Oracle® Communications
Offline Mediation Controller

Cartridge Development Kit NPL Reference Guide

Release 12.0

E91421-01

December 2017

ORACLE®

Oracle Communications Offline Mediation Controller Cartridge Development Kit NPL Reference Guide, Release 12.0

E91421-01

Contents

4 Sample Mapping for ECE Cartridge Pack

A NPL Syntax and Reserved Words

Preface

This document describes the Node Programming Language (NPL) used by nodes within the Offline Mediation Controller framework. This document can be used in conjunction with the Cartridge Developer's Guide and the Cartridge Development Kit (CDK) to develop software cartridges for Offline Mediation Controller.

Audience

This document is intended for developers who develop nodes to use within a Offline Mediation Controller system. It is expected that the reader has an understanding of Offline Mediation Controller concepts.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

1

Node Programming Language

This chapter describes the Oracle Communications Offline Mediation Controller Node Programming Language (NPL).

Overview

Node Programming Language (NPL) is a simple programming language that is part of the Cartridge Development Kit (CDK) for use by nodes developed within the Offline Mediation Controller Framework. The Node Programming Language is used to express the translation of the data input into a node to the data that is output from a node. An NPL program processes one data record at a time and translates that data record based on its program statements into zero or more data records that are passed back to the node for output by that node.

Three major types of nodes use NPL in a Offline Mediation Controller system: Collection Cartridge (CC), Enhancement Processor (EP) and Distribution Cartridge (DC). A CC should use NPL primarily to map input data to NAR(s), which is the known data format within the Offline Mediation Controller system. EP nodes use NPL in a variety of ways including: filtering irrelevant data (individual fields or entire records), arithmetic calculations on fields, adding new fields, and modifying existing fields. A DC should use NPL primarily to map data from the NAR format to an output format for use by another program or the end user.

Interface with CDK components

A node must contain an NPLFieldProcessor object in order to process its data records with an NPL program. The entry point to a field processor's NPL program is one of its processData methods:

- public void processData(DCFieldContainer inputData)
- throws NodeProcessingException
- public void processData(DCFieldContainer[] inputData)
- throws NodeProcessingException

The first method above passes the input data record (represented as a DCFieldContainer) to the NPL program for processing. The second method will essentially loop through the array of input data records and pass each one to the NPL program for processing. A node may use either one of these methods directly to pass its data to an NPL program, or run the NPLFieldProcessor as a thread in which case the field processor will retrieve the input data from the node and perform the processData call for the node. The relevant point here with respect to NPL is to show how a data record is passed to an NPL program for processing.

The NPL write command will return the output record(s) from the NPL program to the NPLFieldProcessor. The NPLFieldProcessor will forward the record(s) to its Data Receiver based on the type of data processing model being used (either push or pull) within the node. The Data Receiver of the NPLFieldProcessor provides the destination for data records output from an NPL program. The Data Receiver of an NPLFieldProcessor object is set using the setDataReceiver method.

Expose clauses

There are two types of expose clauses:

- Expose clause for routing, described below
- Expose clause for aggregator node. See the discussion about programmable aggregation processor node in the *CDK Developer's Guide*.

An expose clause is used to make a subset of the input or output attributes available to the node for a specific reason. An expose clause declaration consists of the keywords Expose for, followed by a reason identifier, followed by one or more expose attributes enclosed within a single set of braces:

Expose for *reason-identifier { (expose-attribute)+ } ;*

where an expose-attribute is the attribute identifier followed by a user-friendly tag to be associated with the attribute:

record-identifier.attribute-identifier string-literal ;

An expose clause's reason-identifier must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. The record-identifier must be the identifier of an input or output record already declared, and the attribute-identifier must be an attribute declared in that input or output record.

Sets of exposed attributes are available to the node via the NPLFieldProcessor's getExposedFields method. To use this method, a node provides a reason identifier to indicate the expose clause of interest. The method will return an ArrayList of FieldDescriptor objects, one for each attribute in the corresponding expose clause. A FieldDescriptor object contains the attribute's type, identifier, and tag.

Expose clause for Routing

- One reason-identifier, Routing, is already pre-defined for all nodes. By declaring an expose clause for routing (i.e. Expose for Routing), a node makes those attributes available for modulus and directed routing.
- Below is an example of NPL statements for an CC, which will expose several attributes that may be used for Directed and Modulus routing.

Example: Routing with NPL

```
Integer seq_num = -1; // Sequence counter
Short source_type = 5;

InputRec
{
    String     Source-Serial-Number;
    TimeInSecs Last-Time;
    String     Acct-Status-Type;
    String     User-Name;
    String     Tunnel-Type;
    Long       Acct-Session-Id;
    IP         Framed-IP-Address;
```

```
    Long       Acct-Link-Count;
    Long       Acct-Session-Time;
    Long       Acct-Input-Octets;
    Long       Acct-Output-Octets;
    Long       Acct-Input-Packets;
    Long       Acct-Output-Packets;
    String     Gprs-Record-Type;
    IP         Gprs-GGSN-Address;
    Long       Acct-Multi-Session-Id;
} in;

OutputRec
{
    Short      20100;
    String     20101;
    TimeInSecs 20003;
    Integer    10004;
    String     20103;
    String     10005;
    String     20104;
    Long       20105;
    IP         20106;
    Long       20107;
    Long       20004;
    Long       10006;
    Long       10007;
    Long       10008;
    Long       10009;
    String     20108;
    IP         20000;
    Long       20001;
} out;

// Expose integral attributes for Directed and Modulus routing
```

Expose for Routing

```
{
    out.10004  "NAR Sequence Number";
    out.10006  "Acct Input Octets";
    out.10007  "Acct Output Octets";
    out.10008  "Acct Input Packets";
    out.10009  "Acct Output Packets";
    out.20001  "Acct Multi-Session ID";
    out.20004  "Acct Session Time";
    out.20100  "NAR Source Type";
    out.20105  "Acct Session ID";
    out.20107  "Acct Link Count";
}

if( seq_num == 2147483647 ) {
    seq_num = 0;
}
else {
    seq_num = seq_num + 1;
}

// Map input fields to output fields.
out.20100  = source_type;
out.20101  = in.Source-Serial-Number;
out.20003  = in.Last-Time;
```

```
out.10004  = seq_num;
out.20103  = in.Acct-Status-Type;
out.10005  = in.User-Name;
out.20104  = in.Tunnel-Type;
out.20105  = in.Acct-Session-Id;
out.20106  = in.Framed-IP-Address;
out.20107  = in.Acct-Link-Count;
out.20004  = in.Acct-Session-Time;
out.10006  = in.Acct-Input-Octets;
out.10007  = in.Acct-Output-Octets;
out.10008  = in.Acct-Input-Packets;
out.10009  = in.Acct-Output-Packets;

// Write out the output record.
write( out );
```

NPL compilation

NPL is a simple programming language that is part of the Cartridge Developer's Kit (CDK) for use by cartridges developed within the Offline Mediation Controller Framework. The Node Programming Language consists of a limited set of language components similar to most programming languages, including: comments, data types, variables, assignment statements, conditional statements and arithmetic expressions. In addition, NPL provides a set of built-in functions that are useful in manipulating the fields in a data record and an expose clause for providing access to attribute information of records declared in an NPL program.

Data Dictionary

The Data Dictionary identifies all the pre-integrated attributes used in the system. It is a flat file identifying each attribute and its corresponding numeric ID. Using the data dictionary ensures attributes do not overlap and each is always used for a specific purpose. This greatly simplifies the NPL work involved when multiple nodes are chained together.

The Data Dictionary file is located in *OMC_Home*/datadict, where *OMC_Home* is the directory in which you installed Offline Mediation Controller.

The runtime Offline Mediation Controller environment does not use this .xml file. Its purpose is to provide the reference to the NAR IDs.

Language components

The Node Programming Language consists of a limited set of language components common to most programming languages including:

- comments
- data types
- variables
- assignment statements
- conditional statements
- arithmetic expressions.

In addition, NPL provides a set of built-in functions that are useful in manipulating the fields in a data record, an expose clause for providing access to attribute

information of records declared in an NPL program, and a configuration clause to set configuration values that may be needed by the node. See "EBNF for NPL" for the syntax of NPL expressed in extended Backus-Naur form.

Comments

Comments in NPL are expressed in a manner similar manner to Java or C++. Any text following on the same line as a `//` character sequence is a comment. In addition, the characters `/*` introduce a comment which is terminated with the characters `*/`. Note however that comments delimited by the `/*` and `*/` character sequences cannot be nested.

Some common examples of these comment styles are:

```
// A comment.
X = 1; // another comment
Y = 2; /* another comment variation */
/* A comment variation. */
/*
 * A block comment.
 */
```

Data types

NPL is a strongly typed language. The primary purpose of NPL is to provide the capability to manipulate the fields in the data records being processed by the Offline Mediation Controller system. Therefore, the data types supported by NPL correspond to the data types of the fields in these data records. Section Interface with CDK Components above illustrates that a data record enters an NPL program as a `DCFieldContainer`. As the name implies, the `DCFieldContainer` that represents a data record consists of DCField objects representing the fields of that data record. The data types of NPL correspond to the data types represented by the `DCField` classes supported by the CDK. In addition, NPL provides types for the declaration of input and output record variables. See Table 1–1 for a summary of the data types supported by NPL.

Table 1–1 NPL Data Types

Type	Description	Range	DCField Type
Byte	8-bit integer type	-128 to 127	BYTE
Short	16-bit integer type	-32768 to 32767	SHORT
Integer	32-bit integer type	-2 147 483 648 to 2 147 483 647	INTEGER
Long	64-bit long integer type	-9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	LONG
TimeInSecs	32-bit integer type representing the number of seconds since January 1, 1970 00:00:00 UTC	-2147483648 to 2147483647	TIMEINSECONDS
TimeInMilliSecs	64-bit long integer type representing the number of milliseconds since January 1, 1970 00:00:00.000 UTC	-9223372036854775808 to 9223372036854775807	TIMEINMILLIS

Table 1–1 (Cont.) NPL Data Types

Type	Description	Range	DCField Type
Float	IEEE 754 32-bit single-precision floating-point type	approximately ± 3.40282347E+38	FLOAT
Double	IEEE 754 64-bit double-precision floating-point type	approximately ±1.7976931348623157E+308	DOUBLE
String	character string type	N/A	String
IP	32-bit IP address type	N/A	IP
IPv6	128-bit IP address type (Internet Protocol Version 6)	N/A	IPV6
MAC	Media Access Control address type	N/A	MAC
Bytes	array of bytes	N/A	BYTES
Object	externalizable Java object type	N/A	OBJECT
List	externalizable list of externalizable Java objects	N/A	LIST
UInt128	128-bit unsigned integer type	0 to	UINT128
InputRec	NPL input record type	N/A	N/A
OutputRec	NPL output record type	N/A	N/A
Map	Map of DC field objects, keyed on field ID	N/A	MAP

Note that NPL data types use big-endian order.

Implicit conversions between numeric types

Table 1–2 shows the implicit conversions supported by NPL when assigning a value of one numeric type to the value of another numeric type. These implicit conversions can occur in an assignment statement or when passing a value to an NPL built-in function.

Table 1–2 Implicit Numeric Type Conversions

To Type	From Type
Short	Byte, Short
Integer	Byte, Short, Integer, TimeInSecs
TimeInSecs	Byte, Short, Integer, TimeInSecs

Table 1–2 (Cont.) Implicit Numeric Type Conversions

To Type	From Type
Long	Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs
TimeInMilliSecs	Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs
Float	Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs, Float
Double	Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs, Float, Double

Literals

The Node Programming Language provides the capability to specify integer, floating-point and string literals. The syntax of these literals are defined in the `IntLiteralExpression, FloatingPointLiteralExpression and StringLiteralExpression` productions in "EBNF for NPL".

Integer literals can be expressed as decimal, octal or hexadecimal values and have a range from -9223372036854775808 to 9223372036854775807 inclusive.

Floating-point literals have an approximate range of ±1.7976931348623157E+308.

A string literal consists of zero or more characters enclosed in double quotes. A character may be represented by an escape sequence for the representation of some non-graphic characters as well as the single quote, double quote and backslash characters.

Variables

NPL supports three types of variables: local variables, input record variables and output record variables. A variable must be declared (i.e. given a type and identifier) before being used in an NPL program. Variable names must be unique within an NPL program. All characters in the name of a variable are significant and case is also significant. The length of a variable name is essentially unlimited. NPL reserved words (see "NPL reserved words") cannot be used as variable names.

Local variables

As the name implies, local variables are variables local to the NPL program in which they are declared. A local variable declaration consists of the type followed by the variable's identifier, an optional initialization assignment, and ending with a semicolon:

```
type identifier [= initialization-expression] ;
```

The type of a local variable is one of those listed in Table 1–1 (excluding `InputRec` and `OutputRec`). A local variable's `identifier` must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. The optional *`initialization-expression`* includes expressions such as literals, other variables that have already been initialized and arithmetic expressions. A local variable's *`initialization-expression`* must evaluate to the data type of the variable. See "Expressions" for a more complete description of expressions in NPL.

Obfuscatory Local Variable Information

This section contains information about local variables that may not be generally intuitive.

A local variable behaves like a static or global variable in other languages in that it retains the value it was last assigned. What this means from an NPL execution standpoint is, a local variable given a value during one pass through the NPL program's statements via a call to an `NPLFieldProcessor` object's `processData` method, will have the last value it was assigned the next time the NPL program is entered through a `processData` call. If a local variable is assigned a value by an *`initialization-expression`*, it will have that value on the first call to `processData`. If a local variable is given a value by an *`initialization-expression`* and that value is not changed within the NPL program statements, that local variable will have the same value for every execution of the NPL statements.

The following NPL built-in functions should not be used to initialize a local variable:

- substr bytes2TimeInSecs
- bytes2byte time2str
- str2byte str2TimeInSecs
- bytes2short bytes2TimeInMilliSecs
- str2short str2TimeInMilliSecs
- convertDateTime bytes2int
- IP2bytes str2int
- bytes2IP bytes2long
- IP2str str2long
- str2IP bytes2float

- randomInt str2float
- randomDouble bytes2double
- subbytes str2double
- copyBits

Input record variables

An input record variable is used to represent the data record passed into an NPL program as a result of an `NPLFieldProcessor` object's `processData` method being called. It is the NPL representation of the `DCFieldContainer` parameter to the `processData` method. An input record variable declaration consists of the keyword `InputRec` followed by one or more attribute declarations enclosed within a single set of braces, followed by the variable's identifier and ending with a semicolon:

```
InputRec { (attribute-declaration)+ } identifier ;
```

where an *`attribute-declaration`* is the attribute data type followed by the attribute identifier and ending in a semicolon:

```
attribute-type attribute-identifier ;
```

An input record variable's identifier must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. The *`attribute-type`* of an attribute declared in an input record is one of those listed in Table 1–1 (excluding `InputRec` and `OutputRec` of course). The attribute-identifier has two different forms:

- A sequence of characters beginning with a letter, underscore or colon followed by a sequence of zero or more letters, digits, underscores or hyphens.
- An integer literal in the range from 0 to 2147483647 inclusive.

Either form of *`attribute-identifier`* can be used by A CC. An EP or DC should use the second form of *`attribute-identifier`* since the input to these node types is in the NAR format, which uses integers for attribute identifiers. All the attribute identifiers within an InputRec declaration should be unique.

As mentioned above, an input record declaration is the NPL representation of the `DCFieldContainer` object being input into the NPL program for processing. As such the attribute types and identifiers declared in an NPL input record should match those of the incoming `DCFieldContainer` object.

The individual fields of an input record are accessed as `identifier.attribute-identifier`. For example, in.10000 would be used to access attribute 10000 of input record in.

Obfuscatory input record information

This section contains information about input record variables that may not be generally intuitive.

An input record declaration only needs to contain the attributes that are accessed individually in the NPL program.

However, an input record declaration must contain at least one attribute declaration.

An input record declaration may contain a declaration for an attribute that does not exist in the input record currently being processed. This feature is necessary to handle data streams that contain records that do not always contain a fixed set of attributes in all records passing through a Offline Mediation Controller system.

Another case where an input record declaration may contain a declaration for an attribute that does not exist in the incoming record is for input record enhancement. In this instance the incoming data record would probably never contain this attribute. Then if this attribute were assigned a value within the NPL program the input record would be enhanced by the addition of this new attribute that did not previously exist in the input data record. Only EP nodes can use input record enhancement where the output data contains the same attributes as the input data with the addition of new attributes and/or the modification of existing attribute values.

Output record variables

An output record variable is used to define the format of an output data record of an NPL program. Output records are used in an NPL program to contain the attributes that result from the translation of the data input to a node to the desired output format for a node. An output record variable in NPL represents the type a node provided as the outputDataType parameter when constructing its NPLFieldProcessor:

```
public NPLFieldProcessor( Class outputDataType, LoggerIfc inLogger,
              String scratchDir, String fileName )
  throws NodeProcessingException

public NPLFieldProcessor( Class outputDataType, LoggerIfc inLogger,
              String scratchDir, String filename,
              IDCMethodHandler, methodHandler )
  throws NodeProcessingException
```

This `outputDataType` Class object should implement the `DCFieldContainer` interface and have a zero argument constructor.

An output record variable declaration consists of the keyword `OutputRec` followed by one or more attribute declarations enclosed within a single set of braces, followed by the variable's identifier and ending with a semicolon:

`OutputRec { (`*`attribute-declaration`*`)+ } `*`identifier`*` ;`

where an *`attribute-declaration`* is the attribute data type followed by the attribute identifier and ending in a semicolon:

`attribute-type attribute-identifier ;`

An output record variable's identifier must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. The attribute-type of an attribute declared in an output record is one of those listed in Table 1–1 (excluding `InputRec` and `OutputRec` of course). The *`attribute-identifier`* has two different forms:

- A sequence of characters beginning with a letter, underscore or colon followed by a sequence of zero or more letters, digits, underscores or hyphens.
- An integer literal in the range from 0 to 2147483647 inclusive.

Either form of *`attribute-identifier`* can be used by an DC. A CC or EP should use the second form of *`attribute-identifier`* since the output from these node types is in the NAR format, which uses integers for attribute identifiers. All the attribute identifiers within an `OutputRec` declaration should be unique.

The individual fields of an output record are accessed as *`identifier.attribute-identifier`*. For example, `out.10000` would be used to access attribute 10000 of input record out.

Obfuscatory output record information

This section contains information about output record variables that may not be generally intuitive.

An attribute does not actually exist in an output record until it is assigned a value. What this means is the actual output data record will not contain an attribute unless it has been assigned a value within the NPL program statements.

Expose clause

An expose clause is used to make a subset of the input or output attributes available to the node for a specific reason. An expose clause declaration consists of the keywords Expose for, followed by a reason identifier, followed by one or more expose attributes enclosed within a single set of braces:

```
Expose for reason-identifier { (expose-attribute)+ } ;
```

where an expose-attribute is the attribute identifier followed by an optional user-friendly tag to be associated with the attribute:

```
record-identifier.attribute-identifier string-literal ;
```

An expose clause's *reason-identifier* must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. The *reason-identifier* must be the identifier of an input or output record already declared, and the *attribute-identifier* must be an attribute declared in that input or output record.

Sets of exposed attributes are available to the node via the `NPLFieldProcessor`'s `getExposedFields` method. To use this method, a node provides a reason identifier to indicate the expose clause of interest. The method will return an `ArrayList` of `FieldDescriptor` objects, one for each attribute in the corresponding expose clause. A `FieldDescriptor` object contains the attribute's type, identifier, and tag.

One *reason-identifier*, `Routing`, is already pre-defined for all nodes. By declaring an expose clause for routing (i.e. `Expose for Routing`), a node makes those attributes available for modulus and directed routing. See the appropriate CDK documentation for a more detailed description of routing.

Configuration clause

A configuration clause is used to set configuration values that may be needed by the node. There can only be one configuration clause in an NPL file. A configuration clause declaration consists of the keyword `Config`, followed by one or more configuration settings enclosed within a single set of braces:

```
Config { (config-attribute)+ } ;
```

where a *config-attribute* is a configuration key followed by a configuration value defined as a string:

```
configuration-key string-literal ;
```

A configuration clause's `configuration-key` must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores.

Configuration settings are available to the node via the `NPLFieldProcessor`'s `getConfigValue` method. To use this method, a node provides a configuration key to

indicate the configuration setting of interest. The method will return a string representing the value that matches the key provided.

Operators

NPL supports a subset of the usual operators found in most programming languages as identified in the following sections.

Arithmetic operators

The usual arithmetic operators + - * / are used in NPL for addition, subtraction, multiplication and division respectively. These arithmetic operations can be performed on any combination of Byte, Short, Integer, TimeInSecs, TimeInMilliSecs, Long, Float and Double expressions. The + and - operators have the same precedence. The * and / operators have the same precedence, but have a higher precedence than the + and - operators. Operators with the same precedence are evaluated from left to right. As in all programming languages, it is good practice to use parenthesis to indicate the order of evaluation for complex arithmetic expressions.

NPL also supports the unary + and - operators. These unary operators have a higher precedence than the addition, subtraction, multiplication and division arithmetic operators. The unary operators are only valid with the Byte, Short, Integer, Long, Float and Double data types.

Relational operators

NPL has a full complement of relational operators for use within conditional statements. These relational operators are: == (equality), != (inequality), < (less than), > (greater than), <= (less than or equal) and >= (greater than or equal).

Any numeric type (Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs, Float, or Double) expression can be compared to any other numeric type expression. In addition, NPL supports IP comparisons, MAC comparisons, IPv6 comparisons and String comparisons. Note that String comparisons are lexicographic and case sensitive.

Logical operators

NPL provides the && (logical and) and || (logical or) operators to be used in combination with relational expressions in conditional statements.

String concatenation operator

NPL uses the + sign as the string concatenation operator.

Explicit cast operator

NPL supports explicit casting among the numeric types: Byte, Short, Integer, TimeInSecs, Long, TimeInMilliSecs, Float and Double. The syntax for casting is similar to that of Java or C/C++; the target type in parentheses, followed by an expression that evaluates to one of the numeric types. Explicit cast conversions may result in a loss of information. Casting from a floating-point value to an integer value discards the fractional part (does not round). If you try to cast a value of one type to another that is out of range for the target type, the result will be truncated and have a different value.

Built-in functions

This section provides a brief synopsis of the built-in functions provided by NPL. Built-in functions are provided for passing a data record from NPL to the `NPLFieldProcessor` object's data receiver, writing messages to the log file, various string utilities, converting a string representation of a value to its value and vice versa, and for copying bits from one variable to another.

Note: The InputRec and OutputRec variables are passed by reference to built-in functions while all other types are passed by value.

The write built-in function is the most often used built-in function. This overloaded function is used to pass either an `InputRec` or `OutputRec` variable that has been processed by NPL program statements back to a node via the `NPLFieldProcessor` object's data receiver `processData` method.

void for (Integer i1, Integer i2)

Creates a simple looping mechanism. Code within the block executes i2-i1 times.

Parameters

Table 1–3 Void for (Integer i1, Integer i2) Parameters

Parameter	Description	Acceptable Values
i1	initial loop counter	integer value
i2	last loop counter value	integer value

Return Value

None.

Error Conditions

None.

Example

```
Integer startVal;
Integer endVal;
Integer internalVal;

startVal = 1;
endVal = 10;
internalVal = startVal;

for (startVal, endVal)
{
logInfo("Outputting a record");
write(cdr);
internalVal = internalVal + 1;
}
```

write(OutputRec rec)

This `write` function is used to pass the given output record to the `processData` method of the `NPLFieldProcessor` object's data receiver. Depending on the data receiver, in most cases calling this write function results in the given output record being output by the node.

Note that the output record will only contain attributes that it was assigned during execution of the NPL program statements. If an attribute was declared in the output record declaration but not assigned a value during the execution of the NPL program statements, this attribute will not be present in the record being passed to the `NPLFieldProcessor` object's data receiver.

Parameters

Table 1–4 Write(OutputRec rec) Parameters

Parameter	Description	Acceptable Values
Rec	output record to return to the field processor	an output record declared in the NPL program

Return Value

None.

Error Conditions

None.

Example

```
InputRec {
 Integer 10004;
} in;
OutputRec {
 Integer 10004;
} out;
out.10004 = in.10004;
write( out ); // "write" the output record to the field processor
```

write(InputRec rec)

This write function is used to pass the given input record to the `processData` method of the `NPLFieldProcessor` object's data receiver. Depending on the data receiver, in most cases calling this write function results in the given input record being output by the node.

"Obfuscatory input record information" states that an input record declaration need only contain the attributes accessed by the NPL program statements. Therefore more attributes may actually be present in an input record than what is declared in NPL. Such an input record passed to this write method will contain all the attributes of the actual input record, not just those declared in the NPL representation of the input record.

Parameters

Table 1–5 Write(InputRec rec) Parameters

Parameter	Description	Acceptable Values
rec	input record to return to the field processor	an input record declared in the NPL program

Return Value

None.

Error Conditions

None.

Example

```
Integer counter = -1;
InputRec {
 Integer 10004;
} in;
if( counter < 0 ) {
 counter = 0;
}
else {
 counter = counter + 1;
}
in.10004 = counter;
write( in ); // "write" the modified input record to the field processor
```

InputRec clone(InputRec rec)

Returns a clone of the given input record.

> **Note:** The clone function is only supported for records in the NAR format.

Parameters

Table 1–6 InputRec clone(InputRec rec) Parameters

Parameter	Description	Acceptable Values
rec	input record to clone	an input record declared in the NPL program that is in NAR format

Return Value

Returns an InputRec that is a clone of the given input record.

Error Conditions

If the record that is passed to the clone function is not in NAR format, the statement that contains the function call is skipped and execution will continue with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

Example

```
InputRec {
 Integer 10015;
 Long  10017;
} in;
in.10017 = 0;
write( clone( in ) ); // Clone record. It is passed by reference to write.
in.10017 = 1;
write( in );
```

The clone function is used here because records are passed by reference to built-in functions. If the clone function were not used, the second assignment to attribute 10017 would change the value of that attribute from the first write call.

OutputRec clone(InputRec rec)

Returns a clone of the given input record.

Note: The clone function is only supported for records in the NAR format.

Parameters

Table 1–7 OutputRec clone(InputRec rec) Parameters

Parameter	Description	Acceptable Values
rec	input record to clone	an input record declared in the NPL program that is in NAR format

Return Value

Returns an OutputRec that is a clone of the given input record.

Error Conditions

If the record that is passed to the clone function is not in NAR format, the statement that contains the function call is skipped and execution will continue with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

Example

```
InputRec {
 Integer 10015;
 Long  10017;
} in;
OutputRec {
 Long 10017;
} out;
out = clone( in );
if( out.10017 != 0 ) {
 out.10017 = 0;
}
else {
 out.10017 = 1;
}
write( out );
```

Note: Using record-to-record assignment (see "Assignment Statement") is more efficient than the way in which clone is used in this example.

OutputRec clone(OutputRec rec)

Returns a clone of the given output record.

Note: The clone function is only supported for records in the NAR format.

Parameters

Table 1–8 OutputRec clone(OutputRec rec) Parameters

Parameter	Description	Acceptable Values
rec	output record to clone	an output record declared in the NPL program that is in NAR format

Return Value

None.

Error Conditions

If the record that is passed to the clone function is not in NAR format, the statement that contains the function call is skipped and execution will continue with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

Example

```
Integer NARcounter = -1;
InputRec {
 Integer 30026;
 Integer 30027;
} in;
OutputRec {
 Integer 10004;
 String 30036;
} out;
if( NARcounter < 0 ) {
 NARcounter = 0;
}
else {
 NARcounter = NARcounter + 1;
}
out.10004 = NARcounter;
out.30036 = int2str( in.30026 );
write( clone( out ) ); // Clone record. It is passed by reference to write.
if( NARcounter < 0 ) {
 NARcounter = 0;
}
else {
 NARcounter = NARcounter + 1;
```

```
}
out.10004 = NARcounter;
out.30036 = int2str( in.30027 );
write( out );
```

The clone function is used here because records are passed by reference to built-in functions. If the clone function were not used, the second assignment to attribute 30036 would change the value of that attribute from the first write call.

InputRec clone(OutputRec rec)

Returns a clone of the given output record.

Parameters

Table 1–9 InputRec clone(OutputRec rec) Parameters

Parameter	Description	Acceptable Values
TBD	TBD	TBD

Return Value

Returns an OutputRec that is a clone of the given output record.

Error Conditions

None.

Example

```
InputRec {
```

sample(OutputRec rec)

Returns a clone of the given output record.

Note: The clone function is only supported for records in the NAR format.

Parameters

Table 1–10 sample(OutputRec rec) Parameters

Parameter	Description	Acceptable Values
Rec	output record to clone	an output record declared in the NPL program that is in NAR format

Return Value

Returns an InputRec that is a clone of the given output record.

Error Conditions

If the record that is passed to the clone function is not in NAR format, the statement that contains the function call is skipped and execution will continue with the next statement in the NPL program. A warning of this condition will be written to the

node's log file if the node's configured debug level is set at the logging warnings level.

Example

No example is provided because this form of clone is only provided for completeness and is very unlikely to be used.

logError(String message)

Writes the error message to the node's log file if the node's configured debug level is set at the error logging level. An error is indicated with a red alarm on the Administration Client GUI.

Parameters

Table 1–11 logError(String message) Parameters

Parameter	Description	Acceptable Values
message	string expression representing the error message to write to the node's log file	string expression consisting of one or more non-null strings

A null string in the error message string expression results in the word “null” being written to the log file at the place where that string would appear in the message. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

Return Value

None.

Error Conditions

If a non-existent input or output attribute is encountered in the error message string expression, the logError call is skipped and execution continues with the next statement in the NPL program.

Example

```
Integer REC_TYPE = 23;
InputRec {
 Integer 10015;
} in;
if( in.10015 != REC_TYPE ) {
 logError( "Encountered invalid record type " + int2str( in.10015 ) );
}
else {
 write( in );
}
```

Assuming in.10015 equals 17 and the node's configured debug level is set at the error logging level, then a message similar to the following would be written to the node's log file:

```
Jan 5, 2003 10:47:56 AM Error: Encountered invalid record type 17
```

logWarning(String message)

Writes the warning message to the node's log file if the node's configured debug level is set at the logging warnings level. A warning is indicated by a blue (cyan) alarm on the Administration Client GUI.

Parameters

Table 1–12 logWarning(String message) Parameters

Parameter	Description	Acceptable Values
message	string expression representing the warning message to write to the node's log file	string expression consisting of one or more non-null strings

A null string in the warning message string expression results in the word “null” being written to the log file at the place where that string would appear in the message. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

Return Value

None.

Error Conditions

If a non-existent input or output attribute is encountered in the warning message string expression, the logWarning call is skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String DATE;
 String TIME;
 String BINTYPE;
} in;
OutputRec {
 String 70000;
 String 70001;
 String 70002;
} out;
out.70000 = in.DATE;
out.70001 = in.TIME;
if( in.BINTYPE != "15m" && in.BINTYPE != "24h" ) {
 logWarning("Incorrect bin value, supported values are \"15m\", " +
       "\"24h\". Continuing to process data without this field.");
}
else {
 out.70002 = in.BINTYPE;
}
write( out );
```

Assuming in.BINTYPE does not equal “15m” or “24h” and the node's configured debug level is set at the logging warnings level, then a message similar to the following would be written to the node's log file:

```
Jan 7, 2003 3:31:04 PM Warning: Incorrect bin value, supported values are "15m",
"24h". Continuing to process data without this field.
```

logWarning(String faultCategory, String specificFault, String additionalFaultText)

Writes the warning message to the node's log file if the node's configured debug level is set at the logging warnings level. A warning is indicated with a blue alarm on the Administration Client GUI.

Parameters

Table 1–13 logWarning(String faultCategory, String specificFault, String additionalFaultText) Parameters

Parameter	Description	Acceptable Values
faultCategory	general category for the error	Initialization, Configuration, Connectivity, Invalid data, Resources, Internal Error, Node Control
specificFault	clearly worded description of the fault condition, with no variable component	string expression consisting of one or more non-null strings
additionalFault Text	any additional fault text (the variable information)	string expression consisting of one or more non-null strings

A null string in the `specificFault` or `additionalFaultText` string expressions results in the word "null" being written to the log file at the place where that string would appear in the message. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

Return Value

None.

Error Conditions

None.

Example

If a non-existent input or output attribute is encountered in any of the parameter string expressions, the `logWarning` call is skipped and execution continues with the next statement in the NPL program.

```
InputRec {
 String DATE;
 String TIME;
 String BINTYPE;
} in;
OutputRec {
 String 70000;
 String 70001;
 String 70002;
} out;
out.70000 = in.DATE;
out.70001 = in.TIME;
if( in.BINTYPE != "15m" && in.BINTYPE != "24h" ) {
 logWarning("Invalid Data","Incorrect field value","Incorrect bin value, supported
values are \"15m\", " + "\"24h\". Continuing to process data without field.");
}
```

```
else {
 out.70002 = in.BINTYPE;
}
write( out );
```

Assuming `in.BINTYPE` does not equal “15m” or “24h” and the node's configured debug level is set at the logging warnings level, then a message similar to the following would be written to the node's log file:

```
2002-03-21 14:16:56; Warning; Invalid Data; Incorrect field value; Incorrect bin
value, supported values are "15m", "24h". Continuing to process data without
field.
```

logInfo(String message)

Writes the informational message to the node's log file if the node's configured debug level is set at the error logging level. No alarm is generated.

Parameters

Table 1–14 logInfo(String message) Parameters

Parameter	Description	Acceptable Values
message	string expression representing the informational message to write to the node's log file	string expression consisting of one or more non-null strings

A null string in the informational message string expression results in the word “null” being written to the log file at the place where that string would appear in the message. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

Return Value

None.

Error Conditions

If a non-existent input or output attribute is encountered in the informational message string expression, the logInfo call is skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String DATE;
 String TIME;
 String Layer;
} in;
OutputRec {
 String 70000;
 String 70001;
 String 70017;
} out;
if( in.Layer != "OPT_TS" &&
  in.Layer != "OPT_MS" &&
  in.Layer != "OPT_CH" )
{
```

```
 out.70000 = in.DATE;
 out.70001 = in.TIME;
 out.70017 = in.Layer;
write( out );
}
else
{
 logInfo("The record with Layer = " + in.Layer +
     " has not been processed!");
}
```

Assuming in.Layer equals “XYZ” and the node's configured debug level is set at the error logging level, then a message similar to the following would be written to the node's log file:

```
Jan 27, 2001 5:31:04 AM Informational: The record with Layer = XYZ has not been
processed!
```

logTrace(String message)

Writes the trace message to the node's log file if the node's configured debug level is set at the trace logging level. No alarm is generated.

Parameters

Table 1–15 logTrace(String message) Parameters

Parameter	Description	Acceptable Values
message	string expression representing the trace message to write to the node's log file	string expression consisting of one or more non-null strings

A null string in the trace message string expression results in the word “null” being written to the log file at the place where that string would appear in the message. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

Return Value

None.

Error Conditions

If a non-existent input or output attribute is encountered in the trace message string expression, the logTrace call is skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String 70019;
 Integer 70029;
} in;
OutputRec {
 String 70023;
} out;
logTrace("Status is " + in.70019 );
out.70023 = " ";
if( in.70019 == "U" ) {
```

```
 out.70023 = out.70023 + int2str( in.70029 );
}
write( out );
```

Assuming that in.70019 equals “U” and the node's configured debug level is set at the trace logging level, then a message similar to the following would be written to the node's log file:

```
Jan 11, 2001 5:01:14 AM Informational: Status is U
```

Integer strlen(String s)

Returns the length of String s.

Parameters

Table 1–16 Integer strlen(String s) Parameters

Parameter	Description	Acceptable Values
s	string to get the length of	string expression consisting of one or more non-null strings

Return Value

Returns an Integer representing the number of characters in the string.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in the string expression passed to the strlen function, the statement containing the function call is skipped and execution continues with the next statement in the NPL program.

Example

```
Integer dpos;
Integer len;
InputRec {
 String SOFTVER;
} csg;
OutputRec {
 String 50201;
} call;
//
// Remove trailing characters after .A or .E
//
dpos = -1;
dpos = strpos( csg.SOFTVER, ".A", 1);
len = 0;
len = strlen( csg.SOFTVER );
if( dpos > 0 ) {
 len = dpos + 2;
```

```
}
else {
 dpos = strpos( csg.SOFTVER, ".E", 1);
 if( dpos > 0 ) {
  len = dpos + 2;
 }
}
call.50201 = substr( csg.SOFTVER, 0, len );
write( call );
```

Integer strpos(String string, String substring, Integer nth_occurrence)

Returns the index of the nth_occurrence of substring in string, or -1 if there is not a nth_occurrence of substring in string.

Parameters

Table 1–17 Integer strpos(String string, String substring, Integer nth_occurrence) Parameters

Parameter	Description	Acceptable Values
string	string to search in	string expression consisting of one or more non-null strings
substring	sub-string to search for	string expression consisting of one or more non-null strings
nth_occurrence	occurrence to search for	integer expression with value greater than zero

Return Value

Possible Integer return values are:

- -1 if there is not an nth occurrence of substring in string, or if nth_occurrence < 1
- index in the range from 0 to string's length minus 1 if there is an nth occurrence of substring in string nth_occurrence - 1 if substring is the empty ("") string

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in any of the parameter expressions passed to the strpos function, the statement that contains the function call is skipped and execution continues with the next statement in the NPL program.

Example

```
Integer len;
Integer dot1;
Integer dot2;
Integer dot3;
String ipStr;
Long  ipLong;
```

```
Long  addr;

InputRec {
 IP 70003;
} in;
OutputRec {
 Long ControllerIPAddr;
} out;
out.ControllerIPAddr = 0;
//
// Convert IP string in dot notation to a long value.
//
dot1 = -1;
dot2 = -1;
dot3 = -1;
len = -1;
ipStr = "";
ipStr = IP2str( in.70003 );
len = strlen( ipStr );
if( len >= 7 ) {
 dot1 = strpos( ipStr, ".", 1);
 if( dot1 != -1 ) {
  dot2 = strpos( ipStr, ".", 2);
  if( dot2 != -1 ) {
   dot3 = strpos( ipStr, ".", 3);
  }
 }
}
if( (len >= 7 ) && (dot1 != -1) && (dot2 != -1) && (dot3 != -1) ) {
 addr = -1;
 addr = str2long( substr( ipStr, 0, dot1 ) );
 if( addr >= 0 ) {
  ipLong = 16777216 * addr;
  addr = -1;
  addr = str2long( substr( ipStr, dot1 + 1, dot2 ) );
  if( addr >= 0 ) {
   ipLong = ipLong + (65536 * addr);
   addr = -1;
   addr = str2long( substr( ipStr, dot2 + 1, dot3 ) );
   if( addr >= 0 ) {
    ipLong = ipLong + (256 * addr);
    addr = -1;
    addr = str2long( substr( ipStr, dot3 + 1, len ) );
    if( addr >= 0 ) {
     out.ControllerIPAddr = ipLong + addr;
    }
    else {
     logError( "Invalid IP address (ControllerIPAddr) " + ipStr );
    }
   }
   else {
    logError( "Invalid IP address (ControllerIPAddr) " + ipStr );
   }
  }
  else {
   logError( "Invalid IP address (ControllerIPAddr) " + ipStr );
  }
 }
 else {
  logError( "Invalid IP address (ControllerIPAddr) " + ipStr );
```

```
 }
}
else {
 logError( "Invalid IP address (ControllerIPAddr) " + ipStr );
}
write( out );
```

String substr(String s, Integer beginIndex, Integer endIndex)

Returns the substring of s that begins at the specified beginIndex and extends to the character at index `endIndex` - 1.

Parameters

Table 1–18 String substr(String s, Integer beginIndex, Integer endIndex) Parameters

Parameter	Description	Acceptable Values
s	string to get sub-string of	string expression consisting of one or more non-null strings
beginIndex	beginning index, inclusive	integer expression with value >= 0 and <= end index
endIndex	ending index, exclusive	integer expression with value >= begin index and <= string length

Return Value

Returns the sub-string as a String.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If either the begin index or end index expression does not evaluate to an acceptable value as defined in Section 0, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning indicating the invalid expression value will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in any of the parameter expressions passed to the substr function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
Integer dpos;
Integer len;
InputRec {
 String SOFTVER;
} csg;
OutputRec {
 String 50201;
} call;
```

```
//
// Remove trailing characters after .A or .E
//
dpos = -1;
dpos = strpos( csg.SOFTVER, ".A", 1);
len = 0;
len = strlen( csg.SOFTVER );
if( dpos > 0 ) {
 len = dpos + 2;
}
else {
 dpos = strpos( csg.SOFTVER, ".E", 1);
 if( dpos > 0 ) {
  len = dpos + 2;
 }
}
call.50201 = substr( csg.SOFTVER, 0, len );
write( call );
```

String str2lower(String s)

Returns a string representing the string s converted to lower case.

Parameters

Table 1–19 String str2lower(String s) Parameters

Parameter	Description	Acceptable Values
s	string from which to get the lower case string	string expression consisting of one or more non-null strings

Return Value

Returns a String representing the string expression converted to lower case.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in the string expression passed to the `str2lower` function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String 11007;
} in;
in.11007 = str2lower( in.11007 );
write( in );
```

String str2upper(String s)

Returns a string representing the string s converted to upper case.

Parameters

Table 1–20 String str2upper(String s) Parameters

Parameter	Description	Acceptable Values
s	string from which to get the upper case string	string expression consisting of one or more non-null strings

Return Value

Returns a String representing the string expression converted to upper case.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in the string expression passed to the str2upper function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Long  10017;
 String 20104;
} in;
if( str2upper(in.20104) == "LOCAL" ) { // Perform case insensitive compare
 in.10017 = 0;
}
else {
 in.10017 = 1;
}
write( in );
```

Bytes str2bytes(String s)

Convert the String s to its byte array representation according to the platform's default character encoding. The length of the array of bytes is a function of the encoding, and thus may not be equal to the length of the string.

Parameters

Table 1–21 Bytes str2bytes(String s) Parameters

Parameter	Description	Acceptable Values
s	string from which to get the byte array representation	string expression consisting of one or more non-null strings

Return Value

Returns a Bytes value representing the string expression converted to its byte array representation according to the platform's default character encoding. An empty ("") string is represented as a zero length byte array.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in the string expression passed to the str2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String 110000;
} in;
OutputRec {
 Bytes 111000;
} out;
out.111000 = str2bytes( in.110000 );
write( out );
```

Bytes str2bytes(String s, String e)

Convert the String s to its byte array representation using the specified character encoding. The character encoding is that which is supported by Java.

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Character.html

The length of the array of bytes is a function of the encoding, and thus may not be equal to the length of the string.

Parameters

Table 1–22 Bytes str2bytes(String s, String e) Parameters

Parameter	Description	Acceptable Values
s	string from which to get the byte array representation	string expression consisting of one or more non-null strings
e	string indicating the character encoding to use	character encoding supported by Java (see link above)

Return Value

Returns a Bytes value representing the string expression converted to its byte array representation according to the platform's default character encoding. An empty ("") string is represented as a zero length byte array.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output attribute is encountered in the string expression passed to the str2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String 110000;
} in;
OutputRec {
 Bytes 111000;
} out;
out.111000 = str2bytes( in.110000, "US-ASCII" );
write( out );
```

String bytes2hexstr(Bytes b)

Returns the value of the specified array of bytes as a string of space-separated hexadecimal values, where each byte corresponds to one value.

Parameters

Table 1–23 String bytes2hexstr(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a string of space-separated hexadecimal values	non-null byte array

Return Value

Returns the String value representing the given byte array as a series of space-separated hexadecimal values, where each byte corresponds to one value. For example, for a byte array with values (116, 115, 0, 3, 117) the returned string will be "74 73 00 03 75 ".

A byte array of length zero returns the empty string ("").

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output Bytes attribute is passed to the bytes2hexstr function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 20206;
} in;
OutputRec {
 String ServiceCentre;
} out;
out.ServiceCentre = bytes2hexstr( in.20206 );
write( out );
```

String bytes2str(Bytes b)

Returns the value of the specified array of bytes b as a string, converting the bytes using the platform's default character encoding. The length of the string is a function of the encoding, and thus may not be equal to the length of the array of bytes.

Parameters

Table 1–24 String bytes2str(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a string	non-null byte array

Return Value

Returns the String resulting from the conversion of the specified byte array using the platform's default character encoding.

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output Bytes attribute is passed to the bytes2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 1;
} in;
OutputRec {
 String 10005;
} out;
out.10005 = bytes2str( in.1 );
write( out );
```

As an example, if the in.1 byte array has the value (116, 101, 115, 116), then out.10005 will contain the string “test”.

String bytes2str(Bytes b, String e)

Returns the value of the specified array of bytes b as a string, converting the bytes using the specified character encoding. The character encoding is that which is supported by Java.

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Character.html

The length of the string is a function of the encoding, and thus may not be equal to the length of the array of bytes.

Parameters

Table 1–25 String bytes2str(Bytes b, String e) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a string	non-null byte array
e	string indicating the character encoding to use	character encoding supported by Java (see link above)

Return Value

Returns the resulting from the conversion of the specified byte array using the platform's default character encoding.

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If a non-existent input or output Bytes attribute is passed to the bytes2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 1;
} in;
OutputRec {
 String 10005;
} out;
out.10005 = bytes2str( in.1, "ISO-8859-1" );
write( out );
```

As an example, if the in.1 byte array has the value (116, 101, 115, 116), then out.10005 will contain the string "test".

Bytes byte2bytes(Byte b)

Returns the array of bytes representation for the Byte b, essentially an array of 1 byte.

Parameters

Table 1–26 Bytes byte2bytes(Byte b) Parameters

Parameter	Description	Acceptable Values
b	byte to represent as an array of bytes	a byte expression

Return Value

Returns a Bytes value that is an array of 1 byte.

Error Conditions

If a non-existent input or output attribute is encountered in the byte expression passed to the byte2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Byte 123456;
} in;
OutputRec {
 Bytes 654321;
} out;
out.654321 = byte2bytes( -in.123456 );
write( out );
```

Byte bytes2byte(Bytes b)

Returns the value of the array of bytes b as a Byte. The array of bytes b must contain only one byte.

Parameters

Table 1–27 Byte bytes2byte(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to return as a Byte	non-null byte array containing one byte

Return Value

Returns the Byte value of the only byte in the array.

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2byte function is not one byte in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to

the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2byte function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 27;
} in;
OutputRec {
 Byte 30004;
} out;
out.30004 = -1;
out.30004 = bytes2byte( in.27 );
write( out );
```

String byte2str(Byte b)

Returns a string representing the specified Byte b. The radix is assumed to be 10.

Parameters

Table 1–28 String byte2str(Byte b) Parameters

Parameter	Description	Acceptable Values
b	byte value to represent as a string	a byte expression

Return Value

Returns the String representing the specified byte expression in radix 10.

Error Conditions

If a non-existent input or output attribute is encountered in the byte expression passed to the byte2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Byte 30005;
} in;
OutputRec {
 String Source-IP-Prefix-Mask;
} out;
out.Source-IP-Prefix-Mask = "";
out.Source-IP-Prefix-Mask = byte2str( in.30005 );
write( out );
```

Byte str2byte(String s)

Returns the Byte value represented by the specified string. Accepts decimal, hexadecimal, and octal numbers in the following formats.

Table 1–29 Byte str2byte(String s) Formats

Numeric type	Acceptable formats	Valid range
Decimal	("-")? ["1"-"9"] (["0"-"9"])*	"-128" to "+127"
Hexadecimal	("-")? "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+	"0x80" to "0x7F"
Octal	("-")? "0" (["0"-"7"])*	"0200" to "-0177"

Parameters

Table 1–30 Byte str2byte(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing a byte value	String expression consisting of one or more non-null strings representing a decimal, hexadecimal, or octal number. See the table above for valid formats and ranges.

Return Value

Returns the Byte value represented by the given string.

The following table illustrates some example strings and resulting byte values.

Table 1–31 Example Strings and Resulting Byte Values

String value	Byte value
"-21"	-21
"123"	123
"128"	none - bad string value
"-0X13"	-19
"0x7f"	127
"-0X81"	none - bad string value
"017"	15
"-020"	-16
"0400"	none - bad string value

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2byte function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2byte function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String PREPAID;
} in;
OutputRec {
 Byte 21005;
} out;
out.21005 = 0;
out.21005 = str2byte( in.PREPAID );
write( out );
```

Bytes short2bytes(Short s)

Returns the array of bytes representation for the specified Short (16-bit integer type) value s. The byte order is big-endian.

Parameters

Table 1–32 Bytes short2bytes(Short s) Parameters

Parameter	Description	Acceptable Values
s	Short value to represent as an array of bytes	a short expression

Return Value

Returns a Bytes value corresponding to the big-endian byte array representation of the given short expression.

Error Conditions

If a non-existent input or output attribute is encountered in the short expression passed to the short2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Short 111111;
} in;
OutputRec {
 Bytes 222222;
} out;
out.222222 = short2bytes( in.111111 );
write( out );
```

As an example, if in.111111 equals 4386 (0x1122), then the value of the out.222222 byte array will be (0x11, 0x22) where 0x11 is the value of the first element of the byte array.

Short bytes2short(Bytes b)

Returns the value of the array of bytes b in big-endian order as a Short (16-bit integer type). The length of the array of bytes may be less than the 2 bytes to store a Short

value, in which case the high order bytes will be padded with zeroes. However, the length of the array of bytes cannot be greater than 2.

Parameters

Table 1–33 Short bytes2short(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a Short value	non-null byte array containing one or two bytes

Return Value

Returns the Short value represented by the given byte array.

The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting short values.

Table 1–34 Short bytes2short(Bytes b) Return Values

Byte array	Short value
(127)	127
(-1)	255
(1,2)	258
(-1,1)	-255
(1,-1)	511
(-1,-1)	-1

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2short function is not one or two bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2short function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 14;
 Bytes 24;
} in;
OutputRec {
 Integer 30027;
 Short  10012;
} out;
```

```
out.30027 = bytes2short( in.14 );
out.10012 = bytes2short( in.24 );
write( out );
```

String short2str(Short s)

Returns a string representing the specified Short s. The radix is assumed to be 10.

Parameters

Table 1–35 String short2str(Short s) Parameters

Parameter	Description	Acceptable Values
s	short value to represent as a string	a short expression

Return Value

Returns the String representing the specified short expression in radix 10.

Error Conditions

If a non-existent input or output attribute is encountered in the short expression passed to the short2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Short 30010;
} in;
OutputRec {
 String Application-Protocol;
} out;
out.Application-Protocol = short2str( in.30010 );
write( out );
```

Short str2short(String s)

Returns the Short value represented by the specified string. Accepts decimal, hexadecimal, and octal numbers in the following formats.

Table 1–36 Short str2short(String s) Formats

Numeric type	Acceptable formats	Valid range
Decimal	("-")? ["1"-"9"] (["0"-"9"])*	"-32768" to "32767"
Hexadecimal	("-")? "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+	"-0x8000" to "0x7FFF"
Octal	("-")? "0" (["0"-"7"])*	"-0100000" to "077777"

Parameters

Table 1–37

Parameters	Description	Acceptable Values
s	string representing a short value	String expression consisting of one or more non-null strings representing a decimal. hexadecimal, or octal number. See the table above for valid formats and ranges.

Return Value

Returns the Short value represented by the given string.

The following table illustrates some example strings and resulting short values.

Table 1–38 Short str2short(String s) Return Value

String value	Short value
"-21"	-21
"123"	123
"32768"	none - bad string value
"-0X13"	-19
"0x7fff"	32767
"-0X8001"	none - bad string value
"017"	15
"-020"	-16
"-0100001"	none - bad string value

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2short function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2short function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String Dest-ToS;
} in;
OutputRec {
 Short 30024;
```

```
} out;
out.30024 = str2short( in.Dest-ToS );
write( out );
```

Bytes int2bytes(Integer i)

Returns the array of bytes representation for the specified Integer (32-bit integer type) value i. The byte order is big-endian.

Parameters

Table 1–39 Bytes int2bytes(Integer i) Parameters

Parameter	Description	Acceptable Values
i	Integer value to represent as an array of bytes	an integer expression

Return Value

Returns a Bytes value corresponding to the big-endian byte array representation of the given integer expression.

Error Conditions

If a non-existent input or output attribute is encountered in the integer expression passed to the int2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Integer 30035;
} in;
OutputRec {
 String SE:System-ID;
} out;
// Convert from an IP stored as an integer to a system ID.
out.SE:System-ID = IP2str( bytes2IP( int2bytes( in.30035 ) ) );
write( out );
```

Integer bytes2int(Bytes b)

Returns the value of the array of bytes b in big-endian order as an Integer (32-bit integer type). The length of the array of bytes may be less than the 4 bytes to store an Integer, in which case the high order bytes will be padded with zeroes. However, the length of the array of bytes cannot be greater than 4.

Parameters

Table 1–40 Integer bytes2int(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a Integer value	non-null byte array containing from one to four bytes

Return Value

Returns the Integer value represented by the given byte array.

The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting integer values.

Table 1–41 Integer bytes2int(Bytes b) Return Value

Byte array	Integer value
(127)	127
(-1)	255
(1,2)	258
(-1,1)	65281
(-1,-1)	65535
(1,0,2)	65538
(2,0,1)	131073
(1,0,0,2)	16777218
(2,0,0,1)	33554433

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2int function is not from one to four bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2int function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 40;
} in;
OutputRec {
 Integer 40040;
} out;
out.40040 = bytes2int( in.40 );
write( out );
```

String int2str(Integer i)

Returns a string representing the specified Integer i. The radix is assumed to be 10.

Parameters

Table 1–42 String int2str(Integer i) Parameters

Parameter	Description	Acceptable Values
i	integer value to represent as a string	an integer expression

Return Value

Returns the String representing the specified integer expression in radix 10.

Error Conditions

If a non-existent input or output attribute is encountered in the integer expression passed to the int2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Integer 43031;
} in;
OutputRec {
 String callingDse;
} out;
out.callingDse = int2str( in.43031 );
write( out );
```

Integer str2int(String s)

Returns the Integer value represented by the specified string. Accepts decimal, hexadecimal, and octal numbers in the following format:

Table 1–43 Integer str2int(String s) Formats

Numeric type	Acceptable formats	Valid range
Decimal	("-")? ["1"-"9"] (["0"-"9"])*	""-2147483648" to "2147483647"
Hexadecimal	("-")? "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+	"-0x80000000" to "0x7FFFFFFF"
Octal	("-")? "0" (["0"-"7"])*	"-020000000000" to "017777777777"

Parameters

Table 1–44 Integer str2int(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing an integer value	String expression consisting of one or more non-null strings representing a decimal, hexadecimal, or octal number. See the table above for valid formats and ranges.

Return Value

Returns the Integer value represented by the given string.

The following table illustrates some example strings and resulting integer values.

Table 1–45 Integer str2int(String s) Return Values

String value	Integer value
"-21"	-21
"123"	123
"-2147483649"	none - bad string value
"-0X13"	-19
"0x7FFFFFFF"	2147483647
"-0X80000001"	none - bad string value
"017"	15
"-020"	-16
"020000000000"	none - bad string value

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2int function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2int function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String recordIdentifier;
} in;
OutputRec {
 Integer 43004;
} out;
out.43004 = str2int( in.recordIdentifier );
write( out );
```

Integer randomInt(Integer min, Integer max)

Returns a pseudorandom Integer value in the range from min to max inclusive, where the value of min must be less than the value of max. Refer to Java's Random class for a description of the random number generator algorithm.

See http://docs.oracle.com/javase/1.3/docs/api/java/util/Random.html

Parameters

Table 1–46 Integer randomInt(Integer min, Integer max) Parameters

Parameter	Description	Acceptable Values
min	minimum value of inclusive range	an integer expression that is not greater than the maximum value
max	maximum value of inclusive range	an integer expression that is not less than the minimum value

Return Value

Returns a pseudorandom Integer value within the specified inclusive range.

Error Conditions

If the value of the minimum parameter is greater than or equal to the value of the maximum parameter, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in either one of the integer expressions passed to the randomInt function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String 10013;
 Long  10017;
} in;
in.10017 = randomInt( -5, 5 );
write( in );
```

Bytes long2bytes(Long l)

Returns the array of bytes representation for the specified Long (64-bit integer type) value l. The byte order is big-endian.

Parameters

Table 1–47 Bytes long2bytes(Long l) Parameters

Parameter	Description	Acceptable Values
l	Long value to represent as an array of bytes	a long expression

Return Value

Returns a Bytes value corresponding to the big-endian byte array representation of the given long expression.

Error Conditions

If a non-existent input or output attribute is encountered in the long expression passed to the long2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Long 120000;
 Bytes 220000;
} in;
in.220000 = long2bytes( in.120000 );
write( in );
```

Long bytes2long(Bytes b)

Returns the value of the array of bytes b in big-endian order as a Long (64-bit integer type). The length of the array of bytes may be less than the 8 bytes to store a Long value, in which case the high order bytes will be padded with zeroes. However, the length of the array of bytes cannot be greater than 8.

Parameters

Table 1–48 Long bytes2long(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a Long value	non-null byte array containing from one to eight bytes

Return Value

Returns the Long value represented by the given byte array.The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting long values.

Table 1–49 Long bytes2long(Bytes b) Return Value

String value	Long value
(-1)	255
(1, 2)	258
(2, 0, 1)	131073
(1, 0, 0, 2)	16777218
(1, 2, 3, 4, 5)	4328719365
(1, -1, -1, -1, -1, -1)	2199023255551
(1, 0, 1, 0, 1, 0, 1)	281479271743489
(2, 0, 0, 0, 0, 0, 0, 0)	144115188075855872
(0, 0, 0, 0, 0, 0, 0, 2)	2

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the

NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2long function is not from one to eight bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2long function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 42;
} in;
OutputRec {
 Long 10006;
} out;
out.10006 = bytes2long( in.42 );
write( out );
```

String long2str(Long l)

Returns a string representing the specified Long l. The radix is assumed to be 10.

Parameters

Table 1–50 String long2str(Long l) Parameters

Parameter	Description	Acceptable Values
l	long value to represent as a string	a long expression

Return Value

Returns the String representing the specified long expression in radix 10.

Error Conditions

If a non-existent input or output attribute is encountered in the long expression passed to the long2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Long 40001;
} in
OutputRec {
 String Timestamp-Rounded2Hrs;
} out;
out.Timestamp-Rounded2Hrs = long2str( in.40001 );
write( out );
```

Long str2long(String s)

Returns the Long value represented by the specified string. Accepts decimal, hexadecimal, and octal numbers in the following format.

Table 1–51 Long str2long(String s) Formats

Numeric type	Acceptable formats	Valid range
Decimal	("-")? ["1"-"9"] (["0"-"9"])*	"-9223372036854775808" to "9223372036854775807"
Hexadecimal	("-")? "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+	"-0x8000000000000000" to "0x7FFFFFFFFFFFFFFF"
Octal	("-")? "0" (["0"-"7"])*	"-0100000000000000000000" to"0777777777777777777777"

Parameters

Table 1–52 Long str2long(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing a long value	String expression consisting of one or more non-null strings representing a decimal, hexadecimal, or octal number. See the table above for valid formats and ranges.

Return Value

Returns the Long value represented by the given string.

The following table illustrates some example strings and resulting long values.

Table 1–53 Long str2long(String s) Return Values

String value	Long value
"-21"	-21
"123"	123
"9223372036854775808"	none - bad string value
"-0X13"	-19
"-0x8000000000000000"	-9223372036854775808
"0x8000000000000000"	none - bad string value
"017"	15
"-020"	-16
"01000000000000000000000"	none - bad string value

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2long function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2long function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String COOKIESIN;
} in;
OutputRec {
 Long 21015;
} out;
out.21015 = str2long( in.COOKIESIN );
write( out );
```

Bytes float2bytes(Float f)

Returns the array of bytes representation for the specified Float (IEEE 754 32-bit single-precision floating-point type) value f.

Parameters

Table 1–54 Bytes float2bytes(Float f) Parameters

Parameter	Description	Acceptable Values
f	Float value to represent as an array of bytes	a float expression

Return Value

If a non-existent input or output attribute is encountered in the float expression passed to the float2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Error Conditions

None

Example

```
InputRec {
 Float 130000;
 Bytes 230000;
} in;
in.230000 = float2bytes( in.130000 );
write( in );
```

Float bytes2float(Bytes b)

Returns the value of the array of bytes b in big-endian order as a Float. This function expects an array of 4 bytes in IEEE 754 32-bit single-precision floating-point type format.

Parameters

Table 1–55 Float bytes2float(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a Float value	non-null array of 4 bytes in IEEE 754 32-bit single-precision floating-point type format

Return Value

Returns the Float value represented by the given IEEE 32-bit single-precision floating-point type formatted byte array.

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2float function is not an IEEE 754 32-bit single-precision floating-point type byte array, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2float function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes Round-Trip-Time;
} in;
OutputRec {
 Float 30047;
} out;
out.30047 = bytes2float( in.Round-Trip-Time );
write( out );
```

String float2str(Float f)

Returns a string representing the specified Float f. This string representation format is defined according to the Java documentation for the Float.toString(float f) method.

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Float.html (toString(float)).

Parameters

Table 1–56 String float2str(Float f) Parameters

Parameter	Description	Acceptable Values
f	float value to represent as a string	a float expression

Return Value

Returns the String representing the specified float expression. This string representation format is defined according to the Java documentation for the Float.toString(float f) method

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Float.html (toString(float))

Error Conditions

If a non-existent input or output attribute is encountered in the float expression passed to the float2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Float 30047;
} in;
OutputRec {
 String Round-Trip-Time;
} out;
out.Round-Trip-Time = float2str( in.30047 );
write( out );
```

Float str2float(String s)

Returns the Float value represented by the specified string. The floating-point literal format is defined in section 3.10.2 of the Java Language Specification.

See http://docs.oracle.com/javase/specs/.

Parameters

Table 1–57 Float str2float(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing a float value	string expression consisting of one or more non-null strings where the expression is the floating-point literal format defined in section 3.10.2 of the Java Language Specification (see link above)

Return Value

Returns the Float value represented by the given string.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2float function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2float function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String Response-Time;
} in;
OutputRec {
 Float 30048;
} out;
out.30048 = str2float( in.Response-Time );
write( out );
```

Bytes double2bytes(Double d)

Returns the array of bytes representation for the specified Double (IEEE 754 64-bit double-precision floating-point type) value d.

Parameters

Table 1–58 Bytes double2bytes(Double d) Parameters

Parameter	Description	Acceptable Values
d	Double value to represent as an array of bytes	a double expression

Return Value

Returns the specified IEEE 754 64-bit double-precision floating-point expression represented as an array of bytes.

Error Conditions

If a non-existent input or output attribute is encountered in the float expression passed to the float2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Double 130010;
 Bytes 230010;
```

```
} in;
in.230010 = double2bytes( in.130010 );
write( in );
```

Double bytes2double(Bytes b)

Returns the value of the array of bytes b in big-endian order as a Double. This function expects an array of 8 bytes in IEEE 754 64-bit double-precision floating-point type format.

Parameters

Table 1–59 Double bytes2double(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a Double value	non-null array of 8 bytes in IEEE 754 64-bit double-precision floating-point type format

Return Value

Returns the Double value represented by the given IEEE 64-bit double-precision floating-point type formatted byte array.

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2double function is not an IEEE 754 64-bit double-precision floating-point type byte array, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2double function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes Double-Precision-Time;
} in;
OutputRec {
 Double 110011;
} out;
out.110011 = bytes2double( in.Double-Precision-Time );
write( out );
```

String double2str(double d)

Returns a string representing the specified Double d. This string representation format is defined according to the Java documentation for the Double.toString(double d) method.

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Double.html (toString(double)).

Parameters

Table 1–60 String double2str(double d) Parameters

Parameter	Description	Acceptable Values
d	double value to represent as a string	a double expression

Return Value

Returns the String representing the specified double expression. This string representation format is defined according to the Java documentation for the Double.toString(double d) method.

See http://docs.oracle.com/javase/1.3/docs/api/java/lang/Double.html (toString(double)).

Error Conditions

If a non-existent input or output attribute is encountered in the double expression passed to the double2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Double 110011;
 Double 110012;
} in;
OutputRec {
 String Total-Duration;
} out;
out.Total-Duration = double2str( 110011 + 110012 );
write( out );
```

Double str2double(String s)

Returns the double value represented by the specified string. The floating-point literal format is defined in section 3.10.2 of the Java Language Specification.

See http://docs.oracle.com/javase/specs/ (230798).

Parameters

Table 1–61 Double str2double(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing a double value	string expression consisting of one or more non-null strings where the expression is the floating-point literal format defined in section 3.10.2 of the Java Language Specification (see link above)

Return Value

Returns the Double value represented by the given string.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2float function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2float function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String version;
 String err;
 String sErr;
} in;
OutputRec {
 String 60072;
} out;
if( str2double( in.version ) > 1.2 ) {
 out.60072 = in.err;
}
else {
 out.60072 = in.sErr;
}
write( out );
```

Double randomDouble(Double min, Double max)

Returns a pseudo-random Double value in the range from min to max inclusive, where the value of min must be less than the value of max. Refer to Java's Random class for a description of the random number generator algorithm.

See http://docs.oracle.com/javase/1.3/docs/api/java/util/Random.html

Parameters

Table 1–62 Double randomDouble(Double min, Double max) Parameters

Parameter	Description	Acceptable Values
min	minimum value of inclusive range	a double expression that is not greater than the maximum value
max	maximum value of inclusive range	a double expression that is not less than the minimum value

Return Value

Returns a pseudo-random Double value within the specified inclusive range.

Error Conditions

If the value of the minimum parameter is not less than the value of the maximum parameter, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in either one of the double expressions passed to the randomDouble function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Double 110044;
 Double 110055;
 Double 110066;
} in;
if( in.110044 < in.110066 ) {
 in.110055 = randomDouble( in.110044, in.110066 );
}
else {
 in.110055 = 0.0;
}
write( in );
```

Bytes time2bytes(TimeInSecs t)

Returns the array of bytes representation for the specified 32-bit TimeInSecs value t. The byte order is big-endian.

Parameters

Table 1–63

Parameter	Description	Acceptable Values
t	time-in-seconds value to represent as an array of bytes	a time-in-seconds expression

Return Value

Returns a Bytes value corresponding to the big-endian byte array representation of the given time-in-seconds expression.

Error Conditions

If a non-existent input or output attribute is encountered in the time-in-seconds expression passed to the time2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 TimeInSecs RecordOpeningTime;
} in;
OutputRec {
 Bytes 111000;
} out;
out.111000 = time2bytes( in.RecordOpeningTime );
write( out );
```

TimeInSecs bytes2TimeInSecs(Bytes b)

Returns the value of the array of bytes b in big-endian order as a 32-bit TimeInSecs value. The length of the array of bytes may be less than the 4 bytes to store a TimeInSecs value, in which case the high order bytes will be padded with zeroes. However, the length of the array of bytes cannot be greater than 4.

Parameters

Table 1–64 TimeInSecs bytes2TimeInSecs(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a TimeInSecs value	non-null byte array containing from one to four bytes

Return Value

Returns the TimeInSecs value represented by the given byte array.

The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting time-in-seconds values.

Table 1–65 TimeInSecs bytes2TimeInSecs(Bytes b) Return Values

Byte array	TimeInSecs Value
(127)	127
(-1)	255
(1,2)	258
(-1,1)	65281
(1, -1)	511
(-1, -1)	65535
(1, 0, 2)	65538

Table 1–65 (Cont.) TimeInSecs bytes2TimeInSecs(Bytes b) Return Values

Byte array	TimeInSecs Value
(2, 0, 1)	131073
(1, 0, 0, 2)	16777218
(2, 0, 0, 1)	33554433

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2TimeInSecs function is not from one to four bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2TimeInSecs function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 55;
} in;
OutputRec {
 Integer 50021;
} out;
out.50021 = bytes2TimeInSecs( in.55 );
write( out );
```

String time2str(TimeInSecs timeInSecs, String format)

Returns a string representing the given timeInSecs according to the specified format. The format is as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class.

See
http://docs.oracle.com/javase/1.3/docs/api/java/text/SimpleDateFormat.html

Parameters

Table 1–66 String time2str(TimeInSecs timeInSecs, String format) Parameters

Parameter	Description	Acceptable Values
timeInSecs	time-in-seconds value to format as a string	a time-in-seconds expression

Table 1–66 (Cont.) String time2str(TimeInSecs timeInSecs, String format) Parameters

Parameter	Description	Acceptable Values
format	pattern specifying time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)

Return Value

Returns the String representing the given time-in-seconds expression in the specified format.

Error Conditions

Passing a null format string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the format string expression passed to the time2str function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the time-in-seconds expression passed to the time2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

If a non-existent input or output attribute is encountered in the format string expression passed to the time2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
String TIME_FORMAT = "EEE MMM dd HH:mm:ss z yyyy";
InputRec {
 Integer Timestamp;
} in;
OutputRec {
 String 42000;
} out;
out.42000 = time2str( in.Timestamp, TIME_FORMAT );
write( out );
```

TimeInSecs str2TimeInSecs(String s, String format)

Returns the TimeInSecs value represented by the string s in the given format. The format is defined according to the Java documentation of the time pattern format syntax for the SimpleDateFormat class.

See
http://docs.oracle.com/javase/1.3/docs/api/java/text/SimpleDateFormat.html

Parameters

Table 1–67 TimeInSecs str2TimeInSecs(String s, String format) Parameters

Parameter	Description	Acceptable Values
s	String representing a TimeInSecs value	a String expression consisting of one or more non-null strings representing a TimeInSecs value in the specified format
format	pattern specifying time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)

Return Value

Returns the TimeInSecs value represented by the given string in the specified format.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the time string expression does not match the pattern of the format string, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If the format string expression passed to the str2TimeInSecs function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output String attribute is encountered in either the time string expression or format string expression passed to the str2TimeInSecs function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
String TIME_FORMAT = "EEE MMM dd HH:mm:ss yyyy";
InputRec {
 String Start-Time;
} in;
OutputRec {
 Integer 30016;
} out;
out.30016 = str2TimeInSecs( in.Start-Time, TIME_FORMAT );
write( out );
```

Bytes time2bytes(TimeInMilliSecs t)

Returns the array of bytes representation for the specified 64-bit TimeInMilliSecs value t. The byte order is big-endian.

Parameters

Table 1–68 Bytes time2bytes(TimeInMilliSecs t) Parameters

Parameter	Description	Acceptable Values
t	time-in-milliseconds value to represent as an array of bytes	a time-in-milliseconds expression

Return Value

Returns a Bytes value corresponding to the big-endian byte array representation of the given time-in-milliseconds expression.

Error Conditions

If a non-existent input or output attribute is encountered in the time-in-milliseconds expression passed to the time2bytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 TimeInMilliSecs StopTime;
} in;
OutputRec {
 Bytes 123321;
} out;
out.123321 = time2bytes( in.StopTime );
write( out );
TimeInMilliSecs StopTime;
} in;
OutputRec {
 Bytes 123321;
} out;
out.123321 = time2bytes( in.StopTime );
write( out );
```

TimeInMilliSecs bytes2TimeInMilliSecs(Bytes b)

Returns the value of the array of bytes b in big-endian order as a 64-bit TimeInMilliSecs value. The length of the array of bytes may be less than the 8 bytes to store a TimeInMilliSecs value, in which case the high order bytes will be padded with zeroes. However, the length of the array of bytes cannot be greater than 8.

Parameters

Table 1–69 TimeInMilliSecs bytes2TimeInMilliSecs(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to a TimeInMilliSecs value	non-null byte array containing from one to eight bytes

Return Value

Returns the TimeInMilliSecs value represented by the given byte array.

The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting time-in-milliseconds values.

Table 1–70 TimeInMilliSecs bytes2TimeInMilliSecs(Bytes b) Return Values

Byte array	TimeInMilliSecs Value
(-1)	255
(1,2)	258
(2, 0, 1)	131073
(1, 0, 0, 2)	16777218
(1, 2, 3, 4, 5)	4328719365
(1, -1, -1, -1, -1, -1)	2199023255551
(1, 0, 1, 0, 1, 0, 1)	281479271743489
(2, 0, 0, 0, 0, 0, 0, 0)	144115188075855872
(0, 0, 0, 0, 0, 0, 0, 2)	2

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2TimeInMilliSecs function is not from one to eight bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2TimeInMilliSecs function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes CallTime;
} in;
OutputRec {
 Long 321123:
} out;
out.321123 = bytes2TimeInMilliSecs( in.CallTime );
write( out );
```

String time2str(TimeInMilliSecs timeInMilliSecs, String format)

Returns a string representing the given timeInMilliSecs according to the specified format. The format is defined according to the Java documentation of the time pattern format syntax for the SimpleDateFormat class.

See
http://docs.oracle.com/javase/1.3/docs/api/java/text/SimpleDateFormat.html

Parameters

Table 1–71 String time2str(TimeInMilliSecs timeInMilliSecs, String format) Parameters

Parameter	Description	Acceptable Values
timeInMilliSecs	time-in-milliseconds value to format as a string	a time-in-milliseconds expression
format	pattern specifying time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)

Return Value

Returns the String representing the given time-in-milliseconds expression in the specified format.

Error Conditions

Passing a null format string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the format string expression passed to the time2str function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the time-in-milliseconds expression passed to the time2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

If a non-existent input or output attribute is encountered in the format string expression passed to the time2str function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
String TIME_FORMAT = "EEE MMM dd HH:mm:ss yyyy";
InputRec {
 TimeInMilliSecs Timestamp;
} in;
OutputRec {
 String 42000;
} out;
out.42000 = time2str( in.Timestamp, TIME_FORMAT );
write( out );
```

TimeInMilliSecs str2TimeInMilliSecs(String s, String format)

Returns the TimeInMilliSecs value represented by the string s in the given format. The format is defined according to the Java documentation of the time pattern format syntax for the SimpleDateFormat class.

See
http://docs.oracle.com/javase/1.3/docs/api/java/text/SimpleDateFormat.html

Parameters

Table 1–72

Parameter	Description	Acceptable Values
s	String representing a TimeInMilliSecs value	a String expression consisting of one or more non-null strings representing a TimeInMilliSecs value in the specified format
format	pattern specifying time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)

Return Value

Returns the TimeInMilliSecs value represented by the given string in the specified format.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the time string expression does not match the pattern of the format string, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If the format string expression passed to the str2TimeInMilliSecs function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output String attribute is encountered in either the time string expression or format string expression passed to the str2TimeInMilliSecs function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String WTIME;
} in;
OutputRec {
```

```
 TimeInMilliSecs 10001;
} out;
out.10001 = str2TimeInMilliSecs( in.WTIME, "yyyyMMddHHmmss" );
write( out );
```

TimeInMilliSecs currentTime()

Returns the current time in milliseconds since January 1, 1970 coordinated universal time (UTC).

Parameters

None

Return Value

Returns the current time in milliseconds since January 1, 1970 coordinated universal time (UTC).

Error Conditions

None

Example

```
InputRec {
 TimeInSecs 20003;
} in;
OutputRec {
 Long 11005;
} out;
if( fieldExists( in, "20003" ) ) {
 out.11005 = in.20003 * 1000;
}
else {
 out.11005 = currentTime();
}
write( out );
```

String convertDateTime(String srcDateTime, … String dstDateTimeFormat)

The actual name of this function is:

String convertDateTime(String srcDateTime, String srcTimeZone, String srcDateTimeFormat, String dstTimeZone, String dstDateTimeFormat)

Converts the given source date-time string in the specified time zone and format to a destination date-time string in the specified time zone and format. Valid time zone strings are those supported by Java, which is the list returned by TimeZone.getAvailableIDs(). Formats are defined according to the Java documentation of the time pattern format syntax for the SimpleDateFormat class.

See
http://docs.oracle.com/javase/1.3/docs/api/java/text/SimpleDateFormat.html

Parameters

Table 1–73 String convertDateTime(String srcDateTime, … String dstDateTimeFormat) Parameters

Parameter	Description	Acceptable Values
srcDateTime	source date-time string	a String expression consisting of one or more non-null strings representing a date-time in the specified format
srcTimeZone	source time zone string	time zone string supported by the Java TimeZone class as returned by the TimeZone.getAvailableIDs()
srcDateTimeFormat	pattern specifying source time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)
dstTimeZone	destination time zone string	time zone string supported by the Java TimeZone class as returned by the TimeZone.getAvailableIDs()
dstDateTimeFormat	pattern specifying destination time format	a string expression as defined in the Java documentation of the time pattern format syntax for the SimpleDateFormat class (see link above)

Return Value

The string representing the conversion of the srcDateTime string in the srcTimeZone and srcDateTimeFormat to the dstTimeZone using the format specified by the dstDateTimeFormat pattern string.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the source date-time string expression does not match the pattern of the source date-time format string, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If the destination date-time string expression does not match the pattern of the destination date-time format string, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If an unsupported time zone string expression is passed to the convertDateTime function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a format string expression passed to the convertDateTime function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output String attribute is encountered in any of the string expressions passed to the convertDateTime function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
String srcFormat1 = "yyyyMMddHHmmssSSS";
String srcTZ1   = "GMT";
String srcFormat2 = "MM/dd/yyyy HH:mm:ss.SSS";
String srcTZ2   = "America/New_York";
String dstFormat = "yyyy/MM/dd HH:mm:ss.SSS";
String dstTZ1   = "EST";
String dstTZ2   = "UTC";
InputRec {
 String srcTime1;
 String srcTime2;
} in;
OutputRec {
 String dstTime1;
 String dstTime2;
} out;
out.dstTime1 = convertDateTime( in.srcTime1, srcTZ1, srcFormat1,
                dstTZ1, dstFormat );
out.dstTime2 = convertDateTime( in.srcTime2, srcTZ2, srcFormat2,
                dstTZ2, dstFormat );
write( out );
```

As an example, if the in.srcTime1 string is "20001027051117020" then the out.dstTime1 string will be "2000/10/27 01:11:17.020" and if the in.srcTime2 string is "1/1/2001 13:12:11.100" then the out.dstTime2 string will be "2001/01/01 18:12:11.100".

Bytes IP2bytes(IP i)

Returns the byte array representation for the specified IP address value i. The byte order is the same as the numbers appear in dotted quad notation of an IP address.

Parameters

Table 1–74 Bytes IP2bytes(IP i) Parameters

Parameter	Description	Acceptable Values
i	IP address to represent as an array of bytes	an IP address value

Return Value

Returns a Bytes value representation of the given IP address. The byte order is the same as the numbers appear in the dotted quad notation of an IP address. For example, if the IP address is 1.2.3.4 then the byte array representation is (1, 2, 3, 4) where element zero of the byte array is the first value in the list of values.

Error Conditions

None

Example

```
InputRec {
 IP 30055;
} in;
OutputRec {
 Integer 30035;
} out;
out.30035 = bytes2int( IP2bytes( in.30055 ) );
write( out );
```

IP bytes2IP(Bytes b)

Returns the value of the array of bytes b as an IP address. This function expects an array of 4 bytes where each byte is interpreted as an unsigned decimal number in the IP address.

Parameters

Table 1–75 IP bytes2IP(Bytes b) Parameters

Parameter	Description	Acceptable Values
b	byte array to convert to an IP address	non-null array of four bytes

Return Value

Returns the IP address represented by the given byte array. Each byte is interpreted as an unsigned decimal number in the IP address.

The following table illustrates some example byte arrays (where the first value listed corresponds to element 0 of the byte array) and resulting IP address.

Table 1–76 IP bytes2IP(Bytes b) Return Value

Byte array	IP address
(1,0,0,2)	1.0.0.2
(2,0,0,1)	2.0.0.1

Error Conditions

Passing a null byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the byte array passed to the bytes2IP function is not four bytes in length, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output Bytes attribute is passed to the bytes2IP function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 11;
} in;
OutputRec {
 IP 30001;
} out;
out.30001 = bytes2IP( in.11 );
write( out );
```

String IP2str(IP IP address)

Returns a string representing the specified IP address as four unsigned decimal numbers, each representing eight bits separated by periods.

Parameters

Table 1–77 String IP2str(IP IP address) Parameters

Parameter	Description	Acceptable Values
IP address	IP address to represent as a string	an IP address value

Return Value

Returns a string representing the specified IP address as four unsigned decimal numbers, each representing eight bits separated by periods.

Error Conditions

None

Example

```
InputRec {
 IP 40004;
} in;
OutputRec {
 String 11035;
} out;
out.11035 = IP2str( in.40004 );
write( out );
```

IP str2IP(String s)

Returns the IP address value represented by the string s, where the expected format is four unsigned decimal numbers, each representing eight bits, separated by periods.

Parameters

Table 1–78 IP str2IP(String s) Parameters

Parameter	Description	Acceptable Values
s	string representing an IP address	string expression in the format of four unsigned decimal numbers, each representing eight bits, separated by periods

Return Value

Returns the IP address value represented by the given string.

Error Conditions

Passing a null string to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If the string expression passed to the str2IP function is not in the acceptable format, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in the string expression passed to the str2IP function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 String NASIP;
} in;
OutputRec {
 IP 21022;
} out;
out.21022 = str2IP( in.NASIP );
write( out );
```

String object2str(Object o)

Returns a string representing the specified object.

Parameters

Table 1–79 String object2str(Object o) Parameters

Parameter	Description	Acceptable Values
o	Object to represent as a string	an Object

Return Value

Returns the string representation of the specified Object.

Error Conditions

None

Example

```
InputRec {
 Object 20204;
} in;
OutputRec {
 String SGSN-Address;
} out;
out.SGSN-Address = object2str( in.20204 );
write( out );
```

String list2str(List l)

Returns a string representing the specified list.

Parameters

Table 1–80 String list2str(List l) Parameters

Parameter	Description	Acceptable Values
l	List to represent as a string	a List

Return Value

Returns the string representation of the specified List.

Error Conditions

None

Example

```
InputRec {
 List 20209;
} in;
OutputRec {
 String List-Of-Traffic-Volumes;
} out;
out.List-Of-Traffic-Volumes = list2str( in.20209 );
write( out );
```

Bytes subbytes(Bytes b, Integer beginIndex, Integer endIndex)

Returns a copy of the array of bytes that begins at the specified beginIndex and extends to the byte at index endIndex.

Parameters

Table 1–81 Bytes subbytes(Bytes b, Integer beginIndex, Integer endIndex) Parameters

Parameter	Description	Acceptable Values
b	byte array to get sub-array of	non-null and non-empty byte array

Table 1–81 (Cont.) Bytes subbytes(Bytes b, Integer beginIndex, Integer endIndex)

Parameter	Description	Acceptable Values
beginIndex	beginning index, inclusive	integer expression with value >= 0 and <= end index
endIndex	ending index, exclusive	integer expression with value >= begin index and <= array length

Return Value

Returns a copy of the array of bytes that begins at the specified `beginIndex` and extends to the byte at index `endIndex`.

Error Conditions

Passing a null or empty byte array to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null string can occur in NPL if a String variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If either the begin index or end index expression does not evaluate to an acceptable value as defined in the above Parameters table, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning indicating the invalid expression value will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is encountered in any of the parameter expressions passed to the subbytes function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
InputRec {
 Bytes 120021;
} in;
OutputRec {
 Short  130031;
 Integer 140041;
} out;
out.130031 = bytes2short( subbytes( in.120021, 4, 5 ) );
out.140041 = bytes2int( subbytes( in.120021, 0, 3 ) );
write( out );
```

Boolean fieldExists(InputRec rec, String fieldID)

Returns true if the specified field exists in the given input record and false if it does not exist. The return type Boolean is a private type internal to NPL.

Note: This function is only intended to be used in a conditional expression of an "if" statement.

Parameters

Table 1–82 Boolean fieldExists(InputRec rec, String fieldID) Parameters

Parameter	Description	Acceptable Values
rec	input record in which to check for field existence	an input record declared in the NPL program
fieldID	field ID to search for	a string expression representing a field ID

Return Value

Returns true if the specified field exists in the given input record and false if it does not exist.

Error Conditions

None

Example

```
InputRec {
 String 20108;
} in1;
InputRec {
 Long 20200;
} in2;
OutputRec {
 String Gprs-Record-Type;
} out1;
OutputRec {
 Long CallEventRecordType;
} out2;
if( fieldExists( in1, "20108" ) ) {
 out1.Gprs-Record-Type = in1.20108;
 write( out1 );
}
else if ( fieldExists( in2, "20200" ) ) {
 out2.CallEventRecordType = in2.20200;
}
```

Boolean fieldExists(OutputRec rec, String fieldID)

Returns true if the specified field exists in the given output record and false if it does not exist. The return type `Boolean` is a private type internal to NPL.

> **Note:** This function is only intended to be used in a conditional expression of an "if" statement.

Parameters

Table 1–83 Boolean fieldExists(OutputRec rec, String fieldID) Parameters

Parameter	Description	Acceptable Values
rec	output record in which to check for field existence	an output record declared in the NPL program

Table 1–83 (Cont.) Boolean fieldExists(OutputRec rec, String fieldID) Parameters

Parameter	Description	Acceptable Values
FieldID	field ID to search for	a string expression representing a field ID

Return Value

Returns true if the specified field exists in the given output record and false if it does not exist.

Error Conditions

None

Example

```
InputRec {
 Byte 221122;
 Long 112211;
} in;
OutputRec {
 Long 121212;
 Long 131313;
} out;
if( in.221122 != 0 ) {
 out.121212 = in.112211;
}
if( fieldExists( out, "121212" ) ) {
 out.131313 = out.121212 / 1000;
}
write( out );
```

CopyBits

As the name implies, a `copyBits` built-in function is used to copy a specified number of bits from one variable to another. More explicitly, a specified number of bits (`slength`) are copied from a source variable (`src`) starting at an offset (`soffset`) to a destination variable (`dst`) starting at an offset (`doffset`) and within a given range (`dlength`). The destination range (`dlength`) must be large enough to accommodate the range from the source. The `copyBits` function is overloaded.

Note: Use of the copyBits function will degrade the performance of a node.

```
Byte copyBits(Byte  src, Integer soffset, Integer slength,
       Byte  dst, Integer doffset, Integer dlength)
Byte copyBits(Short  src, Integer soffset, Integer slength,
       Byte  dst, Integer doffset, Integer dlength)
Byte copyBits(Integer src, Integer soffset, Integer slength,
       Byte  dst, Integer doffset, Integer dlength)
Byte copyBits(Bytes  src, Integer soffset, Integer slength,
       Byte  dst, Integer doffset, Integer dlength)
Byte copyBits(Long  src, Integer soffset, Integer slength,
       Byte  dst, Integer doffset, Integer dlength)
Short copyBits(Byte  src, Integer soffset, Integer slength,
        Short  dst, Integer doffset, Integer dlength)
```

```
Short copyBits(Short  src, Integer soffset, Integer slength,
        Short  dst, Integer doffset, Integer dlength)
Short copyBits(Integer src, Integer soffset, Integer slength,
        Short  dst, Integer doffset, Integer dlength)
Short copyBits(Bytes  src, Integer soffset, Integer slength,
        Short  dst, Integer doffset, Integer dlength)
Short copyBits(Long  src, Integer soffset, Integer slength,
        Short  dst, Integer doffset, Integer dlength)
Integer copyBits(Byte  src, Integer soffset, Integer slength,
         Integer dst, Integer doffset, Integer dlength)
Integer copyBits(Short  src, Integer soffset, Integer slength,
         Integer dst, Integer doffset, Integer dlength)
Integer copyBits(Integer src, Integer soffset, Integer slength,
         Integer dst, Integer doffset, Integer dlength)
Integer copyBits(Bytes  src, Integer soffset, Integer slength,
         Integer dst, Integer doffset, Integer dlength)
Integer copyBits(Long  src, Integer soffset, Integer slength,
         Integer dst, Integer doffset, Integer dlength)
Bytes copyBits(Byte  src, Integer soffset, Integer slength,
        Bytes  dst, Integer doffset, Integer dlength)
Bytes copyBits(Short  src, Integer soffset, Integer slength,
        Bytes  dst, Integer doffset, Integer dlength)
Bytes copyBits(Integer src, Integer soffset, Integer slength,
        Bytes  dst, Integer doffset, Integer dlength)
Bytes copyBits(Bytes  src, Integer soffset, Integer slength,
        Bytes  dst, Integer doffset, Integer dlength)
Bytes copyBits(Long  src, Integer soffset, Integer slength,
        Bytes  dst, Integer doffset, Integer dlength)
Long copyBits(Byte  src, Integer soffset, Integer slength,
       Long  dst, Integer doffset, Integer dlength)
Long copyBits(Short  src, Integer soffset, Integer slength,
       Long  dst, Integer doffset, Integer dlength)
Long copyBits(Integer src, Integer soffset, Integer slength,
       Long  dst, Integer doffset, Integer dlength)
Long copyBits(Bytes  src, Integer soffset, Integer slength,
       Long  dst, Integer doffset, Integer dlength)
Long copyBits(Long  src, Integer soffset, Integer slength,
       Long  dst, Integer doffset, Integer dlength)
```

Parameters

Table 1–84 CopyBits Parameters

Parameter	Description	Acceptable Values
src	variable to copy bits from	Byte, Short, Integer, Long or non-null Bytes expression
soffset	offset into source will copying will begin	Integer expression with value from 0 to length of source in bits minus 1
slength	how much of source value to copy	Integer expression with value that must be less than the length of source in bits
dst	variable having bits copied into	Byte, Short, Integer, Long or non-null and non-empty Bytes expression
doffset	offset into destination where source will be copied	Integer expression with value from 0 to length of destination in bits minus 1
dlength	how much of destination value to copy over	Integer expression with value from 0 to length of destination in bits minus 1

Return Value

Returns a new variable that represents the destination with the specified source bits copied into the appropriate location. The return type is one of Byte, Short, Integer, Long or Bytes depending on which overloaded copyBits function is called.

Table 1–85 illustrates some sample return values for various parameters, assuming an Integer source and destination.

Table 1–85 Return Values

src	soffset	slength	dst	doffset	dlength	return value
32	0	0	512	0	0	512
32	5	0	512	1	1	512
32	5	1	512	1	1	514
32	5	1	512	16	1	66048
32	0	16	512	16	16	2097664
32	0	16	512	0	16	32
32	0	16	512	1	31	64

Error Conditions

Passing a null byte array as the source or destination parameter to this function will result in the statement containing that function call being skipped and execution continuing with the next statement in the NPL program. A warning of this condition will be written to the node's log file if the node's configured debug level is set at the logging warnings level. A null byte array can occur in NPL if a Bytes variable is not initialized, or perhaps a built-in function or Java hook returns a null value.

If either the source offset or destination offset expression does not evaluate to an acceptable value as defined in the above Parameters table, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning indicating the invalid expression value will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If the source length parameter value exceeds the length for the source type or the destination length parameter value exceeds the length for the destination type, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program. A warning indicating the invalid expression value will be written to the node's log file if the node's configured debug level is set at the logging warnings level.

If a non-existent input or output attribute is used in any expression passed to the copyBits function, the statement that contains the function call will be skipped and execution continues with the next statement in the NPL program.

Example

```
Integer format_bit;
InputRec {
 Integer flags;
 Integer format1;
 Integer format2;
} in;
OutputRec {
 Integer 113311;
```

```
} out;
format_bit = copyBits( in.flags, 6, 1, format_bit, 1, 32 );
if( format_bit != 0 ) {
 out.113311 = copyBits( in.format1, 0, 16, out.113311, 0, 16 );
 out.113311 = copyBits( in.format2, 0, 16, out.113311, 16, 16 );
}
else {
 out.113311 = copyBits( in.format1, 16, 16, out.113311, 0, 16 );
 out.113311 = copyBits( in.format2, 16, 16, out.113311, 16, 16 );
}
write( out );
```

Expressions

An expression can be used in four areas of an NPL program: local variable initialization ("Local variables"), right-hand side of an assignment statement, conditional (relational and/or logical) of an "if' statement and as an argument to a function call. The categories of expressions supported by NPL are similar to those found in most programming languages: arithmetic (addition, subtraction, multiplication, division, unary plus and minus), String concatenation (see "String concatenation operator"), variable value, literal (integer, floating-point or string), explicit cast and function whose return is not void. Parentheses can be used to modify the meaning (or order of evaluation) of an expression.

Program statements

The program statements define the part of an NPL program that is executed each time the `NPLFieldProcessor` object's processData method is called. These statements determine how an incoming data record is processed and what data is passed on to the `NPLFieldProcessor` object's data receiver for output by the node.

Assignment Statement

As its name implies, the assignment statement is used to assign a value to a variable. The syntax of an assignment statement is a variable identifier, followed by an equal (=) sign, followed by the expression whose value is being assigned to the variable, and ending with a semicolon:

```
identifier = expression ;
```

The identifier must have already been declared in the NPL program and the expression must evaluate to the type of the variable named by identifier to be a valid assignment statement.

NPL supports record-to-record assignment only in EP nodes. When one record is assigned to another in an NPL assignment statement, the destination record will become a copy of the source record. Any attributes in the destination record prior to the assignment will be lost.

Assignment Statement Execution

During the execution of an NPL program, an assignment statement is either successful or ignored. Whether the assignment is successful or ignored, execution will continue with the next statement in the NPL program. An assignment statement is ignored if the expression on the right-hand side references a non-existent attribute of a data record or an error occurs when evaluating that expression.

Assignment from a non-existent attribute is ignored because this is a likely occurrence during normal processing of data records by a node. Each data record processed by a node may not contain all the possible attributes in the data format expected by a node.

If an error occurs when evaluating the expression on the right-hand side of an assignment statement, the assignment is ignored with a warning output to the node's log. A typical reason for an error when evaluating an assignment expression is an error from a function call (such as an inappropriate parameter to a built-in function).

If Statement

The "if" statement is a conditional statement used to determine the control flow in an NPL program. The general form of an "if" statement in NPL is:

```
if (if-condition)
 statement-or-block
else if (else-if-condition)
 statement-or-block
else
 statement-or-block
```

A "if" statement may contain zero or more "else if" parts and an optional "else" part.

The condition (*if-condition or else-if-condition*) is an expression that evaluates to true or false. This conditional expression may be one of the following:

- A relational expression (see "Relational operators")
- Two or more relational expressions separated by logical operators (see "Relational operators")
- A call to the `fieldExists` built-in function

The statement-or-block is one of:

- A single program statement (i.e. assignment statement, if statement or function statement)
- Zero or more program statements enclosed in a single set of braces

If Statement Execution

The conditional expressions (*if-condition or else-if-condition*) are evaluated in order; if an expression is true, the *statement-or-block* associated with it is executed, and this terminates the whole if chain. The *statement-or-block* associated with the last `else` part will be executed if none of the other conditions is satisfied.

If a conditional expression references an attribute that does not exist, the entire if statement will be ignored with execution continuing with the next statement in the NPL program. If an error occurs during the evaluation of a conditional expression, the entire if statement is ignored with a warning output to the node's log file.

Function Statement

A function statement is simply a call to a built-in function or Java hook (see "Java hooks") that is not part of an expression. A semicolon terminates the function statement. The NPL built-in functions that fall into the function statement category are: logError, logWarning, longInfo, logTrace and write.

Java hooks

Java hooks are an advanced feature of NPL that makes it possible to call a Java method from an NPL program. The Java hook method(s) to be called from NPL need to be defined in an interface that extends `com.nt.udc.ndk.node.IDCMethodHandler` or extends an interface that is already a sub-interface of `IDCMethodHandler`. In the second case all methods defined in the interface inheritance hierarchy up to `IDCMethodHandler` are potentially callable from NPL.

A Java method that is callable from NPL may have zero parameters, or one or more parameters of the types defined in Table 1–86. The table lists valid Java hook method parameter types, their corresponding NPL types, and whether a parameter of the given type is passed by value or by reference. The implicit and explicit type conversions supported by NPL also apply when calling a Java hook from NPL.

Table 1–86 Java Hook Parameter Types

Java Hook Method Parameter Type	Corresponding NPL Type	Parameter Passed By
com.nt.udc.ndk.node.DCFieldContainer	OutputRec, InputRec	reference
com.nt.udc.ndk.node.StringField	String	value
com.nt.udc.ndk.node.IntField	Integer	value
com.nt.udc.ndk.node.LongField	Long	value
com.nt.udc.ndk.node.SecondsField	TimeInSecs	value
com.nt.udc.ndk.node.IPField	IP	value
com.nt.udc.ndk.node.MillisField	TimeInMilliSecs	value
com.nt.udc.ndk.node.MacField	MAC	value
com.nt.udc.ndk.node.UINT128Field	UInt128	value
com.nt.udc.ndk.node.ByteField	Byte	value
com.nt.udc.ndk.node.ShortField	Short	value
com.nt.udc.ndk.node.IPV6Field	IPv6	value
com.nt.udc.ndk.node.FloatField	Float	value
com.nt.udc.ndk.node.DoubleField	Double	value
com.nt.udc.ndk.node.BytesField	Bytes	value
com.nt.udc.ndk.node.ObjectField	Object	value
com.nt.udc.ndk.node.ListField	List	value
com.nt.udc.ndk.node.DCField	String, Integer, Long, TimeInSecs, IP, TimeInMilliSecs, MAC, UInt128, Byte, Short, IPv6, Float, Double, Bytes, Object, List	value
com.nt.udc.ndk.node.MapField	Map	reference

Note: When the abstract DCField is used as a parameter type in a Java hook method it is up to the implementation to verify that the correct field type has been passed to the method.

Valid return types for a Java hook method are: `StringField`, `IntField`, `LongField`, `SecondsField`, `IPField`, `MillisField`, `MacField`, `UINT128Field`, `ByteField`,

ShortField, IPV6Field, FloatField, DoubleField, BytesField, ObjectField, ListField and void. Note that the abstract DCField **is not** a valid return type for a Java hook method since NPL needs a concrete type to perform type resolution in expressions.

There are two different mechanisms to make Java hooks accessible to a NPL program: import declaration or JavaHook declaration. Both mechanisms can be used in the same NPL program.

Import Declaration

An NPL program may import one method handler that contains one or more Java hooks to be called from the NPL program. This mechanism requires the object instance implementing the method handler to be passed to the NPL program via the NPLFieldProcessor constructor.

Below is an example method handler interface based on the time enhancer from an earlier release (PSA 2.5):

```
package com.nt.udc.enhancer;
import com.nt.udc.ndk.node.*;
public interface ITEMethodHandler extends IDCMethodhandler
{
  public MillisField finalDuration(StringField session_id,
                   MillisField connect_time,
                   MillisField disconnect_time);
}
```

Note: The public modifiers on the interface and method declarations are necessary for NPL to be able to access the Java hook methods.

An object instance of the class implementing the Java hook method handler must be passed to the NPLFieldProcessor constructor in order for the Java hook methods to be callable from an NPL program. The NPLFieldProcessor constructor that accepts a method handler parameter is:

```
public NPLFieldProcessor( Class outputDataType, LoggerIfc inLogger,
             String scratchDir, String filename,
             IDCMethodHandler methodHandler )
  throws NodeProcessingException
```

A simple example of a NPL program that uses the time enhancer method handler defined by the example ITEMethodHandler above is:

```
import com.nt.udc.enhancer.ITEMethodHandler;
InputRec {
 String     1010; // session id
 TimeInMilliSecs 7002; // call connect time
 TimeInMilliSecs 7003; // call disconnect time
} in;
OutputRec {
 TimeInMilliSecs 7022; // final duration
} out;
out.7022 = Java.finalDuration(in.1010, in.7002, in.7003);
write(out);
```

Note the import statement to make ITEMethodHandler methods available to this NPL program and the "Java." prefix on the call to finalDuration. The NPL program imports the method handler interface, not the class implementing the actual Java

hooks. An imported Java hook must be prefixed with "Java." when called so NPL will recognize that the Java hook is from the imported method handler interface.

The steps for implementing and using imported Java hooks are summarized here:

1. Write a method handler interface that contains the Java hook method(s). This interface must either extend IDCMethodHandler or extend an interface that is already a sub-interface of IDCMethodHandler.
2. Implement the Java hook methods.
3. Make sure an instance of the class that implements the Java hook method(s) is passed to the NPLFieldProcessor.
4. Write the NPL file that will be making calls to the Java hook methods declared in the method handler interface. Make sure to import the method handler interface and prefix any Java hook calls with a ".Java".

JavaHook Declaration

The JavaHook declaration is the second mechanism for making Java hooks accessible from an NPL program. A NPL program may have zero or more JavaHook declarations. The JavaHook declaration consists of the `JavaHook` keyword followed by an identifier, an assignment to the fully qualified name of the class implementing the method handler, and a semicolon terminator:

```
JavaHook identifier = fully-qualified-class-name ;
```

The identifier provides the name used to reference the method handler from within the NPL program. A method handler identifier must begin with a letter or underscore followed by a sequence of zero or more letters, digits or underscores. A Java hook declared with this syntax is called in NPL by prefixing the method name with the identifier followed by a period.

> **Note:** The *fully-qualified-class-name* is the fully qualified name of the class that implements the Java hooks of the method handler. This class must implement an interface that extends `IDCMethodHandler` and have a zero-argument constructor.

The following example method handler implementation class uses the `ITEMethodHandler` interface example from the section "Import Declaration".

```
package com.nt.udc.enhancer;
import com.nt.udc.ndk.node.*;
public class ITEMethodHandlerImpl implements ITEMethodHandler
{
  // Method handler implementation classes used in NPL JavaHook declarations
  // must have a zero-argument constructor.
  public ITEMethodHandlerImpl()
  {
  }
// finalDuration Java hook declared in ITEMethodHandler
  public MillisField finalDuration(StringField session_id,
                  MillisField connect_time,
                  MillisField disconnect_time)
  {
    // implement finalDuration here
  }
}
```

The following simple NPL program is the example from the section "Import Declaration" updated to use a JavaHook declaration instead of an import statement:

```
JavaHook TEHandler = com.nt.udc.enhancer.ITEMethodHandlerImpl;
InputRec {
 String     1010; // session id
 TimeInMilliSecs 7002; // call connect time
 TimeInMilliSecs 7003; // call disconnect time
} in;
OutputRec {
 TimeInMilliSecs 7022; // final duration
} out;
out.7022 = TEHandler.finalDuration(in.1010, in.7002, in.7003);
write(out);
```

Note: Note that the JavaHook declaration uses the fully qualified name of the class that implements the method handler, not the method handler interface like the import statement.

The method handler class is instantiated internally by the NPL program using the zero-argument constructor. An accessor method is provided by the `NPLFieldProcessor` to retrieve a reference to a NPL program method handler if the zero-argument constructor is not sufficient to initialize a method handler instance:

```
public IDCMethodHandler getMethodHandler(String name)
```

The name is the identifier given the Java hook in the NPL `JavaHook` declaration.

The steps for implementing and using JavaHook declared method handlers are summarized here:

1. Write a method handler interface that contains the Java hook method(s). This interface must either extend IDCMethodHandler or extend an interface that is already a sub-interface of IDCMethodHandler.
2. Write a class that implements the interface from Step 1. Make sure the class has a zero-argument constructor.
3. Write the NPL file that will be making calls to the Java hook methods implemented by the class from Step 2. Make sure to use the fully qualified name of the class from Step 2 in the `JavaHook` declaration and prefix any Java hook calls with the identifier from the declaration and a period.

2

Working with Oracle CDR Format Java Hooks in NPL

This chapter lists and describes the Java hooks available for the Oracle Communications Offline Mediation Controller Oracle CDR Format Collection Cartridge (CC).

About Oracle CDR Format Java Hooks

Java hooks are an advanced feature of NPL (Node Programming Language) that enable Offline Mediation Controller to call a Java method from an NPL program. For more information on using Java hooks with NPL, see the discussion on Java hooks in *Offline Mediation Controller Cartridge Development Kit NPL Reference Guide*.

Table 2–1 lists the Oracle CDR Format Java hooks methods.

Table 2–1 Oracle CDR Format Java Hooks Method Summary

Modifier and Type	Method and Description
IntField	hasHeaderFields(DCFieldContainer *in*) Verifies if the record contains header fields.
IntField	hasTrailerFields(DCFieldContainer *in*) Verifies if the record contains trailer fields.
IntField	hasAnyAssociated(DCFieldContainer *in*) Verifies if the record contains any associated block.
IntField	hasAssociatedData(DCFieldContainer *in*, StringField *associatedBlockName*) Verifies if the record contains the associated block *associatedBlockName*.
ListField	getAssociatedData(DCFieldContainer *in*, StringField *associatedBlockName*) Returns the *associatedBlockName* as a Listfield
IntField	getAssociatedIntField(DCFieldContainer *in*, StringField *associatedBlockName*, StringField *fieldName*) Returns the integer field *fieldName* from the associated block *associatedBlockName*.
DoubleField	getAssociatedDoubleField(DCFieldContainer *in*, StringField *associatedBlockName*, StringField *fieldName*) Returns the double field *fieldName* from the associated block *associatedBlockName*.
LongField	getAssociatedLongField(DCFieldContainer *in*, StringField *associatedBlockName*, StringField *fieldName*) Returns the long field *fieldName* from the associated block *associatedBlockName*.

Table 2–1 (Cont.) Oracle CDR Format Java Hooks Method Summary

Modifier and Type	Method and Description
StringField	getAssociatedLongField(DCFieldContainer *in*, StringField *associatedBlockName*, StringField *fieldName*) Returns the string field *fieldName* from the associated block *associatedBlockName*.
IntField	getIntFieldFromList(ListField *lstFld*, StringField *fieldName*) Returns the integer field *fieldName* from *lstFld*.
LongField	getLongFieldFromList(ListField *lstFld*, StringField *fieldName*) Returns the long field *fieldName* from *lstFld*.
DoubleField	getDoubleFieldFromList(ListField *lstFld*, StringField *fieldName*) Returns the double field *fieldName* from *lstFld*.
StringField	getStringFieldFromList(ListField *lstFld*, StringField *fieldName*) Returns the string field *fieldName* from *lstFld*.

Oracle CDR Format Java Hook Method Details

The section describes the Oracle CDR Format Java hook methods.

hasHeaderFields

```
IntField hasHeaderFields(DCFieldContainer in)
```

Usage

This function verifies if the record contains header fields.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

Returns

1 (true) if the record contains header fields.

0 (false) if the record does not contain header fields.

hasTrailerFields

```
IntField hasTrailerFields(DCFieldContainer in)
```

Usage

This function verifies if the record contains trailer fields.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

Returns

1 (true) if the record contains trailer fields.

0 (false) if the record does not contain trailer fields.

hasAnyAssociated

```
IntField hasAnyAssociated(DCFieldContainer in)
```

Usage

This function verifies if the record contains any associated block.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

Returns

1 (true) if the record contains the associated block.

0 (false) if the record does not contain the associated block.

hasAssociatedData

```
IntField hasAssociatedData(DCFieldContainer in, StringField associatedBlockName)
```

Usage

This function verifies if the record contains the associated block *associatedBlockName*.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

Returns

1 (true) if the record contains the associated block.

0 (false) if the record does not contain the associated block.

getAssociatedData

```
ListField getAssociatedData(DCFieldContainer in, StringField associatedBlockName)
```

Usage

This function returns the *associatedBlockName* as a Listfield.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

Returns

The *associatedBlockName* as a List field if the value is found. Returns an empty List field if the value is not found.

getAssociatedIntField

```
IntField getAssociatedIntField(DCFieldContainer in,StringField
associatedBlockName,StringField fieldName)
```

Usage

This function returns the integer field *fieldName* from the associated block *associatedBlockName*.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

fieldName is the field name for which the value is to be returned.

Returns

The integer field matching the search criteria.

getAssociatedDoubleField

```
DoubleField getAssociatedDoubleField(DCFieldContainer in,StringField
associatedBlockName,StringField fieldName)
```

Usage

This function returns the double field *fieldName* from the associated block *associatedBlockName*.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

fieldName is the field name for which the value is to be returned.

Returns

The double field matching the search criteria.

getAssociatedLongField

```
LongField getAssociatedLongField(DCFieldContainer in,StringField
associatedBlockName,StringField fieldName)
```

Usage

This function returns the long field *fieldName* from the associated block *associatedBlockName*.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

fieldName is the field name for which the value is to be returned.

Returns

The long field matching the search criteria.

getAssociatedStringField

```
StringField getAssociatedStringField(DCFieldContainer in,StringField
associatedBlockName,StringField fieldName)
```

Usage

This function returns the string field *fieldName* from the associated block *associatedBlockName*.

Parameters

in is the DCFieldContainer that contains the field names and their associated values.

associatedBlockName is the name of the associated block.

fieldName is the field name for which the value is to be returned.

Returns

The string field matching the search criteria.

getIntFieldFromList

```
IntField getIntFieldFromList(ListField lstFld, StringField fieldName)
```

Usage

This function returns the integer field *fieldName* from *lstFld*.

Parameters

lstFld is the ListField that contains the field names and values.

fieldName is the field name for which the value is to be returned.

Returns

The integer field matching the search criteria.

getLongFieldFromList

```
LongField getLongFieldFromList(ListField lstFld, StringField fieldName)
```

Usage

This function returns the long field *fieldName* from *lstFld*.

Parameters

lstFld is the ListField that contains the field names and values.

fieldName is the field name for which the value is to be returned.

Returns

The long field matching the search criteria.

getDoubleFieldFromList

```
DoubleField getDoubleFieldFromList(ListField lstFld, StringField fieldName)
```

Usage

This function returns the double field *fieldName* from *lstFld*.

Parameters

lstFld is the ListField that contains the field names and values.

fieldName is the field name for which the value is to be returned.

Returns

The double field matching the search criteria.

getStringFieldFromList

```
StringField getStringFieldFromList(ListField lstFld, StringField fieldName)
```

Usage

This function returns the string field *fieldName* from *lstFld*.

Parameters

lstFld is the ListField that contains the field names and values.

fieldName is the field name for which the value is to be returned.

Returns

The string field matching the search criteria.

3

Working with Record Enhancement Charging Java Hooks in NPL

This chapter lists and describes the Java hooks available for the Oracle Communications Offline Mediation Controller record enhancement charging Enhancement Processor (EP) cartridge.

About Record Enhancement Charging Java Hooks

Java hooks are an advanced feature of NPL (Node Programming Language) that enable Offline Mediation Controller to call a Java method from an NPL program. For more information on using Java hooks with NPL, see the discussion on Java hooks in *Offline Mediation Controller Cartridge Development Kit NPL Reference Guide*.

Table 3–1 lists the record enhancement charging Java hooks methods.

Table 3–1 Record Enhancement Charging Java Hooks Method Summary

Modifier and Type	Method and Description
void	load() throws NodeProcessingException Loads data from the database into memory.
IntField	exists(StringField *key*) throws NodeProcessingException Searches for a key in the data collection for the configuration service.
MapField	get(StringField *key*) throws NodeProcessingException Searches for the field that contains the record that matches *key*.
StringField	getLoadedInfo() throws NodeProcessingException Searches for the string that represents the cached data and time it was cached.
StringField	getMapField(MapField *map*, StringField *fieldName*) throws NodeProcessingException Searches for the value of *fieldName* in *map*.
IntField	isEmpty(DCField *field*) throws NodeProcessingException Verifies if *field* contains any values.
IntField	TRUE() throws NodeProcessingException Use this method instead of checking if the return value is **1** (true).
IntField	FALSE() throws NodeProcessingException Use this method instead of checking if the return value is **0** (false).
StringField	VALUE(IntField *val*) throws NodeProcessingException Use this method to return the string representation of *val*.

Table 3–1 (Cont.) Record Enhancement Charging Java Hooks Method Summary

Modifier and Type	Method and Description
MapField	search(StringField *areacode*) throws NodeProcessingException Searches for the longest best match in the cached data for *areacode*.
MapField	search(StringField *mapGroup*, StringField *extServicecode*, StringField *usageclass*, StringField *locarindVasevent*, StringField *qosRequested*, StringField *qosUsed*, StringField *recordtype*) throws NodeProcessingException Searches for the first ranked record matching the search criteria.
MapField	search(StringField *mapGroup*, StringField *extUsageclass*, StringField *usagetype*, StringField *zoneWs*, StringField *tariffclass*, StringField *tariffsubclass*, StringField *recordtype*, StringField *connecttype*, StringField *connectsubtype*, StringField *transitAreacode*, StringField *apnAddress*, StringField *ssPacket*) throws NodeProcessingException Searches for the first ranked record matching the search criteria.
MapField	search(StringField *apnGroup*, StringField *servicecode*, StringField *accesspointname*) throws NodeProcessingException Searches for the first ranked record matching the search criteria for the access point name (APN) group.
MapField	getByNo(IntField *no*) throws NodeProcessingException Searches for the network operator record having the internal ID *no*.

Record Enhancement Charging Java Hook Method Details

The section describes the record enhancement charging Java hook methods.

load

```
void load() throws NodeProcessingException
```

Usage

This function loads data from the database into memory.

Parameters

This method has no parameters.

Returns

This function returns nothing.

exists

```
IntField exists(StringField key) throws NodeProcessingException
```

Usage

This function searches for a key in the data collection for the configuration service.

Parameters

key is the key field in the record to search for.

Returns

1 (true) if the key is found in the record.

0 (false) if the key is not found in the record.

get

```
MapField get(StringField key) throws NodeProcessingException)
```

Usage

This function searches for the record that matches *key*. The database column name (case insensitive) is used as the field name in the MapField. For Service Code Map and Usage Class Map, which are keyed by the map_group, the first ranked record is returned if found.

Parameters

key is the key field in the record.

Returns

The record that matches the key.

getLoadedInfo

```
StringField getLoadedInfo() throws NodeProcessingException)
```

Usage

This function searches for the string that represents the cached data and time it was cached.

Parameters

This method has no parameters.

Returns

The string that represents the cached data and the time the data was cached.

getMapField

```
StringField getMapField(MapField map, StringField fieldName) throws
NodeProcessingException)
```

Usage

This function searches for the value of *fieldName* in *map*.

Parameters

map is the MapField in which the value for *fieldName* is to be found.

fieldName is the field name for which the value is to be returned.

Returns

The string associated with *fieldName*.

An empty string ("") is returned if the field is not found.

isEmpty

```
IntField isEmpty(DCField field) throws NodeProcessingException)
```

Usage
This function verifies if *field* contains any values.

Parameters
field is the DCField that contains the field type and field value.

Returns
1 (true) if the field is empty.

0 (false) if the field is not empty.

TRUE

```
IntField TRUE() throws NodeProcessingException)
```

Usage
This function can be used to verify that the return value is **1** (true).

Parameters
This method has no parameters.

Returns
1 (true) if the return value is true.

FALSE

```
IntField FALSE() throws NodeProcessingException)
```

Usage
This function can be used to verify that the return value is **0** (false).

Parameters
This method has no parameters.

Returns
0 (false) if the return value is false.

VALUE

```
StringField VALUE(IntField val) throws NodeProcessingException)
```

Usage
This function returns the string representation of *val*.

Parameters
val is the IntField to be converted from an integer to a string.

Returns
The string representation of *val*.

search

```
MapField search(StringField areacode) throws NodeProcessingException)
```

Usage

This function searches for the longest best match in the cached data for *areacode*.

Parameters

areacode is the area code to search.

Returns

Returns the longest best match for *areacode*.

search

```
MapField search(StringField mapGroup, StringField extServicecode, StringField
usageclass, StringField locarindVasevent, StringField qosRequested, StringField
qosUsed, StringField recordtype) throws NodeProcessingException
```

Usage

This function searches for the first ranked record matching the search criteria.

Parameters

mapGroup is the map group.

extServicecode is the name of the external service code map.

usageclass is the name of the usage class map.

locarindVasevent is the MSC responsible for handling the call and the location of the equipment making or receiving the call.

qosRequested is the type of QoS requested.

qosUsed the type of QoS negotiated by the network.

recordtype is the record type.

Returns

The first ranked record matching the search criteria.

search

```
MapField search(StringField mapGroup, StringField extUsageclass, StringField
usagetype, StringField zoneWs, StringField tariffclass, StringField
tariffsubclass, StringField recordtype, StringField connecttype, StringField
connectsubtype, StringField transitAreacode, StringField apnAddress, StringField
ssPacket) throws NodeProcessingException
```

Usage

This function searches for the first ranked record matching the search criteria.

Parameters

mapGroup is the map group.

extUsageclass is the external usage class.

usagetype is the customer-related usage scenario.

zoneWs is the impact category for wholesale zone.

tariffclass is the tariff class that contains the tariff information.

tariffsubclass is the detailed tariff information.

recordtype is the record type.

connecttype is the type of connection.

connectsubtype is the detailed description of the connection or call type.

transitAreacode is the area code.

apnAddress is the logical name of the connected access point to the external packet data network.

ssPacket is the number of supplementary service records.

Returns

The first record matching the search criteria.

search

```
MapField search(StringField apnGroup, StringField servicecode, StringField
accesspointname) throws NodeProcessingException
```

Usage

This function searches for the first ranked record matching the search criteria for the access point name (APN) group.

Parameters

apnGroup is the APN group.

servicecode is the service code.

accesspointname is the APN name.

Returns

The first record matching the search criteria for the APN.

getByNo

```
MapField getByNo(IntField no) throws NodeProcessingException
```

Usage

This function searches for the network operator record having the internal ID *no*.

Parameters

no is the internal ID for the network operator.

Returns

The network operator record having the internal ID *no*.

4

Sample Mapping for ECE Cartridge Pack

This chapter contains the sample mapping for the Oracle Communications Offline Mediation Controller Elastic Charging Engine (ECE) Distribution Cartridge (DC). The mapping includes:

- The input attribute-value pairs (AVPs) in the call detail records (CDRs)
- The values in the ECE payload specification files
- The ECE usage requests and payload parameters Offline Mediation Controller uses to build the usage request to send to ECE

Supported Usage Types

The ECE DC supports the following usage types for offline usage requests:

- Terminate
- Update
- Refund_Unit
- Refund_Amount
- Cancel
- Debit_unit
- Debit_amount

> **Note:** For refund usage types, the input CDR must contain the **correlation_identifier** value of the original debit request.

Mapping for ASCII

By default, the ECE DC supports the following ASCII product types:

- Voice
- Data
- SMS

Voice

Product Type = VOICE

Event Type = USAGE

Usage Type = Terminate

Table 4–1 Sample ASCII Voice Mapping for the Terminate Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
duration	USED_UNITS[0].DURATION	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number

Product Type = VOICE

Event Type = USAGE

Usage Type = Update

Table 4–2 Sample ASCII Voice Mapping for the Update Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
Req_Duration	REQUESTED_UNITS[0].DURATION	
Used_Duration	USED_UNITS[0].DURATION	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number

Product Type = VOICE

Event Type = USAGE

Usage Type = Refund_Unit

Table 4–3 Sample ASCII Voice Mapping for Refund_Unit Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
Used_duration	USED_UNITS[0].DURATION	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number
correlation_identifier	CORRELATION_IDENTIFIER	

Product Type = VOICE

Event Type = USAGE

Usage Type = Refund_Amount

Table 4–4 Sample ASCII Voice Mapping for the Refund_Amount Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number
balance_element_id	IMPACT_AMOUNT[0].BALANCE_ELEMENT_ID	
amount	IMPACT_AMOUNT[0].AMOUNT	
correlation_identifier	CORRELATION_IDENTIFIER	

Data

Product Type = DATA

Event Type = USAGE

Usage Type = Terminate

Table 4–5 Sample ASCII Data Mapping for the Terminate Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cell_id	CELL_ID	
usedUnitsInputVolume	USED_UNITS[0].INPUT_VOLUME	
usedUnitsOutputVolume	USED_UNITS[0].OUTPUT_VOLUME	
usedUnitsTotalVolume	USED_UNITS[0].TOTAL_VOLUME	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number

Product Type = DATA

Event Type = USAGE

Usage Type = Update

Table 4–6 Sample ASCII Data Mapping for the Update Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cell_id	CELL_ID	
requestedInputVolume	REQUESTED_UNITS[0].INPUT_VOLUME	
requestedOutputVolume	REQUESTED_UNITS[0].OUTPUT_VOLUME	
requestedTotalVolume	REQUESTED_UNITS[0].TOTAL_VOLUME	
usedUnitsInputVolume	USED_UNITS[0].INPUT_VOLUME	
usedUnitsOutputVolume	USED_UNITS[0].OUTPUT_VOLUME	
usedUnitsTotalVolume	USED_UNITS[0].TOTAL_VOLUME	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number

Product Type = DATA

Event Type = USAGE

Usage Type = Refund_Unit

Table 4–7 Sample ASCII Data Mapping for the Refund_Unit Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cell_id	CELL_ID	
usedUnitsInputVolume	USED_UNITS[0].INPUT_VOLUME	
usedUnitsOutputVolume	USED_UNITS[0].OUTPUT_VOLUME	
usedUnitsTotalVolume	USED_UNITS[0].TOTAL_VOLUME	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number
correlation_identifier	CORRELATION_IDENTIFIER	

Product Type = DATA

Event Type = USAGE

Usage Type = Refund_Amount

Table 4–8 Sample ASCII Data Mapping for the Refund_Amount Usage Type

ASCII NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cell_id	CELL_ID	
CalledId		CALLED_ID
calling_number		UserIdentity
session_id		Session Id
start_time		Start time
end_time		End time
seq_no		Seq number
balance_element_id	IMPACT_AMOUNT[0].BALANCE_ELEMENT_ID	
amount	IMPACT_AMOUNT[0].AMOUNT	
correlation_identifier	CORRELATION_IDENTIFIER	

Mapping for SGSN

The ECE DC supports the following SGSN product types:

- Data
- SMS

Data

SGSN PDP Record

Table 4–9 Sample SGSN Data Mapping for the SGSN PDP Record

ASN 1 Fields	SGSN Cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
listOfTrafficVolumes.dataVolumeGPRSUplink	part of 20209	USED_UNITS[0].OUTPUT_VOLUME	
listOfTrafficVolumes.dataVolumeGPRSDownlink	part of 20209	USED_UNITS[0].INPUT_VOLUME	
cellIdentifier	20223(CellIdentifier)	CELL_ID	
servedIMSI	20234 (ServedIMSI)		UserIdentity
chargingID	20001(ChargingID)		Session id
recordOpeningTime			start time
duration	duration		end_time(calculated from the duration)
recordSequenceNumber	20005 (record sequence number)		sequence number

S-GW Record

Table 4–10 Sample SGSN Data Mapping for the S-GW Record

ASN1 Fields	SGSN	USED_ UNITS[0].OUTPUT_ VOLUME	
listOfTrafficVolumes.data VolumeGPRSUplink	part of 20209	USED_ UNITS[0].OUTPUT_ VOLUME	
listOfTrafficVolumes.data VolumeGPRSDownlink	part of 20209	USED_UNITS[0].INPUT_ VOLUME	
servedIMSI	20234 (ServedIMSI)		UserIdentity
chargingID	20001(ChargingID)		Session id
startTime	20274(StartTime)		Start time
stopTime	20275(StopTime) if it exists, otherwise derived from duration(20004)		End time
recordSequenceNumber	20005 (record sequence number)		sequence number

P-GW Record

Table 4–11 Sample SGSN Data Mapping for P-GW Record

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
listOfServiceData.datavol umeFBCUplink	part of 20267	USED_ UNITS[0].OUTPUT_ VOLUME	
listOfService.datavolumeF BCDownlink	part of 20267	USED_UNITS[0].INPUT_ VOLUME	
servedIMSI	20234 (ServedIMSI)		UserIdentity
chargingID	20001(ChargingID)		Session id
startTime	20274		Start time
stopTime	20275(StopTime) if it exists, otherwise derived from duration(20004)		End time
recordSequenceNumber	20005 (record sequence number)		sequence number

SMS

The SGSNMMRecord (mobility management), SGSNMORecord (Mobile originated) and SGSNMTRecord (Mobile Terminated) record types are supported for Mobility management and SMS services.

20234 (ServedIMSI)

20219

Product Type = SMS

Event Type = USAGE

Usage Type = Terminate

Table 4–12 Sample SGSN SMS Mapping for the Terminate Usage Type

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cellIdentifier	20223(CellIdentifier)	CELL_ID	
destinationNumber	if SGSNMORecord 20303(DestinationNumber), if SGSNMTRecord same as 20234(servedIMSI)	CALLED_ID	
NA		USED_ UNITS[0].SPECIFIC_ UNIT	
servedMSISDN or servedIMSI			UserIdentity
chargingID	20001(ChargingID)		Session id
eventTimeStamp	20240(EventTimeStamp)		Start time
eventTimeStamp	20240(EventTimeStamp)		end time
localSequenceNumber			seq number

Product Type = SMS

Event Type = USAGE

Usage Type = Update

Table 4–13 Sample SGSN SMS Mapping for the Update Usage Type

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cellIdentifier	20223(CellIdentifier)	CELL_ID	
destinationNumber	if SGSNMORecord 20303(DestinationNumber), if SGSNMTRecord same as 20234(servedIMSI)	CALLED_ID	
NA		USED_ UNITS[0].SPECIFIC_ UNIT	
servedMSISDN or servedIMSI			UserIdentity
chargingID	20001(ChargingID)		Session id
eventTimeStamp	20240(EventTimeStamp)		Start time
eventTimeStamp	20240(EventTimeStamp)		end time
localSequenceNumber			seq number

Product Type = SMS

Event Type = USAGE

Usage Type = Refund_Unit

Table 4–14 Sample SGSN SMS Mapping for the Refund_Unit Usage Type

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cellIdentifier	20223(CellIdentifier)	CELL_ID	
destinationNumber	if SGSNMORecord 20303(DestinationNumber), if SGSNMTRecord same as 20234(servedIMSI)	CALLED_ID	
NA		USED_ UNITS[0].SPECIFIC_ UNIT	
servedMSISDN or servedIMSI			UserIdentity
chargingID	20001(ChargingID)		Session id
eventTimeStamp	20240(EventTimeStamp)		Start time
eventTimeStamp	20240(EventTimeStamp)		end time
localSequenceNumber			seq number
correlation_identifier		CORRELATION_ IDENTIFIER	

Product Type = SMS

Event Type= USAGE

Usage Type = Refund_Amount

Table 4–15 Sample SGSN SMS Mapping for the Refund_Amount Usage Type

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
cellIdentifier	20223(CellIdentifier)	CELL_ID	
destinationNumber	if SGSNMORecord 20303(DestinationNumber), if SGSNMTRecord same as 20234(servedIMSI)	CALLED_ID	
NA		USED_ UNITS[0].SPECIFIC_ UNIT	
servedMSISDN or servedIMSI			UserIdentity
chargingID	20001(ChargingID)		Session id
eventTimeStamp	20240(EventTimeStamp)		Start time
eventTimeStamp	20240(EventTimeStamp)		end time
localSequenceNumber			seq number

Table 4–15 (Cont.) Sample SGSN SMS Mapping for the Refund_Amount Usage Type

ASN1 Fields	SGSN cartridge NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
balance_element_id		IMPACT_ AMOUNT[0].BALANCE_ ELEMENT_ID	
amount		IMPACT_ AMOUNT[0].AMOUNT	
correlation_identifier		CORRELATION_ IDENTIFIER	

Mapping for IMS

The ECE DC supports the following IMS product types:

- Voice

In case of IMS cartridge pack, the input is a Diameter ACR request, which the cartridge pack node chains convert into the 3GPP ASN1 format. The converted ASN1 format is the input to the ECE DC.

Voice

Table 4–16 Sample IMS Voice Mapping

ASN1 Fields	IMS cartridge NPL Fields	ECE Payload Fields
Called_Party_Address	Called_Party_Address	CALLED_ID
derived using service delivery start time stamp and end time stamp	Duration	REQUESTED_UNITS[0].DURATION
PrivateUserID	PrivateUserID	UserIdentity
ServiceDeliveryStartTimeStamp	ServiceDeliveryStartTimeStamp	start time
ServiceDeliveryEndTimeStamp	ServiceDeliveryEndTimeStamp	end time
Session_Id	Session_Id	session id
	seqNo	seq number

Mapping for Oracle CDR Format

The ECE DC supports the following Oracle CDR format product types:

- Voice
- Data
- SMS
- TelcoGsmTelephony
- TelcoGprs

Voice

Product Type = VOICE

Event Type = USAGE

Usage Type = Terminate

Table 4–17 Sample Oracle CDR Format Voice Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
A_NUMBER		UserIdentity
		Seq number
		Session Id
CHARGING_START_ TIMESTAMP		Start time
CHARGING_END_ TIMESTAMP		End time
BASIC_SERVICE		cdr_service
DURATION	USED_UNITS[0].DURATION	
B_NUMBER	CALLED_ID	
CELL_ID	CELL_ID	
USAGE_DIRECTION	USAGE_DIRECTION	
CALL_COMPLETION_ INDICATOR	TERMINATION_CAUSE	
QOS_USED	QUALITY_OF_SERVICE	

Data

Product Type = DATA

Event Type = DATA_USAGE

Usage Type = Terminate

Table 4–18 Sample Oracle CDR Format Data Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
A_NUMBER		UserIdentity
		Seq number
		Session Id
CHARGING_START_TIMESTAMP		Start time
CHARGING_END_TIMESTAMP		End time
BASIC_SERVICE		cdr_service
VOLUME_SENT	USED_UNITS[0].INPUT_VOLUME	
VOLUME_SENT + VOLUME_ RECEIVED	USED_UNITS[0].TOTAL_VOLUME	
	USED_UNITS[0].CC_TIME	
CHARGING_START_TIMESTAMP	USED_UNITS[0].VALIDITY_START	
CHARGING_END_TIMESTAMP	USED_UNITS[0].VALIDITY_END	

Table 4–18 (Cont.) Sample Oracle CDR Format Data Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
B_NUMBER	CALLED_ID	
CELL_ID	CELL_ID	
CALL_COMPLETION_ INDICATOR	TERMINATION_CAUSE	

SMS

Product Type = SMS

Event Type = SMS_USAGE

Usage Type = Terminate

Table 4–19 Sample Oracle CDR Format SMS Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
A_NUMBER		UserIdentity
		Seq number
		Session Id
CHARGING_START_TIMESTAMP		Start time
CHARGING_END_TIMESTAMP		End time
BASIC_SERVICE		cdr_service
DURATION	USED_UNITS[0].SPECIFIC_UNIT	
CALL_COMPLETION_INDICATOR	TERMINATION_CAUSE	
B_NUMBER	CALLED_ID	
CELL_ID	CELL_ID	

TelcoGsmTelephony

Product Type = TelcoGsmTelephony

Event Type = ConvergentVoice

Usage Type = Terminate

Table 4–20 Sample Oracle CDR Format TelcoGsmTelephony Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
A_NUMBER		UserIdentity
		Seq number
		Session Id
CHARGING_START_TIMESTAMP		Start time
CHARGING_END_TIMESTAMP		End time
BASIC_SERVICE		cdr_service

Table 4–20 (Cont.) Sample Oracle CDR Format TelcoGsmTelephony Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
CALL_COMPLETION_INDICATOR	TERMINATION_CAUSE	
	ZONE_ORIGIN	
	ZONE_DEST	
USAGE_TYPE	USAGE_TYPE	
B_NUMBER	CALLED_ID	
	TELCO_INFO[0].NETWORK_ SESSION_ID	
	TELCO_INFO[0].NETWORK_ SESSION_CORRELATION_ID	
DESTINATION_NETWORK	TELCO_INFO[0].DESTINATION_ NETWORK	
MSID	TELCO_INFO[0].PRIMARY_MSID	
	TELCO_INFO[0].SECONDARY_ MSID	
A_NUMBER	TELCO_INFO[0].CALLING_FROM	
B_NUMBER	TELCO_INFO[0].CALLED_TO	
	TELCO_INFO[0].SVC_TYPE	
	TELCO_INFO[0].SVC_CODE	
USAGE_CLASS	TELCO_INFO[0].USAGE_CLASS	
VOLUME_RECEIVED	TELCO_INFO[0].BYTES_ DOWNLINK	
VOLUME_SENT	TELCO_INFO[0].BYTES_UPLINK	
ACTION_CODE	SERVICE_CODES[0].SS_ACTION_ CODE	
SS_CODE	SERVICE_CODES[0].SS_CODE	
ESN_IMEI	GSM_INFO[0].IMEI	
B_MODIFICATION_INDICATOR	GSM_INFO[0].CALLED_NUM_ MODIF_MARK	
USAGE_DIRECTION or CALL_ DIRECTION	GSM_INFO[0].DIRECTION	
B_NUMBER	GSM_INFO[0].DIALED_NUMBER	
QOS_REQUESTED	GSM_INFO[0].QOS_REQUESTED	
QOS_USED	GSM_INFO[0].QOS_NEGOTIATED	
LONG_DURATION_INDICATOR	GSM_INFO[0].SUB_TRANS_ID	
HOME_CARRIER_SID	GSM_INFO[0].ORIGIN_SID	
HOME_CARRIER_SID	GSM_INFO[0].DESTINATION_SID	
LOCATION_AREA_INDICATOR	GSM_INFO[0].LOC_AREA_CODE	
CELL_ID	GSM_INFO[0].CELL_ID	
VOLUME_RECEIVED	GSM_INFO[0].BYTES_IN	
VOLUME_SENT	GSM_INFO[0].BYTES_OUT	

Table 4–20 (Cont.) Sample Oracle CDR Format TelcoGsmTelephony Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
USAGE_CLASS	GSM_INFO[0].USAGE_CLASS	
NUMBER_OF_UNITS	GSM_INFO[0].NUMBER_OF_UNIT	
DURATION	USED_UNITS[0].DURATION	
	USED_UNITS[0].SPECIFIC_UNIT	

TelcoGprs

Product Type = TelcoGprs

Event Type = ConvergentData

Usage Type = Terminate

Table 4–21 Sample Oracle CDR Format TelcoGprs Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
A_NUMBER		UserIdentity
		Seq number
		Session Id
CHARGING_START_TIMESTAMP		Start time
CHARGING_END_TIMESTAMP		End time
BASIC_SERVICE		cdr_service
CALL_COMPLETION_INDICATOR	TERMINATION_CAUSE	
USAGE_TYPE	USAGE_TYPE	
B_NUMBER	CALLED_ID	
	TELCO_INFO[0].NETWORK_ SESSION_ID	
	TELCO_INFO[0].NETWORK_ SESSION_CORRELATION_ID	
DESTINATION_NETWORK	TELCO_INFO[0].DESTINATION_ NETWORK	
MSID	TELCO_INFO[0].PRIMARY_MSID	
PORT_NUMBER	TELCO_INFO[0].SECONDARY_ MSID	
A_NUMBER	TELCO_INFO[0].CALLING_FROM	
B_NUMBER	TELCO_INFO[0].CALLED_TO	
	TELCO_INFO[0].SVC_TYPE	
	TELCO_INFO[0].SVC_CODE	
USAGE_CLASS	TELCO_INFO[0].USAGE_CLASS	
VOLUME_RECEIVED	TELCO_INFO[0].BYTES_ DOWNLINK	
VOLUME_SENT	TELCO_INFO[0].BYTES_UPLINK	

Table 4–21 (Cont.) Sample Oracle CDR Format TelcoGprs Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
ROUTING_AREA	GPRS_INFO[0].ROUTING_AREA	
LOCATION_AREA_INDICATOR	GPRS_INFO[0].LOC_AREA_CODE	
CELL_ID	GPRS_INFO[0].CELL_ID	
SESSION_ID	GPRS_INFO[0].SESSION_ID	
SGSN_ADDRESS	GPRS_INFO[0].SGSN_ADDRESS	
NODE_ID	GPRS_INFO[0].NODE_ID	
TRANS_ID	GPRS_INFO[0].TRANS_ID	
	GPRS_INFO[0].EXTENSIONS	
	GPRS_INFO[0].STATUS	
NETWORK_INITIATED_PDP	GPRS_INFO[0].NI_PDP	
CONNECT_TYPE	GPRS_INFO[0].ANONYMOUS_ LOGIN	
PDP_TYPE	GPRS_INFO[0].PDP_TYPE	
PDP_ADDRESS	GPRS_INFO[0].PDP_ADDRESS	
PDP_REMOTE_ADDRESS	GPRS_INFO[0].PDP_RADDRESS	
PDP_DYNAMIC_ADDRESS	GPRS_INFO[0].PDP_DYNADDR	
DIAGNOSTICS	GPRS_INFO[0].DIAGNOSTICS	
APN_ADDRESS	GPRS_INFO[0].APN	
NUMBER_OF_UNITS	GPRS_INFO[0].NUMBER_OF_UNITS	
NETWORK_CAPABILITY	GPRS_INFO[0].NETWORK_ CAPABILITY	
SGSN_CHANGE	GPRS_INFO[0].SGSN_CHANGE	
CHANGE_CONDITION	GPRS_INFO[0].CHANGE_ CONDITION	
QOS_REQUESTED_PRECEDENCE	GPRS_INFO[0].QOS_REQ_ PRECEDENCE	
QOS_REQUESTED_DELAY	GPRS_INFO[0].QOS_REQ_DELAY	
QOS_REQUESTED_RELIABILITY	GPRS_INFO[0].QOS_REQ_ RELIABILITY	
QOS_REQUESTED_PEAK_ THROUGHPUT	GPRS_INFO[0].QOS_REQ_PEAK_ THROUGH	
QOS_REQUESTED_MEAN_ THROUGHPUT	GPRS_INFO[0].QOS_REQ_MEAN_ THROUGH	
QOS_USED_PRECEDENCE	GPRS_INFO[0].QOS_NEGO_ PRECEDENCE	
QOS_USED_DELAY	GPRS_INFO[0].QOS_NEGO_DELAY	
QOS_USED_RELIABILITY	GPRS_INFO[0].QOS_NEGO_ RELIABILITY	
QOS_USED_PEAK_THROUGHPUT	GPRS_INFO[0].QOS_NEGO_PEAK_ THROUGH	

Table 4–21 (Cont.) Sample Oracle CDR Format TelcoGprs Mapping for the Terminate Usage Type

Oracle CDR Format NPL Fields	ECE Payload Fields	ECE Usage Object Builder Parameter
QOS_USED_MEAN_ THROUGHPUT	GPRS_INFO[0].QOS_NEGO_MEAN_ THROUGH	
	GPRS_INFO[0].SGSN_PLMN_ID	
	GPRS_INFO[0].MM_STATE	
	GPRS_INFO[0].PTMSI	
	GPRS_INFO[0].PTMSI_SIGNATURE	
	GPRS_INFO[0].SERVICE_AREA_ CODE	
	GPRS_INFO[0].EXT_CHARGING_ID	
	GPRS_INFO[0].PS_CHARGING_ DESCR	
VOLUME_SENT	USED_UNITS[0].INPUT_VOLUME	
VOLUME_RECEIVED	USED_UNITS[0].OUTPUT_VOLUME	
VOLUME_SENT + VOLUME_ RECEIVED	USED_UNITS[0].TOTAL_VOLUME	

A

NPL Syntax and Reserved Words

This appendix provides the NPL syntax and reserved words.

Table A–1 lists all the NPL methods and the corresponding syntax.

EBNF for NPL

Table A–1 NPL Methods, Operator and Syntax

Method	Operator	Syntax
Program	::=	(MethodHandlerImportDecl)? (ConfigDeclaration)? (VariableDeclaration)+ (ExposeDeclaration)* (Statement) +<EOF>
MethodHandlerImportDecl	::=	"import" <IDENTIFIER> ("." <IDENTIFIER>)* ";"
ConfigDeclaration	::=	"Config" "{" (ConfigAttribute)? "}"
ConfigAttribute	::=	<IDENTIFIER> <STRING_LITERAL> ";"
VariableDeclaration	::=	CompositeVariableDecl | SimpleVariableDecl
CompositeVariableDecl	::=	InputRecDecl | OutputRecDecl
InputRecDecl	::=	"InputRec" "{" (AttributeVariableDecl)+ "}" <IDENTIFIER> ";"
OutputRecDecl	::=	"OutputRec" "{" (AttributeVariableDecl)+ "}" <IDENTIFIER> ";"
AttributeVariableDecl	::=	AttributeType (<ATTRIBUTE_NAME> | <IDENTIFIER> | <INTEGER_LITERAL>) ";"
SimpleVariableDecl	::=	AttributeType <IDENTIFIER> (SimpleVariableInit)? ";"
SimpleVariableInit	::=	"=" Expression

Table A–1 (Cont.) NPL Methods, Operator and Syntax

Method	Operator	Syntax
AttributeType	::=	"String" | "Integer" | "Long" | "TimeInSecs" | "IP" | "TimeInMilliSecs" | "MAC" | "UInt128" | "Byte" | "Short" | "IPv6" | "Float" | "Double" | "Bytes" | "Object" | "List"
ExposeDeclaration	::=	"Expose for" <IDENTIFIER> "{" (ExposeAttribute)+ "}"
ExposeAttribute	::=	<IDENTIFIER> "." (<ATTRIBUTE_NAME> | <IDENTIFIER> | <INTEGER_LITERAL>) (<STRING_LITERAL>)? ";"
Statement	::=	IfStatement | AssignmentStmt | FunctionStmt | JavaHookStmt
IfStatement	::=	"if" "(" ConditionalExpression ")" IfAction ("else" "if" "(" ConditionalExpression ")" IfAction)* ("else" IfAction)?
ConditionalExpression	::=	RelationalExpression (ConditionalOperator RelationalExpression)*
ConditionalOperator	::=	<OR> | <AND>
RelationalExpression	::=	NestedRelationalExpression | SimpleRelationalExpression | BooleanFunctionExpr
BooleanFunctionExpr	::=	FunctionExpression
NestedRelationalExpression	::=	"(" ConditionalExpression ")"
SimpleRelationalExpression	::=	Expression RelationalOperator Expression
RelationalOperator	::=	<EQ> | <NE> | <LT> | <LE> | <GT> | <GE>
IfAction	::=	StatementBlock | Statement
StatementBlock	::=	"{" (Statement)* "}"
AssignmentStmt	::=	Variable "=" Expression ";"

Table A–1 (Cont.) NPL Methods, Operator and Syntax

Method	Operator	Syntax
Variable	::=	<IDENTIFIER> ("." (<ATTRIBUTE_NAME> | <IDENTIFIER> | <INTEGER_LITERAL>))?
Expression	::=	AdditiveExpression
AdditiveExpression	::=	SubtractiveExpression ("+" SubtractiveExpression)*
SubtractiveExpression	::=	MultiplicativeExpression ("-" MultiplicativeExpression)*
MultiplicativeExpression	::=	DivisionalExpression ("*" DivisionalExpression)*
DivisionalExpression	::=	UnaryExpression ("/" UnaryExpression)*
UnaryExpression	::=	("(" AttributeType ")")? (NestedExpression | VariableExpression | IntLiteralExpression | FloatingPointLiteralExpression | StringLiteralExpression | FunctionExpression | JavaHookExpression)
NestedExpression	::=	(UnaryOperator)? "(" Expression ")"
VariableExpression	::=	(UnaryOperator)? Variable
IntLiteralExpression	::=	(UnaryOperator)? <STRING_LITERAL>
FloatingPointLiteralExpression	::=	(UnaryOperator)? <FLOATING_POINT_LITERAL>
StringLiteralExpression	::=	<STRING_LITERAL>

Table A–1 (Cont.) NPL Methods, Operator and Syntax

Method	Operator	Syntax
FunctionExpression	::=	("byte2bytes" | "byte2str" | "bytes2byte" | "bytes2double" | "bytes2float" | "bytes2hexstr" | "bytes2int" | "bytes2IP" | "bytes2long" | "bytes2short" | "bytes2str" | "bytes2TimeInMilliSecs" | "bytes2TimeInSecs" | "clone" | "convertDateTime" | "copyBits" | "currentTime" | "double2bytes" | "double2str" | "fieldExists" | "float2bytes" | "float2str" | "int2bytes" | "int2str" | "IP2bytes" | "IP2str" | "list2str" | "long2bytes" | "long2str" | "object2str" | "randomInt" | "randomDouble" | "short2bytes" | "short2str" | "strpos" | "str2byte" | "str2bytes"

Table A–1 (Cont.) NPL Methods, Operator and Syntax

Method	Operator	Syntax
		| "str2double" | "str2float" | "str2int" | "str2IP" | "str2long" | "str2lower" | "str2short" | "str2TimeInMilliSecs" | "str2TimeInSecs" | "str2upper" | "strlen" | "subbytes" | "substr" | "time2bytes" | "time2str") "(" (Expression ("," Expression)*)? ")"
UnaryOperator	::=	"+" | "-"
FunctionStmt	::=	("logError" | "logWarning" | "logInfo" | "logTrace" | "write") "(" (Expression ("," Expression)*)? ")" ";"
JavaHookStmt	::=	JavaHookExpression ";"
JavaHookExpression	::=	"Java" "." <IDENTIFIER> "(" (Expression ("," Expression)*)? ")"
<IDENTIFIER>	::=	(<LETTER> | <UNDERSCORE>) (<LETTER> | <DIGIT> | <UNDERSCORE>)*
<ATTRIBUTE_NAME>	::=	(<LETTER> | <UNDERSCORE> | <COLON>) (<LETTER> | <DIGIT> | <UNDERSCORE> | <COLON> | <HYPHEN>)*
<FUNCTION_NAME>	::=	<LETTER> (<LETTER> | <DIGIT>)*
<LETTER>	::=	["a"-"z", "A"-"Z"]
<UNDERSCORE>	::=	"_"
<COLON>	::=	":"
<DIGIT>	::=	["0"-"9"]
<HYPHEN>	::=	<MINUS>
<INTEGER_LITERAL>	::=	<DECIMAL_LITERAL> | <HEX_LITERAL> | <OCTAL_LITERAL>

Table A–1 (Cont.) NPL Methods, Operator and Syntax

Method	Operator	Syntax
<DECIMAL_LITERAL>	::=	["1"-"9"] (["0"-"9"])*
<HEX_LITERAL>	::=	"0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+
<OCTAL_LITERAL>	::=	"0" (["0"-"7"])*
<FLOATING_POINT_LITERAL>	::=	(["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
<EXPONENT>	::=	["e","E"] (["+","-"])? (["0"-"9"])+

<STRING_LITERAL>	::=	"\""
		((~["\"","\\","\n","\r"])
		  | ("\\"
		  (["n","t","b","r","f","\\","'","\""]
		  | ["0"-"7"] (["0"-"7"])?
		  | ["0"-"3"] ["0"-"7"] ["0"-"7"]
		  )
		  )
		)*
		"\""

<OR>	::=	"||"
<AND>	::=	"&&"
<EQ>	::=	"=="
<NE>	::=	"!="
<LT>	::=	"<"
<LE>	::=	"<="
<GT>	::=	">"
<GE>	::=	">="
<MINUS>	::=	"-"

NPL reserved words

- Byte
- byte2bytes
- byte2str
- Bytes
- bytes2byte
- bytes2double
- bytes2float
- bytes2hexstr
- bytes2int
- bytes2IP
- bytes2long

- bytes2short
- bytes2str
- bytes2TimeInMilliSecs
- bytes2TimeInSecs
- Config
- convertDateTime
- copyBits
- currentTime
- Double
- double2bytes
- double2str
- else
- Expose
- fieldExists
- Float
- float2bytes
- float2str
- for
- if
- import
- InputRec
- int2bytesint2str
- Integer
- IP
- IP2bytes
- IP2str
- IPv6
- Java
- List
- list2str
- logError
- logInfo
- logTrace
- logWarning
- Long
- long2bytes
- long2str

- MAC
- null
- Object
- object2str
- OutputRec
- randomInt
- randomDouble
- Short
- short2bytes
- short2str
- str2byte
- str2bytes
- str2double
- str2float
- str2int
- str2IP
- str2long
- str2lower
- str2short
- str2TimeInMilliSecs
- str2TimeInSecs
- str2upper
- String
- strlen
- strpos
- subbytes
- substr
- time2bytes
- time2str
- TimeInMilliSecs
- TimeInSecs
- UInt128
- write