W)arning
  (E)rror
  (L)ogic error
```

The nbLogMsg function is used to write a message to the log, `stderr`. A message is prefixed with a time stamp, message identifier, module name, and node name.

In the example below, the message identifier of NM000E has three components. The number "000" and type "E" are provided by the caller. The "NM" identifies it as coming from the nbLogMsg function used by skulls and modules. The rule engine uses an internal form of this function that generates "NB" message identifiers. If you use non-zero message numbers, they only need to be unique per module because the module name is listed right after the message identifier. The node name helps to identify which node produced the message, since there may be many nodes using a given modules.

```
> define fred node tree;
> fred:sksksks
2014/02/23 17:01:49 NM000E tree .fred: Verb "sksksks" not recognized.
```

The message above was produced by the following call to nbLogMsg.

```
nbLogMsg(context,0,'E',"Verb \"%s\" not recognized.",verb);
```

5.4 nbLogPut - Write text to the log

```
int nbLogMsg(
  nbCELL context,  // Node context handle
  char *format,    // Format string for printf function
  ...);            // Variable number of arguments

Returns:
 -1 - Error  (See message)
  0 - Successfully processed
```

The nbLogPut function is called to write to the log, `stderr`. With this function you have complete control of the output. You may find this function has no advantage over `printf`; This is often true. But use of this function provides as place to insert additional functionality, like temporarily splitting log data out to a second file.

6 Cell Functions

The functions described in this chapter are for interacting with the rule engine at the cell level. It is possible to extend NodeBrain by adding your own "action" commands without using any of these functions. However, most programs that use the NodeBrain Library to extend NodeBrain will need to use some of these functions.

6.1 nbCellCompute - Compute value of disabled cell

```
nbCELL nbCellCompute(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to compute

Returns: Constant cell representing computed value

Warning: When you are finished with the returned value cell, you
         must call nbCellDrop to release it to avoid a memory leak.

See also: nbCellEvaluate, nbCellGetValue, and nbCellSolve.
```

Use nbCellCompute to compute the value of a cell that may be disabled. A cell is disabled when there are no enabled rules depending on it. The value of a disabled cell is not maintained in response to assertions. A call to nbCellCompute will return the value of an enabled cell just like nbCellGetValue. But for a disabled cell, nbCellCompute will recursively call nbCellCompute on operand cells and then nbCellEvaluate to compute the cell value. The resulting value is the same as would have already been assigned if the cell had been enabled.

6.2 nbCellCreate - Create cell from expression

```
nbCELL nbCellCreate(
  nbCELL context,         // Node context handle
  char  *cellExpression); // Cell expression

Returns: Cell handle, or NULL on error (see message)

Warning: When you are finished with the returned cell, you must
         call nbCellDrop to release it to avoid a memory leak.
```

The `nbCellCreate` function is used to obtain a handle to a cell. If the specified cell does not exist, it is created and a handle is returned. If the cell already exists, a handle to the existing cell is returned.

6.3 nbCellCreateReal - Create number cell

```
nbCELL nbCellCreateReal(
  nbCELL context,  // Node context handle
  double real);    // Real number

Returns: Handle for real number cell

Warning: When you are finished with the returned cell, you must
         call nbCellDrop to release it to avoid a memory leak.

Warning: Prior to version 0.9.01 this function failed to grab
         the created cell as intended. If you have code that used
         this function under prior versions that accidentally
         worked, it may break under 0.9.01 and above, requiring
         a fix to the calling code.
```

The `nbCellCreateReal` function is used to obtain a handle to a real number cell. If the specified cell does not exist, it is created and a handle is returned. If the cell already exists, a handle to the existing cell is returned. The value returned by this function is the same as `nbCellCreate` would return if passed the string representation of the number.

6.4 nbCellCreateString - Create string cell

```
nbCELL nbCellString(
  nbCELL context,  // Node context handle
  char  *string);  // String

Returns: Handle for string cell

Warning: When you are finished with the returned cell, you must
         call nbCellDrop to release it to avoid a memory leak.
```

The `nbCellCreateString` function is used to obtain a handle to a string cell. If the specified cell does not exist, it is created and a handle is returned. If the cell already exists, a handle to the existing cell is returned. The value returned by this function is the same as `nbCellCreate` would return if passed the string enclosed in double quotes.

6.5 nbCellDrop - Release cell

```
nbCELL nbCellDrop(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to drop

Returns: NULL (always)
```

Each cell has a reference count. The `nbCellDrop` function is called to decrement the reference count and free the cell if the count drops to zero.

While you may explicitly reserve a cell with a call to `nbCellGrab`, you must also know when other functions that return a cell pointer have placed a reserve on it for you. If so, you are still responsible for issuing a `nbCellDrop` when you are finished with the pointer. While this is a burden, it is required for the rule engine to know when an application is finished with a cell.

As you might imagine, you will get very unhappy results if you make too many calls to `nbCellDrop`. You can easily cause a cell to be released and the memory reused while there are still other pointers to the location of the released cell. One could say the result is unpredictable, but it is probably better to say the result is predictably bad.

```
# drop cell and clear my pointer to it
nodeHandle->somethingCell=nbCellDrop(nodeHandle->somethingCell);
```

6.6 nbCellEvaluate - Compute enabled cell value after change

```
nbCELL nbCellEvaluate(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to compute

Returns: Constant cell representing cell value

Warning: When you are finished with the returned value cell, you
         must call nbCellDrop to release it to avoid a memory leak.

See also: nbCellCompute, nbCellGetValue, and nbCellSolve
```

Use `nbCellEvaluate` to compute the value of an enabled cell after a change to an operand. This function simply provides an interface to the evaluation method of the cell. Normally the rule engine is aware of the need to call a cell evaluation method and performs this step automatically. However, it is possible for an application to implement cells for which the rule engine is not aware of all operands involved in an evaluation. In such a case, there are operands that are not cells known to the rule engine. Any change to these operands would then require the application to call `nbCellEvaluate`, or directly call the evaluation method.

6.7 nbCellGetReal - Get number from number cell

```
double nbCellGetReal(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Real number cell

Returns: Real number stored in specified cell
```

Use `nbCellGetReal` to extract a C `double` value from a real number cell.

6.8 nbCellGetString - Get string from string cell

```
char *nbCellGetString(
  nbCELL context,  // Node context handle
  nbCELL cell);    // String cell

Returns: String stored in specified cell
```

Use `nbCellGetString` to extract a C `char` pointer from a string cell.

6.9 nbCellGetType - Get cell type code

```
int nbCellGetType(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to inspect

Returns: Cell type code

Type Codes:
  NB_TYPE_UNDEFINED
  NB_TYPE_DISABLED
  NB_TYPE_FALSE
  NB_TYPE_UNKNOWN
  NB_TYPE_PLACEHOLDER
  NB_TYPE_STRING
  NB_TYPE_REAL
  NB_TYPE_LIST
  NB_TYPE_TERM
  NB_TYPE_NODE
  NB_TYPE_VERB
  NB_TYPE_TEXT
```

The `nbCELL` data type is a pointer to a cell. When returned by a library function, or passed to a method, you can call `nbCellGetType` to determine what type of cell you have. If it is not an expected type, you can call `nbLogMsg` to report an error.

6.10 nbCellGetValue - Get enabled cell value

```
nbCELL nbCellGetValue(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to compute

Returns: Constant cell representing cell value

Warning: When you are finished with the returned value cell, you
         must call nbCellDrop to release it to avoid a memory leak.

See also: nbCellCompute, nbCellEvaluate, and nbCellSolve.
```

Use nbCellGetValue to get the current value of an enabled cell. This function is used when there is no need to compute a new value in response to changing operands—when you can assume the rule engine has already responded to changes at the level of the cell in the axon tree (bottom up tree of dependencies).

6.11 nbCellGrab - Reserve cell

```
nbCELL nbCellGrab(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to drop

Returns: The cell argment

See also: nbCellDrop
```

Each cell has a reference count. The `nbCellGrab` function is called to increment the reference count to ensure the cell will not be released while in use—while there is a pointer to it.

```
nbCELL cell=...; // obtained somehow without a grab
nodeHandle=malloc(sizeof(mynode));
nodeHandle->somethingCell=nbCellGrab(context,cell);
```

Make sure you know if the process you use to create or obtain a cell pointer has already grabbed the cell on your behalf. If so, you should only call `nbCellGrab` if you need an additional grab—when you store the cell pointer more than once and have the possibility of calling `nbCellDrop` on each pointer independently. When you are finished with a cell pointer that you have reserved using a call to `nbCellGrab`, or that was grabbed for you by another function, you must release the cell pointer with a call to `nbCellDrop` to avoid a memory leak.

6.12 nbCellSolve - Solve for value of cell

```
nbCELL nbCellSolve(
  nbCELL context,  // Node context handle
  nbCELL cell);    // Cell to compute

Returns: Constant cell representing cell value

Warning: When you are finished with the returned value cell, you
         must call nbCellDrop to release it to avoid a memory leak.

See also: nbCellCompute, nbCellEvaluate, and nbCellGetValue.
```

Use nbCellSolve to compute the value of a cell for which the value is unknown because it references terms that are undefined. The goal of nbCellSolve is to obtain definitions for referenced undefined terms sufficient to derive a value other than Unknown for the cell, if possible. A term is *undefined* if it has no formula. The value of an undefined term is Unknown, but a term can have a value of Unknown without being undefined, because a formula can evalute to Unknown. When a term definition is a constant (e.g. "abc"), the value is the same constant. It is often the case in diagnostic mode, where nbCellSolve is used, that we expect to obtain a constant definition for a term, in which case it is the same as directly seeking a value.

For example, a call to nbCellCompute for the cell `A or B`, where A and B are undefined, will first attempt to obtain a formula (definition) for `A` and solve the formula to obtain a value for `A`. If the value is True, the value of `A` is returned and no attempt is made to obtain a definition for `B`. For `A and B`, a False or Unknown value for `A` would be sufficient to compute the cell value without bothering with `B`. Like a good doctor or car mechanic, a NodeBrain diagnostic rule writer should specify the conditions so the least expensive tests are performed first, enabling the most expensive tests to be avoided when possible.

Warning: This feature of NodeBrain requires additional work to refine the methods by which definitions can be obtained. Currently, prompts in interactive mode and scripts in background mode are supported.

6.13 nbListOpen - Open a cell list to iterate

```
nbSET nbListOpen(
  nbCELL context,  // Node context handle
  nbCELL cell);    // List cell to open

Returns: List set cell cursor
```

The `nbListOpen` function is used to obtain a set iterator for a list of cells. Skill methods often use this function, and related functions, to process an argument list, which is represented as a list of cells.

6.14 nbListGetCell - Get next cell in list

```
nbCELL nbListGetCell(
  nbCELL context,  // Node context handle
  nbSET cell);     // List set cell cursor

Returns: Next cell in the set, or NULL at end of list

Warning: The returned cell has been grabbed, so you must
         call nbCellDrop when it is no longer needed.
```

The `nbListGetCell` function is used obtain the next cell in a list of cells. The next cell is returned and the set iterator advances. A NULL value is returned when the end of the list is reached.

6.15 nbListGetCellValue - Get value of next cell in list

```
nbCELL nbListGetCellValue(
  nbCELL context,  // Node context handle
  nbSET cell);     // List set cell cursor

Returns: Value cell for Next cell in the set, or NULL if end of list

Warning: The returned cell has been grabbed, so you must
         call nbCellDrop when it is no longer needed.
```

The `nbListGetCellValue` function returns the value of the next cell in a list. The value of a cell is itself a cell. The set iterator steps within the list on each call and NULL returned at the end of the list.

6.16 nbSynapseOpen - Register an Alarm method

6.17 nbSynapseSetTimer - Set an alarm

6.18 nbTermCreate - Create a new term

```
nbCELL nbTermCreate(
  nbCELL context,     // Node context handle
  char *identifier,   // Term to create within context
  nbCELL definition); // Cell that defines the term

Returns: Term cell, or NULL on error (see message)

Warning: The returned cell has been grabbed, so you must
         call nbCellDrop when it is no longer needed.
```

The `nbTermCreate` function is used to create a new term within the specified context. The definition of the term is provided by a cell handle obtained by another library function, perhaps `nbCellCreate`.

6.19 nbTermLocate - Locate an existing term

```
nbCELL nbTermLocate(
  nbCELL context,     // Node context handle
  char *identifier);  // Term identifier to locate

Returns: Term cell handle, or NULL if not found

Warning: The returned cell has been grabbed, so you must
         call nbCellDrop when it is no longer needed.
```

The `nbTermLocate` function is used lookup a term within the specified context.

6.20 nbTermSetDefinition - Assign new definition to term

```
nbCELL nbTermSetDefinition(
  nbCELL context,     // Node context handle
  nbCELL term,        // Term to update
  nbCELL definition); // Cell that defines the term

Returns: Cell handle of new value

Warning: The returned cell has been grabbed, so you must
         call nbCellDrop when it is no longer needed.
```

The `nbTermSetDefinition` function is used to redefine a term. The new value returned is the value of the new definition.

```
nbCELL term,definition,value;
nbCmd(context,"assert a=1,b=1;",0);
definition=nbCellCreate(context,"a*b");
term=nbTermCreate(context,"bob",definition);
definition=nbCellDrop(context,definition);
definition=nbCellCreate(context,"a+b");
value=nbTermSetDefinition(context,term,definition); // Redefine "bob"
// value
nbCellDrop(context,term);
nbCellDrop(context,definition);
nbCellDrop(context,value);
```

7 Medulla Functions

8 Spine Functions

Index

A

alarm method 20
alert method 20
assert method 21

B

bind method 21

C

cell functions 31
cerebrum 5
command language 1
command method 22
compute method 22
concepts 1
construct method 23

D

destroy method 24
disable method 25

E

enable method 24
evaluate method 24

I

Interface Modules 4
interface structure 1

L

Library Functions 4

M

medulla 6
medulla Functions 39
methods 7
module command method 18
module functions 13

N

nb servant 4
nbBind method 14
nbCellCompute 31
nbCellCreate 31
nbCellCreateReal 32
nbCellCreateString 32
nbCellDrop 32
nbCellEvaluate 33
nbCellGetReal 33
nbCellGetString 34
nbCellGetType 34
nbCellGetValue 34
nbCellGrab 35
nbCellSolve 35
nbCmd 27
nbListenerEnableOnDaemon 27
nbListGetCell 36
nbListGetCellValue 37
nbListOpen 36
nbLogMsg 28
nbLogPut 28
nbServe 11
nbSkillDeclare 15
nbSkillFacet 15
nbSkillMethod 16
nbSkillSetMethod 17
nbStart 11
nbStop 11
nbSynapseOpen 37
nbSynapseSetTimer 37
nbTermCreate 37
nbTermLocate 37
nbTermSetDefinition 38
nbVerbDeclare 18
node functions 27

P

piping commands 2

S

servant interface 3
servant module 3
servants 3
show method 25
Skill Methods 19
skull functions 9
solve method 26
source files 2
spine 6
spine functions 41