NIJA Platform™

The Definitive Guide 1.0

"open solutions for a smart world"

http://www.nijaplatform.com

An Overview

Nija is a powerful and versatile software development platform for supporting embeded electronic designs with a standard programming language.

Today, in parallel with the growth and being widespread of internet, different system have gained communication capabilities with each other. Embedded devices such as P.O.S terminals for credit card transactions, vending machines, access controllers, e-ticket validators are being designed over again regarding the support of those communication and payment systems.

During design R&D studies , developing the software exhausts most of time and resources. For these similar devices Is it possible to develop softwares independent and relaxed from hardwares? Nija arises from the answer “yes” replying the question.

Nija is a simple and productive programming language that can reduce the time and effort required to develop embedded applications.

NIJA IS DIFFERENT FROM OTHER PROGRAMMING PLATFORMS

Today’s program development tools are generally designed according to high-speed micro-processors and their high capacity programming and memory perperials . Nevertheless low-cost, embedded and multi-purpose micro-processor modules can be programmed using C or assembler languages of the processors. Though not very aware of them, in daily life we use these modules in refrigerators, washing-machines, TVs , computers, game-watches and etc. Similar modules can be developed in the way of communicating with each other as far. In a recent time we will be controlling TVs using our cellular phones or downloading the logs of the fridge’s or washing-machine’s to our PC. For all those features, it is required standard communication protocols. Smart -houses, smart-cities will gather in one concept using fast, effective and low-cost standard software platforms.

WHAT IS NIJA LIKE?

Nija is a script language very similar to JAVA. JAVA is a software tool, devoloped by SUN since 1995. The main reason for the popularity and wide usage of JAVA is its internet based architecture. On the other hand improvement of JAVA strengthens the idea of using a standard software development tool in different systems successfully.

Within this concept, NIJA is a platform aims developing a standard software for all embedded systems using micro controllers.
It is based on ECMAScript [ECMA262] standards. Moreover extra facilitating functions have been developed by NITELA NIJA Team. Nitela added many functions to standard library for smart cards, electronic payment systems and encryption algorithms in NITELA's expertise areas.
Nija Plaform is an easy to learn software tool helping software developers and supports international standards.

WHICH MICRO PROCESSORS FOR NIJA?

Comparing with other development platforms, NIJA can work even with a 8 bit microprocessor with low code area.

NIJA codes can easily run on 64 Kbyte code area, 8 Kbyte RAM memory and 8 bit micro-controller. However it is available for as well as 16 and 32 bit micro processors.

Same virtual machine program is used on windows as a part of the simulator.

As a matter of fact all program developing studies can be maid on a PC operating with Windows. The applications can be realized using NIJA simulator and debuger features shortly.

NIJA STARTER PLATFORM FOR WINDOWS

Nija Starter Platform is convenient particularly for applications developing for smart cards.

During writing the application, commands and results can be watched and process can be observed with all details step by step while working in debug mode. Software updates can be done quickly and saves time. Nija Starter Platform works with an EMV Compatible smart card reader.

NIJA PROFESSIONAL PLATFORM FOR WINDOWS

Professional platform works in parallel with the unit of real application. The application developed under Windows environment is ported to module as a real application and works in real time. The control units on the module can be connected to the main device. So, as examples, an e-ticket application can be tested with turnstile or if it is a vending application, vending machine's all operations can be watched and tested by simulator. All test results can be saved optionally .

NIJA SERVER:

All devices using NIJA units become capable of connecting to each other so they can communicate. All logs or data can be send to a centre on internet using Nija server. Additionally Nija Server can connect via GPRS infrastructure , giving a mobile and stand alone facility for mobile applications.

NIJA Script Development Tool

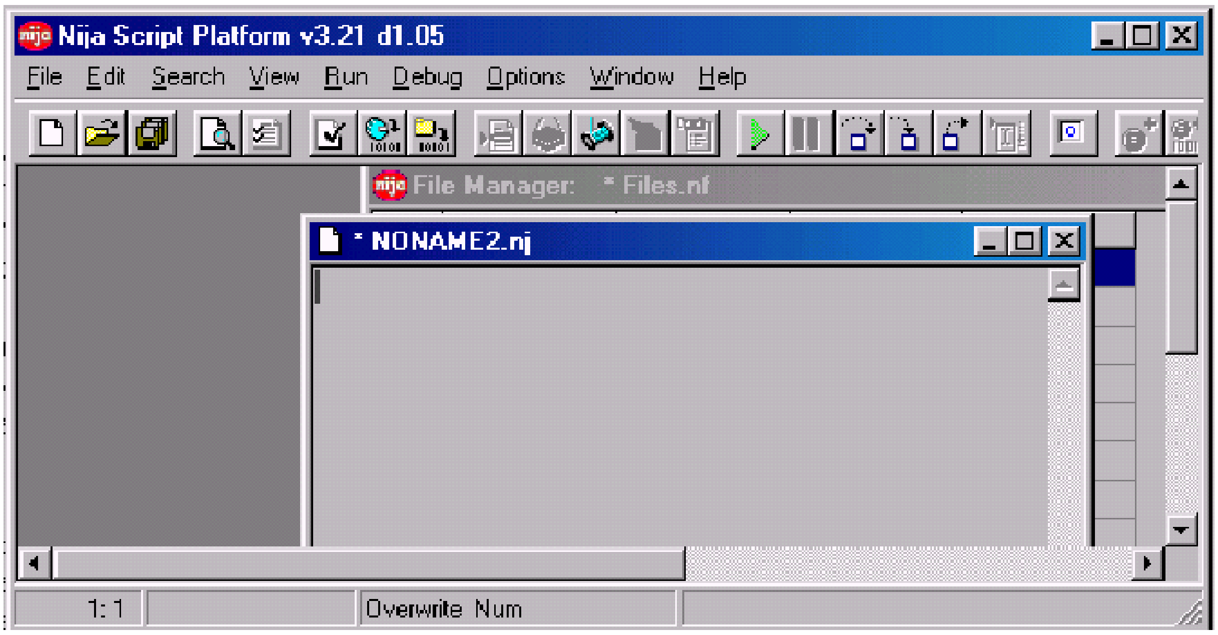
NIJA Platform is written for programmers interested in smart card programming. ,

It is designed for smart card application as high level programming.

Using NIJA a programmer can develop smart card applications for any variety of industries such as door access, vending machine, copy machine or/and any other smart card operated payment system.

PROTEKILA is a leader in the smart card sector with an important experience in a wide range including hardware and card operating system development, manufacturing and system deployment.

Hopefully, NIJA Platform will be a general smart card developing concept meeting all

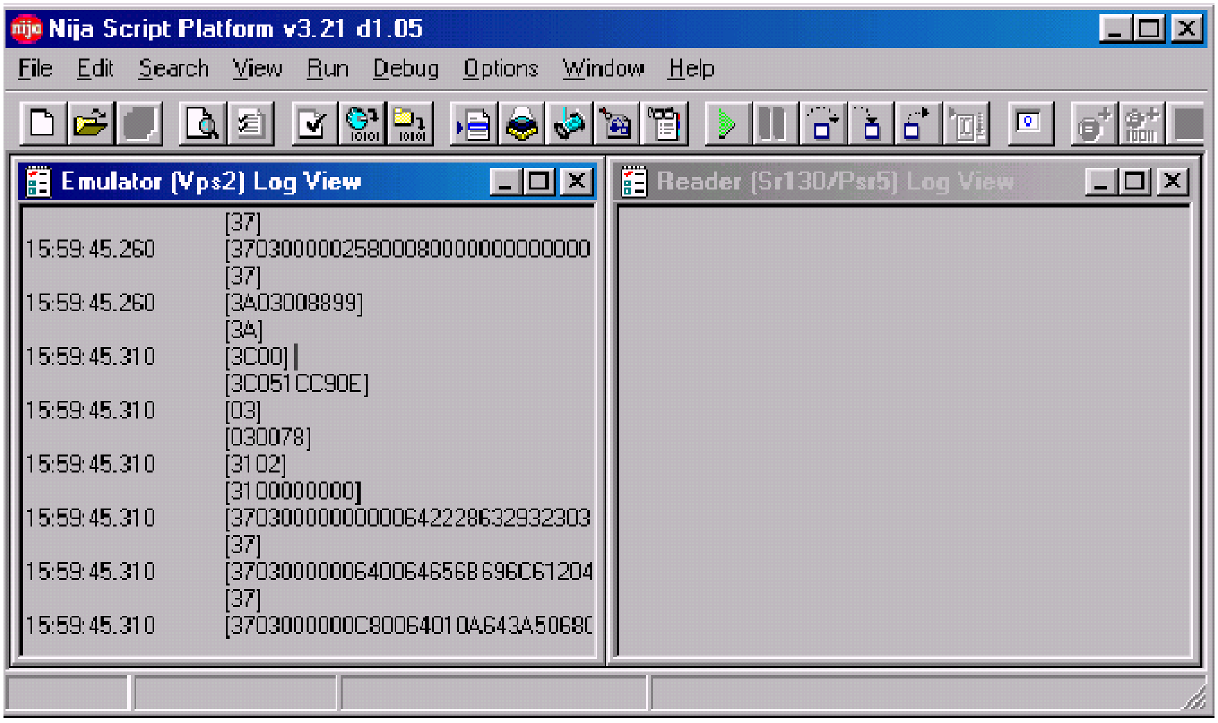

Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
File Manager: * Files.nf
* NONAME2.nj
1: 1
Overwrite Num

smart card based applications.

Figure 1.

Programmers can have access to ISO 7816 standards easily on which NIJA smart card technology is based. PROTEKILA has been developing smart card readers for POS terminals, parking meters, pay phones, and vending devices for customers around the world. PROTEKILA's smart card readers capitalizes on this know-how to provide its customers with the most reliable products.

PROTEKILA's smart card reader and NIJA developing tools works one within in another with high-level functions in card application for all smart cards which support ISO 7816 standard.
When you download your compiled source code to reader you can trace using NIJA script tools while your source code is running on real smart card reader.
The figure 2 shows that communication between reader and NIJA development tools. while you are tracing your source code.

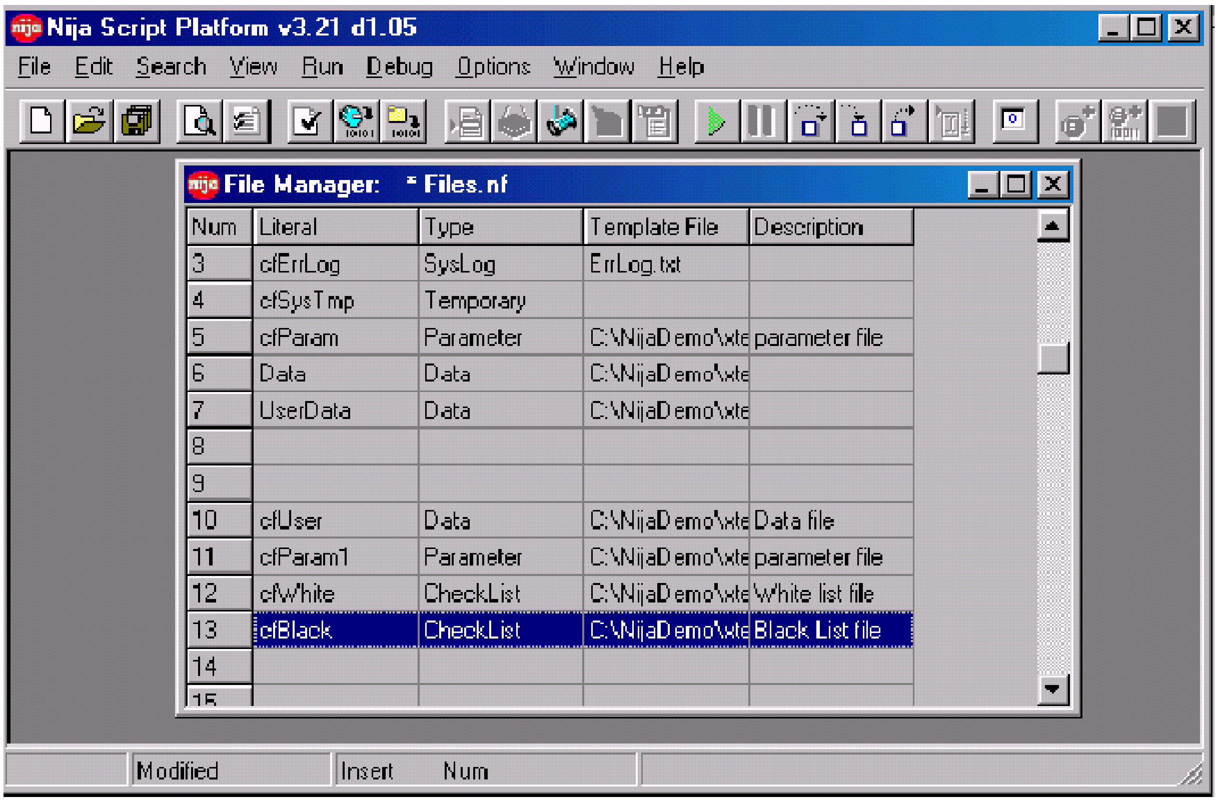

Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
Emulator (Vps2) Log View
Reader (Sr130/Psr5) Log View

```
                    [37]
15:59:45.260        [3703000002580008000000000000
                    [37]
15:59:45.260        [3A03008899]
                    [3A]
15:59:45.310        [3C00]
                    [3C051CC90E]
15:59:45.310        [03]
                    [030078]
15:59:45.310        [3102]
                    [3100000000]
15:59:45.310        [3703000000000064222863293230З
                    [37]
15:59:45.310        [370300000064006465686 96C61204
                    [37]
15:59:45.310        [3703000000C80064010A643A5068C
```

Figure 2.

However NIJA has a file system using high-level file function. File system can be reached very easily and safely. When source files is compiled and downloaded to reader, NIJA development Platform settles on thirty seven files in the kit which is used as a reader. These files is used various purpose such as a log file, black and white list maybe used for products price and similar to above purpose. Also NIJA tool has a menu to manage file system.
You can configure your file system from visual Platform with just a few mouse click move.

Figure 3 shows the NIJA Platform's file manager. The first four files are stable and all the other files can be used according to developer's needs.


Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
File Manager: * Files.nf
Modified
Insert
Num

Num	Literal	Type	Template File	Description
3	cfErrLog	SysLog	ErrLog.txt	
4	cfSysTmp	Temporary		
5	cfParam	Parameter	C:\NijaDemo\xte	parameter file
6	Data	Data	C:\NijaDemo\xte	
7	UserData	Data	C:\NijaDemo\xte	
8				
9				
10	cfUser	Data	C:\NijaDemo\xte	Data file
11	cfParam1	Parameter	C:\NijaDemo\xte	parameter file
12	cfWhite	CheckList	C:\NijaDemo\xte	White list file
13	cfBlack	CheckList	C:\NijaDemo\xte	Black List file
14				
15				

Figure 3.

In addition, you can find a simulator in NIJA Platform. This simulator responses like a real device while your code runs . Consequently you can download your tested code to real device using serial port or a special device simulating smart card protocols.

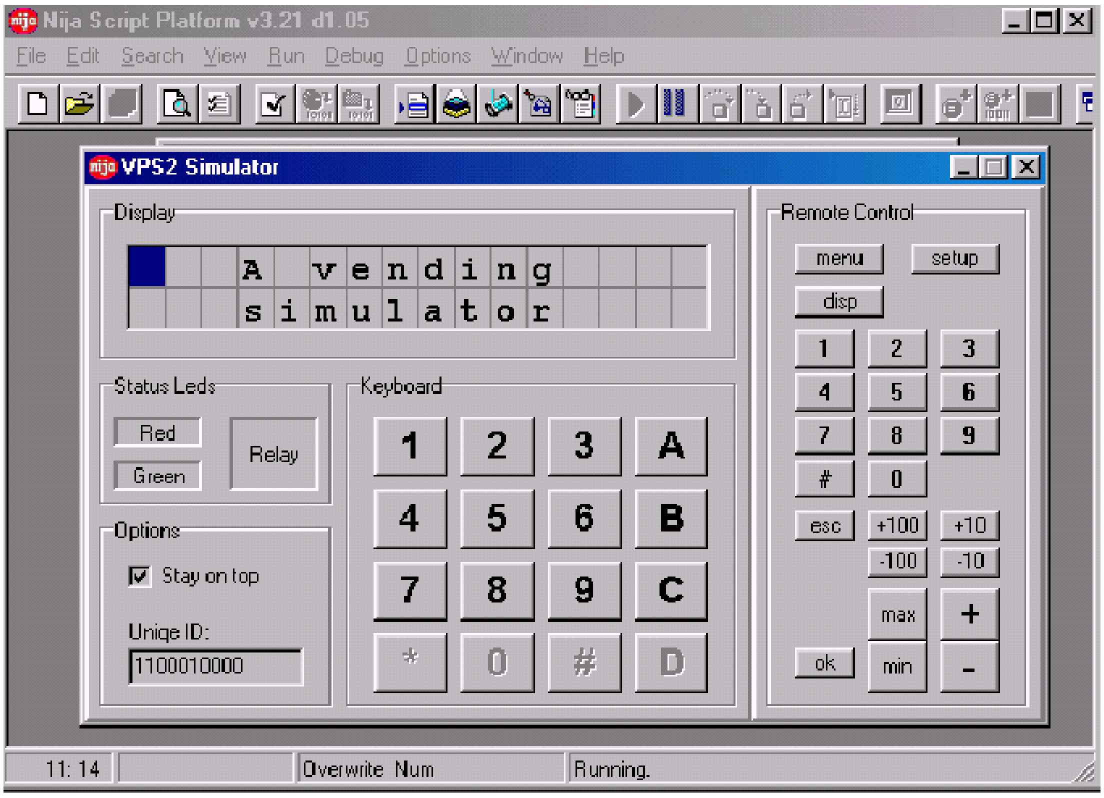
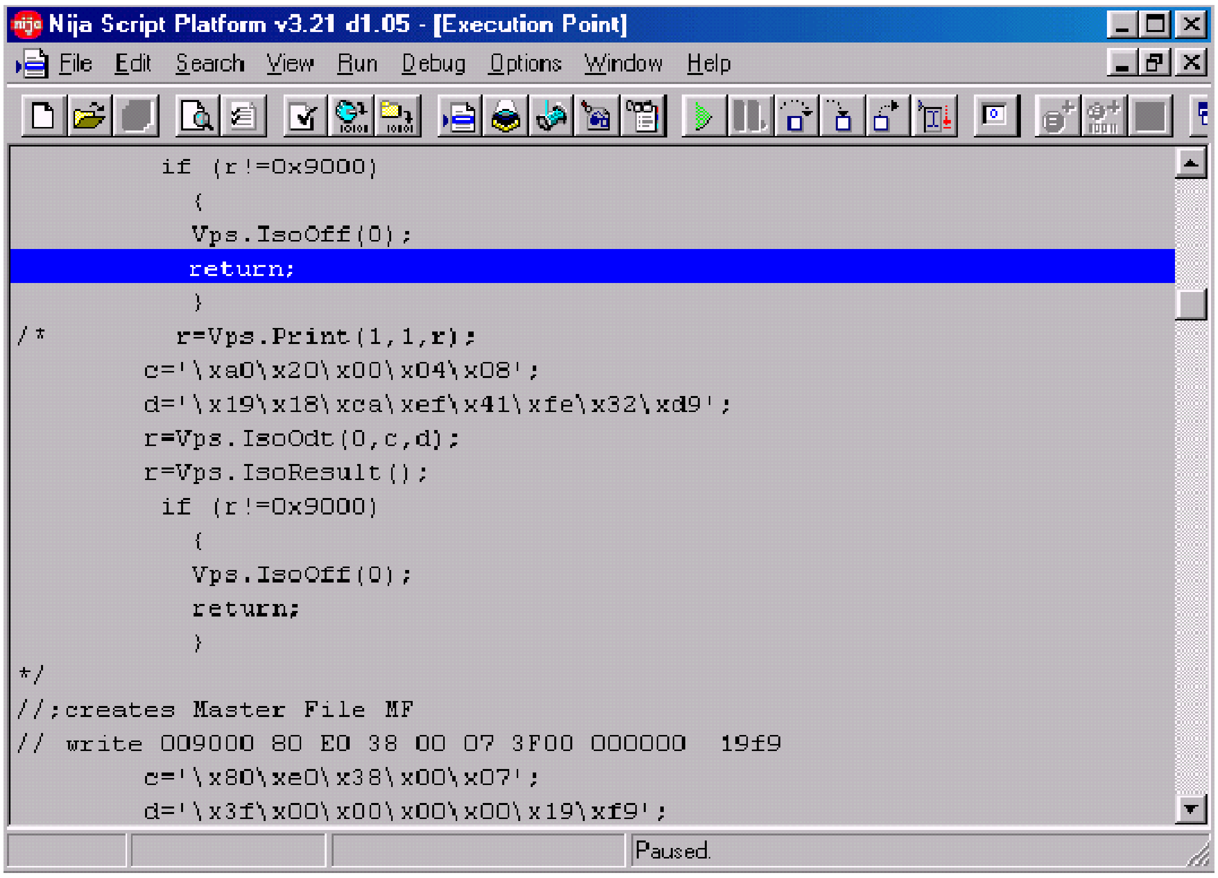


Figure 4.

However , you can watch your source code for what you are sending to serial port and what you are getting from serial port using event log menu file or you can trace your code or you can put break point to anywhere .

When you download your source code to simulator or to a real device you can trace your source code at the same time with pressing key F7 as referred below .

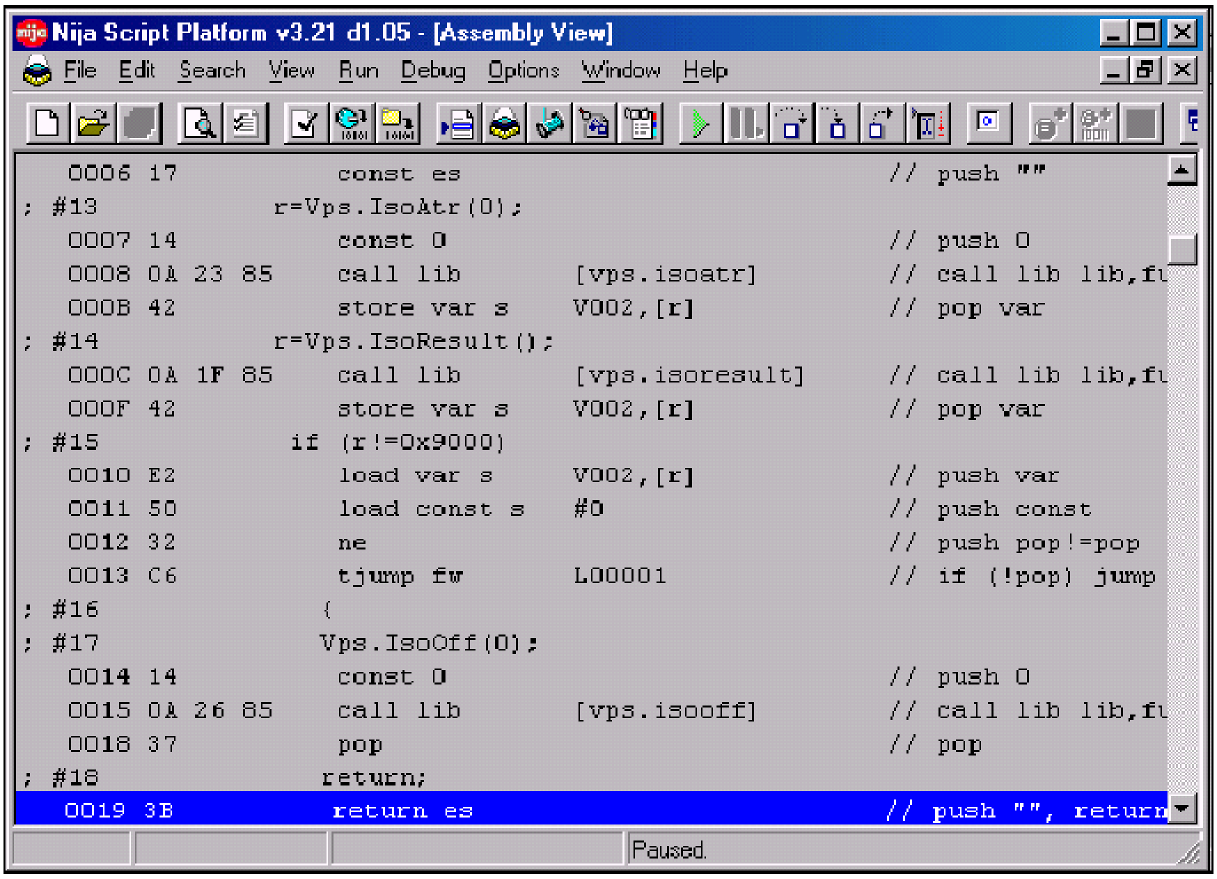

Nija Script Platform v3.21 d1.05 - [Execution Point]
File Edit Search View Run Debug Options Window Help
Paused

```
          if (r!=0x9000)
            {
            Vps.IsoOff(0);
            return;
            }
/*          r=Vps.Print(1,1,r);
         c='\xa0\x20\x00\x04\x08';
         d='\x19\x18\xca\xef\x41\xfe\x32\xd9';
         r=Vps.IsoOdt(0,c,d);
         r=Vps.IsoResult();
          if (r!=0x9000)
            {
            Vps.IsoOff(0);
            return;
            }
*/
//;creates Master File MF
// write 009000 80 E0 38 00 07 3F00 000000  19f9
         c='\x80\xe0\x38\x00\x07';
         d='\x3f\x00\x00\x00\x00\x19\xf9';
```

Figure 5.

or

also you can watch your source code from your assembly source.

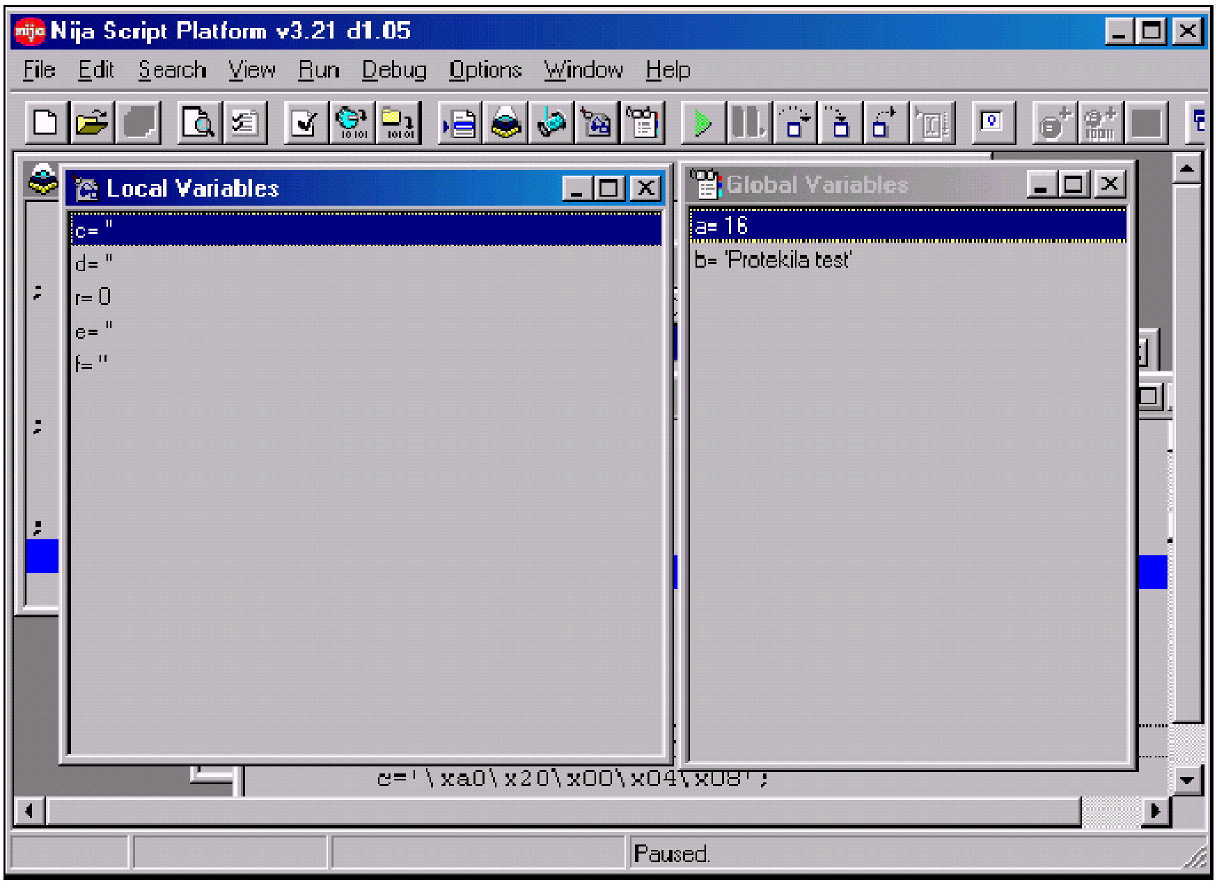

Nija Script Platform v3.21 d1.05 - [Assembly View]
File Edit Search View Run Debug Options Window Help

```
   0006 17          const es                        // push ""
; #13          r=Vps.IsoAtr(0);
   0007 14          const 0                         // push 0
   0008 0A 23 85    call lib        [vps.isoatr]    // call lib lib,fu
   000B 42          store var s     V002,[r]        // pop var
; #14          r=Vps.IsoResult();
   000C 0A 1F 85    call lib        [vps.isoresult] // call lib lib,fu
   000F 42          store var s     V002,[r]        // pop var
; #15           if (r!=0x9000)
   0010 E2          load var s      V002,[r]        // push var
   0011 50          load const s    #0              // push const
   0012 32          ne                              // push pop!=pop
   0013 C6          tjump fw        L00001          // if (!pop) jump
; #16              {
; #17              Vps.IsoOff(0);
   0014 14          const 0                         // push 0
   0015 0A 26 85    call lib        [vps.isooff]    // call lib lib,fu
   0018 37          pop                             // pop
; #18              return;
   0019 3B          return es                       // push "", return
```

Paused.

Figure 6.

NIJA Platform supports global variable declaration so you can reach these variable from anywhere and anytime. Global and local variables can be watched from NIJA Platform using NIJA tools.


Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
Local Variables
Global Variables
Paused

```
c= ''
d= ''
r= 0
e= ''
f= ''
```

```
a= 16
b= 'Protekila test'
```

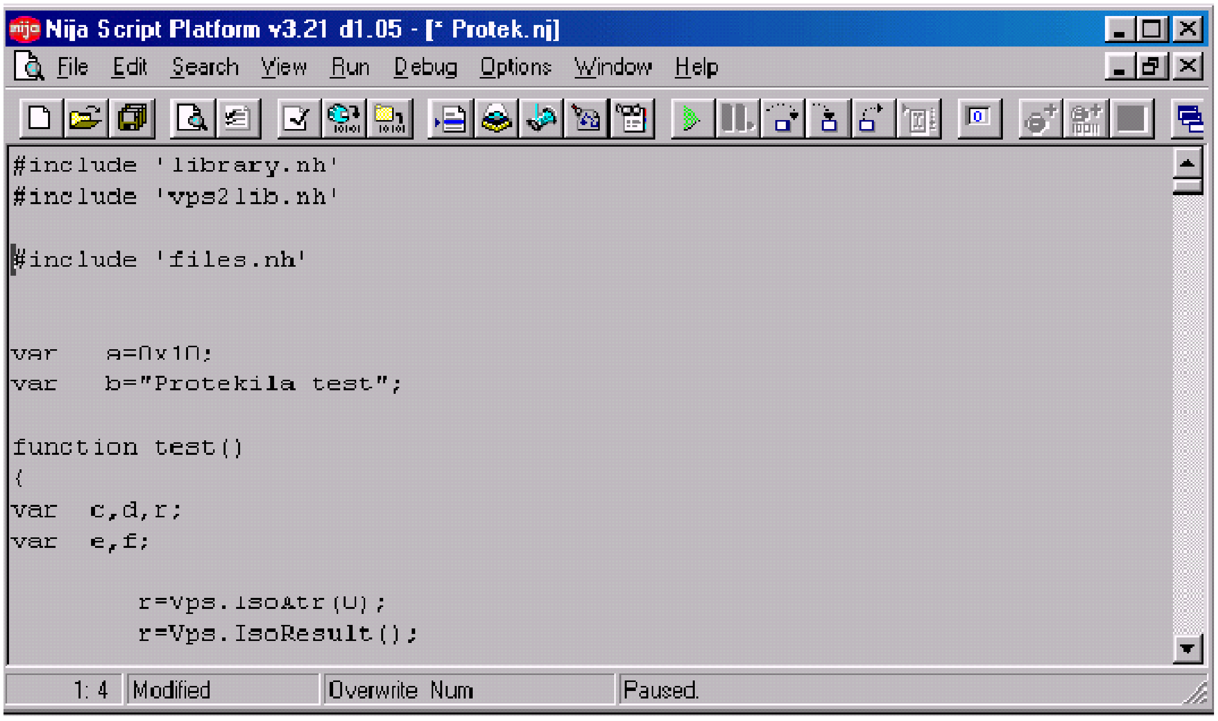
```
c='\xa0\x20\x00\x04\x08';
```

Figure 7.

After learning first principle of NIJA Platform. You can start your first application project.

Open a new script file from NIJA Platform and include files for library (These files must be included in your source file for to use NIJA functions)

Save your source file in your hard disk .

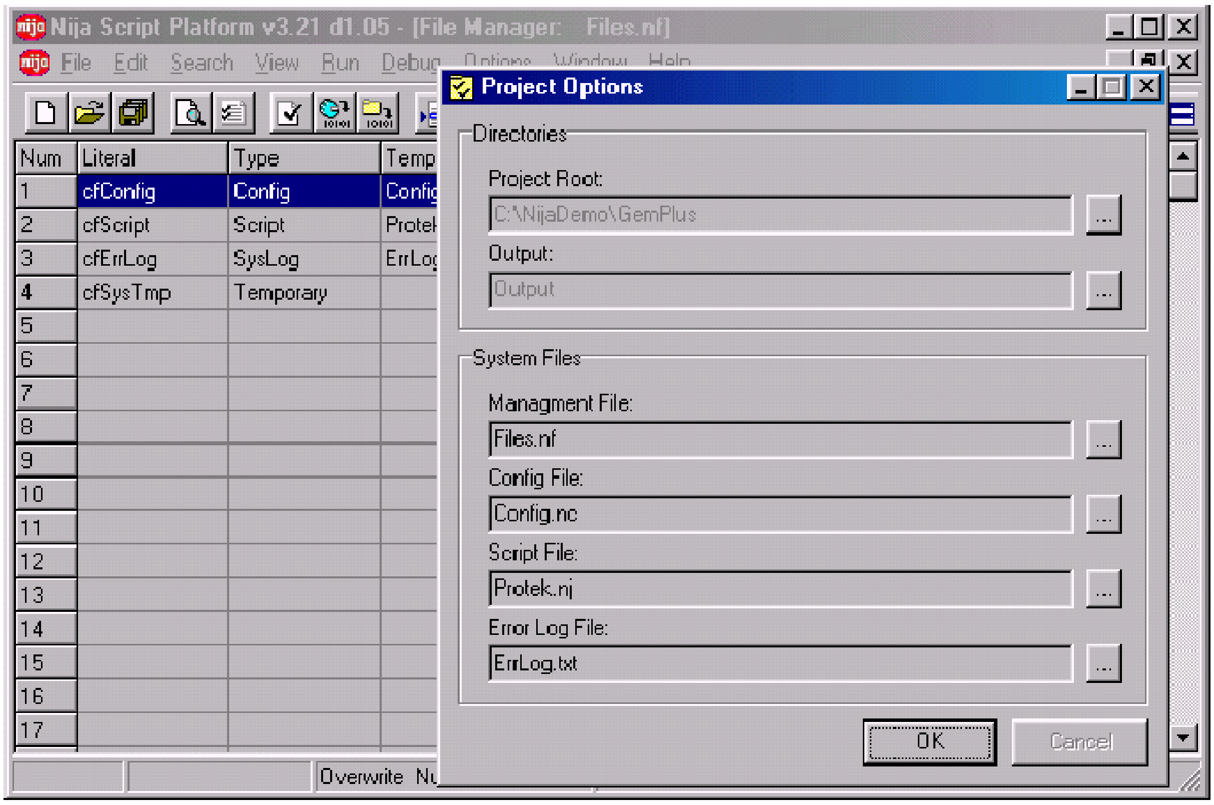

Nija Script Platform v3.21 d1.05 - [* Protek.nj]
File Edit Search View Run Debug Options Window Help

```
#include 'library.nh'
#include 'vps2lib.nh'

#include 'files.nh'


var    a=0x10;
var    b="Protekila test";

function test()
{
var   c,d,r;
var   e,f;

        r=vps.IsoAtr(0);
        r=Vps.IsoResult();
```

1: 4
Modified
Overwrite Num
Paused.

Figure 8.

Then go to Options menu , select Project . You will see project option directory menu and other necessary file for NIJA compiler.


Nija Script Platform v3.21 d1.05 - [File Manager: Files.nf]
File Edit Search View Run Debug Options Window Help
Project Options
OK
Cancel

Num	Literal	Type	Temp
1	cfConfig	Config	Config
2	cfScript	Script	Protek
3	cfErrLog	SysLog	ErrLog
4	cfSysTmp	Temporary	

Directories

Project Root: C:\NijaDemo\GemPlus

Output: Output

System Files

Managment File: Files.nf

Config File: Config.nc

Script File: Protek.nj

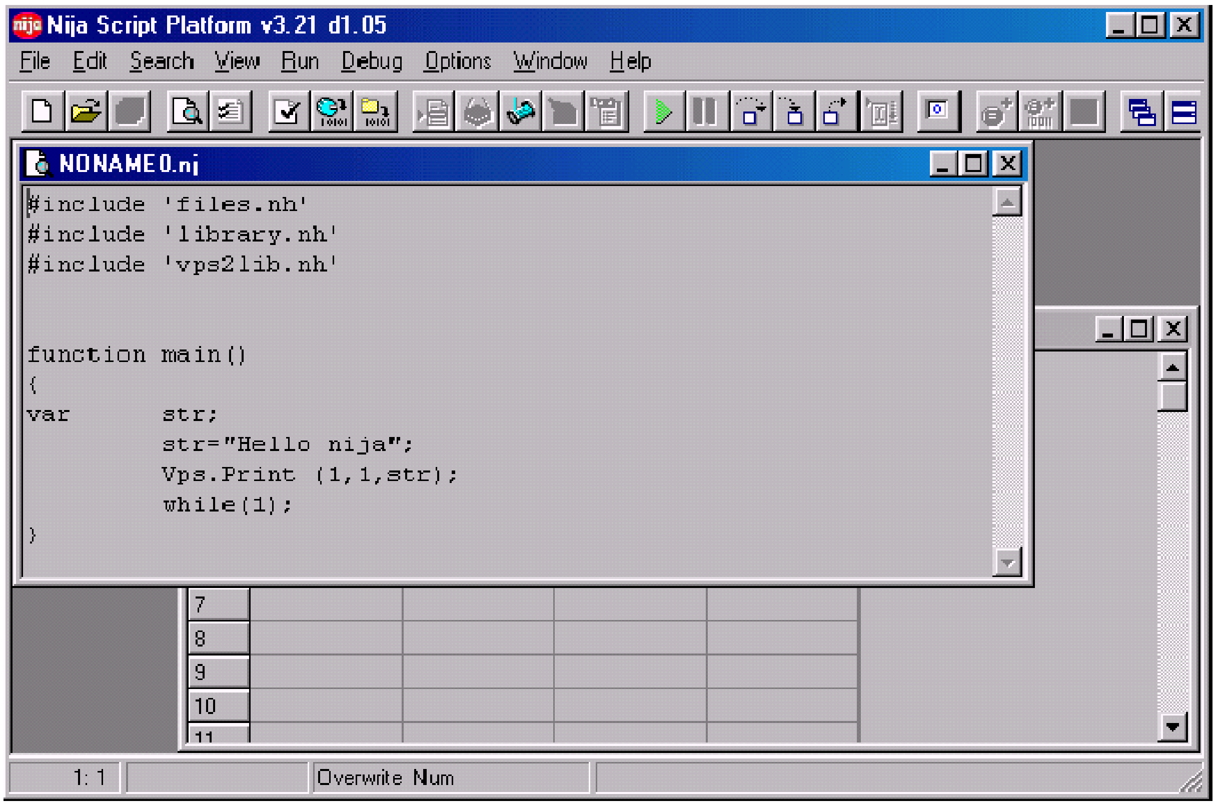
Error Log File: ErrLog.txt

Figure 9.

Move library files to the your working directory.

Project root is main directory which you've already saved to your source file. Output directory is created automatically.

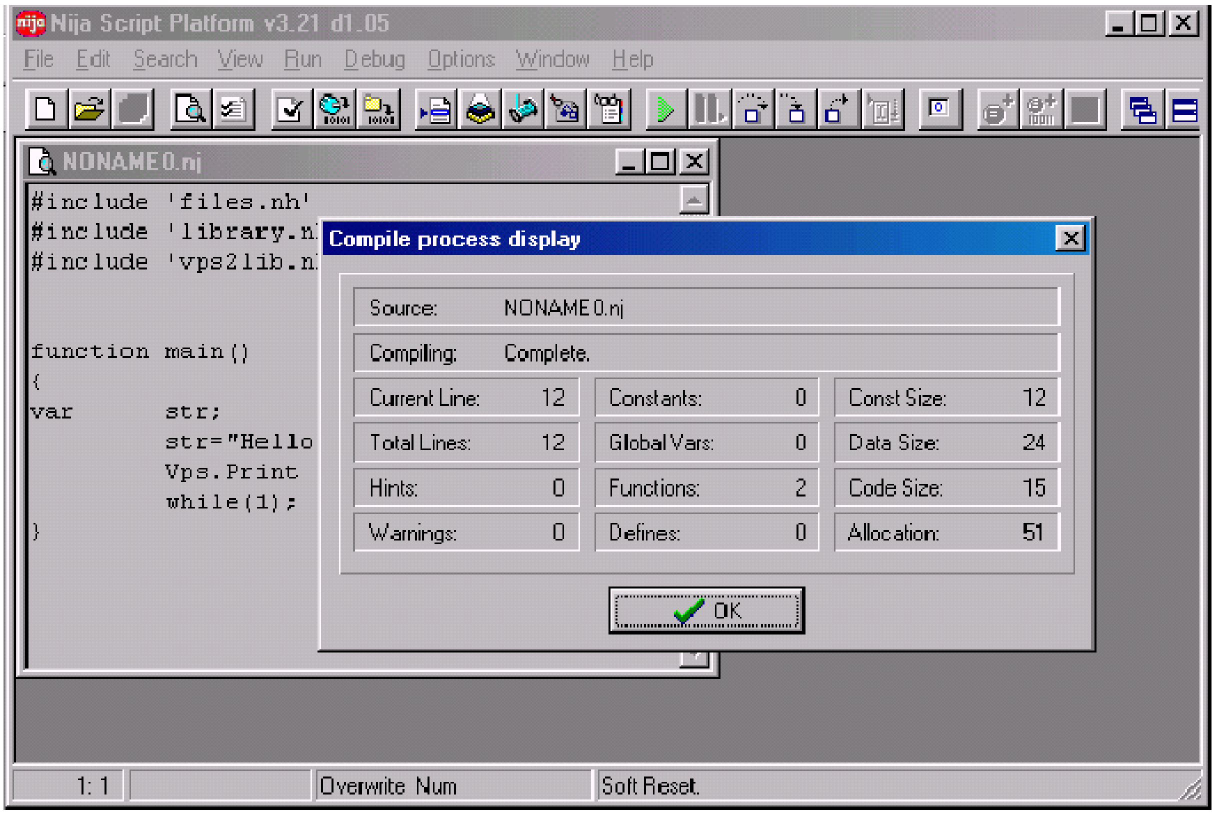
Write source code like below.


Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
NONAME0.nj
1: 1
Overwrite Num

```
#include 'files.nh'
#include 'library.nh'
#include 'vps2lib.nh'


function main()
{
var        str;
           str="Hello nija";
           Vps.Print (1,1,str);
           while(1);
}
```

Figure 10.

Now it is ready to compile your first application. When you press CTRL+F9 you will see below window.

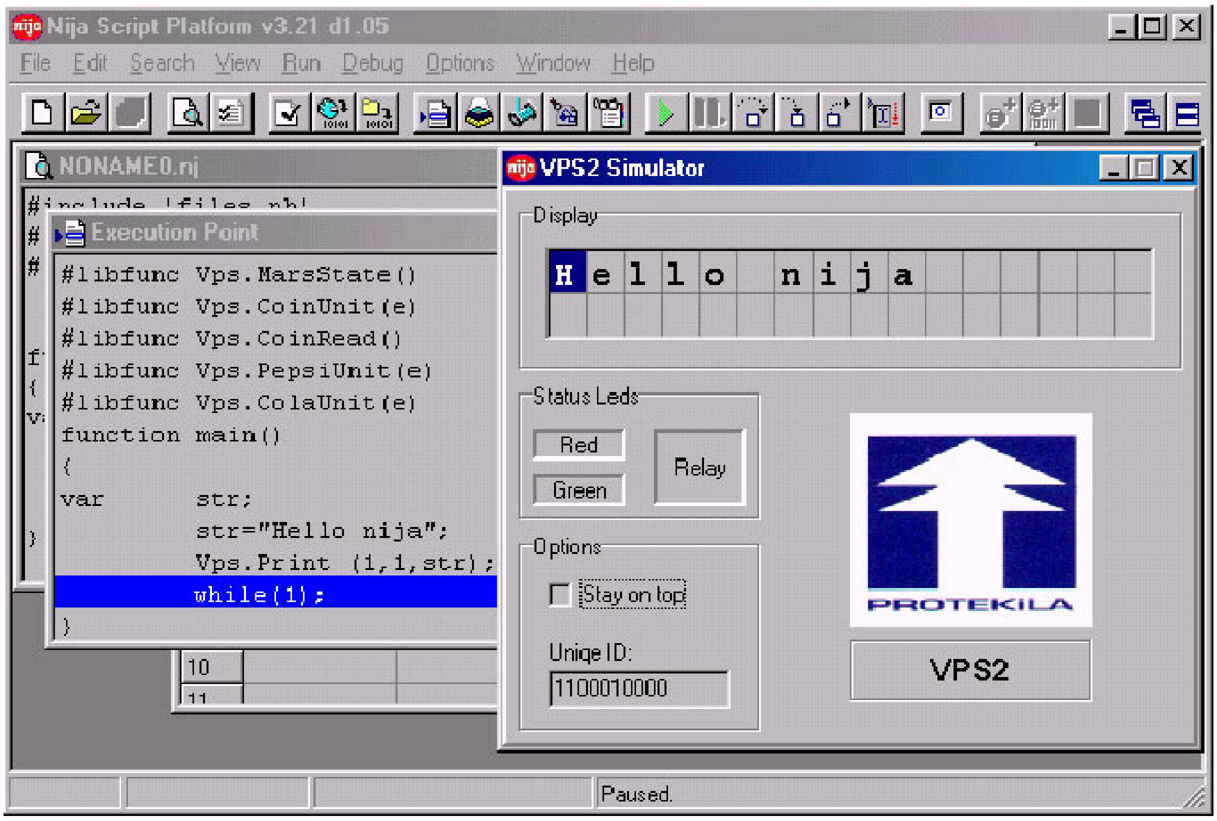

Nija Script Platform v3.21 d1.05
File Edit Search View Run Debug Options Window Help
NONAME0.nj
Compile process display
OK
1: 1
Overwrite Num
Soft Reset.

```
#include 'files.nh'
#include 'library.n
#include 'vps2lib.n


function main()
{
var        str;
           str="Hello
           Vps.Print
           while(1);
}
```

```
Source:         NONAME0.nj
Compiling:      Complete.
Current Line:   12    Constants:    0    Const Size:   12
Total Lines:    12    Global Vars:  0    Data Size:    24
Hints:           0    Functions:    2    Code Size:    15
Warnings:        0    Defines:      0    Allocation:   51
```

Figure 11.

and then press F9 to execute your source file you will see “hello NIJA” on screen (To see this screen select “simulator” from View menu).



```
#libfunc Vps.MarsState()
#libfunc Vps.CoinUnit(e)
#libfunc Vps.CoinRead()
#libfunc Vps.PepsiUnit(e)
#libfunc Vps.ColaUnit(e)
function main()
{
var       str;
          str="Hello nija";
          Vps.Print (1,1,str);
          while(1);
}
```

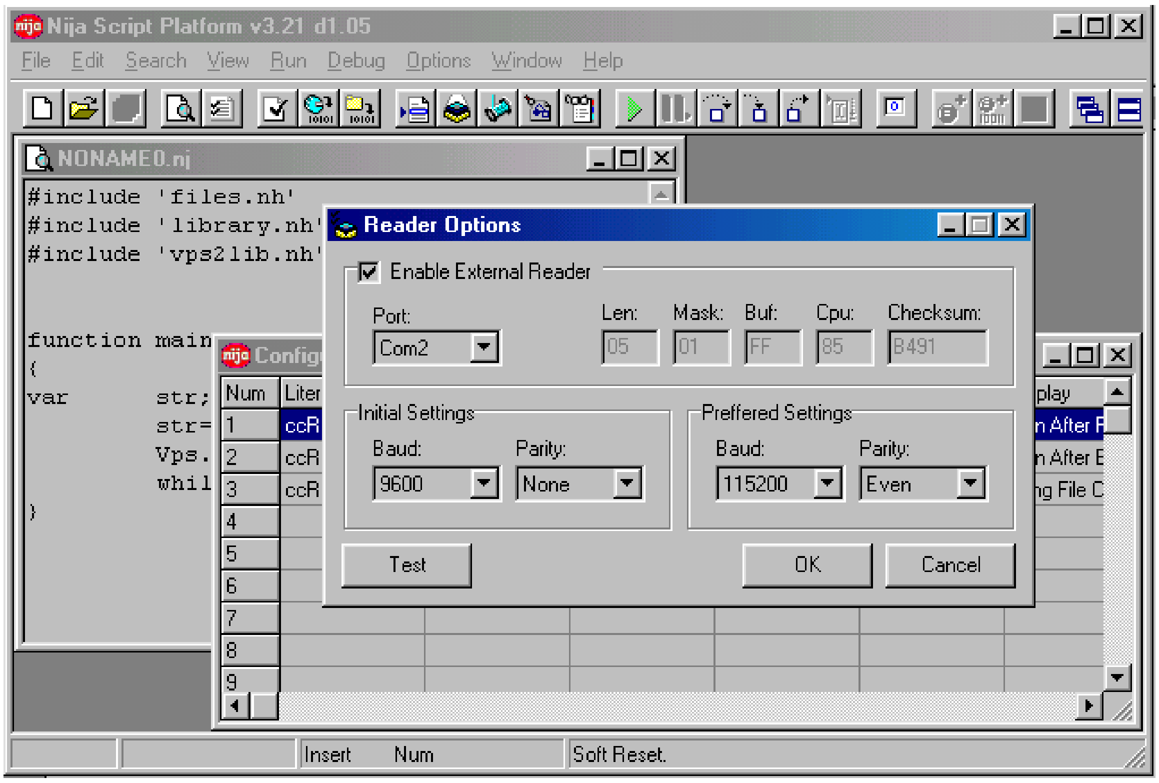
Display: Hello nija

Uniqe ID: 1100010000

Figure 12.

To configure config.nc file which is necessary for NIJA, you must select Option menu and select Configuration menu and then configure your config file.

If you want that your source code runs in external device you should select external device from Option =>Emulator or Reader for enable to external device and then test serial port.



```
#include 'files.nh'
#include 'library.nh'
#include 'vps2lib.nh'


function main
{
var        str;
           str=
           Vps.
           whil
}
```

Figure 13.

Now Press to F9 to run your source code in real device.

NIJA Script Language Specification

1. OVERVIEW

1.1 Why Scripting?

NIJA Script is designed to provide general scripting capabilities to Nitela Products. NIJA is designed to be used to specify application content for payment devices like vending machine, photo-copy printers and other smart card operated payment and control devices and systems. This content can be represented with text, images, selection lists etc. Simple formatting can be used to make the user interfaces more readable as long as the client device used to display the content can support it. However, all this content is static and there is no way to extend the language without modifying NIJA itself.

1.2 Benefits of using NIJA Script

Many of the services that can be used with thin mobile clients can be implemented with NIJA They can be used to supports more advanced UI functions, add intelligence to the client, provide access to the device and its peripheral functionality.
NIJA Script is based on ECMAScript [ECMA262] and does not require the developers to learn new concepts to be able to generate advanced services.

2. NIJA SCRIPT CORE

One objective for the NIJA Script language is to be close to the core of the ECMAScript Language specification [ECMA262]. The part in the ECMAScript Language specification that defines basic types, variables, expressions and statements is called core and can almost be used "as is" for the NIJA Script specification. This section gives an overview of the core parts of NIJA Script.

See chapter NIJA Script Grammar (5) for syntax conventions and precise language grammar.

2.1 Lexical Structure

This section describes the set of elementary rules that specify how you write programs in NIJA Script.

2.1.1 Case Sensitivity

NIJA Script is a case-sensitive language. All language keywords, variables and function names must use the proper capitalisation of letters.

2.1.2 Whitespace and Line Breaks

NIJA Script ignores spaces, tabs, newlines etc. that appear between tokens in programs, except those that are part of string constants.

Syntax:

```
WhiteSpace ::
<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>
LineTerminator ::
<LF>
<CR>
<CR><LF>
```

2.1.3 Usage of Semicolons

The following statements in NIJA Script have to be followed by a semicolon:1

- Empty statement (see 3.5.1)
- Expression statement (see 3.5.2)
- Variable statement (see 3.5.4)
- Break statement (see 3.5.8)
- Continue statement (see 3.5.9)
- Return statement (see 3.5.10)

2.1.4 Comments

The language defines two comment constructs: line comments (ie, start with // and end in the end of the line) and block comments (ie, consisting of multiple lines starting with /* and ending with */). It is illegal to have nested block comments.

Syntax:

Comment ::

MultiLineComment

SingleLineComment

MultiLineComment ::

/* MultiLineCommentChars$_{opt}$ */

SingleLineComment ::

// SingleLineCommentChars$_{opt}$

2.1.5 Literals

2.1.5.1 Integer Literals

Integer literals can be represented in three different ways: decimal, octal and hexadecimal integers.

Syntax:

DecimalIntegerLiteral ::
0
NonZeroDigit DecimalDigits$_{opt}$

NonZeroDigit :: **one of**

1 2 3 4 5 6 7 8 9

DecimalDigits ::
DecimalDigit
DecimalDigits DecimalDigit

DecimalDigit :: **one of**

0 1 2 3 4 5 6 7 8 9

HexIntegerLiteral ::
0x HexDigit
0X HexDigit
HexIntegerLiteral HexDigit

HexDigit :: **one of**
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

OctalIntegerLiteral ::
0 OctalDigit
OctalIntegerLiteral OctalDigit

OctalDigit :: one of
0 1 2 3 4 5 6 7

The minimum and maximum sizes for integer literals and values are specified in the section 3.2.7.1.

An integer literal that is not within the specified value range must result in a compile time error.

2.1.5.2 Floating-Point Literals

Floating-point literals can contain a decimal point as well as an exponent.

Syntax:

DecimalFloatLiteral ::

DecimalIntegerLiteral . $DecimalDigits_{opt}$ $ExponentPart_{opt}$

. DecimalDigits ExponentPartopt

DecimalIntegerLiteral ExponentPart

DecimalDigits ::

DecimalDigit

DecimalDigits DecimalDigit

ExponentPart ::

ExponentIndicator SignedInteger

ExponentIndicator :: **one of**

e E

SignedInteger ::

DecimalDigits

+ DecimalDigits

- DecimalDigits

The minimum and maximum sizes for floating-point literals and values are specified in the section 3.2.7.2. A floating-point literal that is not within the specified value range must result in a compile time error. A floating-point literal underflow results in a floating-point literal zero (0.0).

2.1.5.3 String Literals

Strings are any sequence of zero or more characters enclosed within double (") or single quotes (').

Syntax:

StringLiteral ::

" $DoubleStringCharacters_{opt}$ "

' $SingleStringCharacters_{opt}$ '

Examples of valid strings are:

"Example"   'Specials: \x00 \' \b'   "Quote: \""

Since some characters are not representable within strings, NIJA Script supports special escape
sequences by which these characters can be represented:

Sequence	Character represented	Unicode	Symbol
\’	Apostrophe or single quote	\u0027	'
\"	Double quote	\u0022	"
\\	Backslash	\u005C	\
\/	Slash	\u002F	/
\b	Backspace	\u0008	
\f	Form feed	\u000C	
\n	Newline	\u000A	
\r	Carriage return	\u000D	
\t	Horizontal tab	\u0009	
\xhh	The character with the encoding specified by two hexadecimal digits hh (Latin-1 ISO8859-1)		
\ooo	The character with the encoding specified by the three octal digits ooo (Latin-1 ISO8859-1)		
\uhhhh	The Unicode character with the encoding specified by the four hexadecimal digits hhhh.		

An escape sequence occurring within a string literal always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

2.1.5.4 Boolean Literals

A "truth value" in NIJA Script is represented by a boolean literal. The two boolean literals are: true and false.

Syntax:

```
BooleanLiteral ::
        true
        false
```

2.1.5.5 Invalid Literal

NIJA Script supports a special invalid literal to denote an invalid value.

Syntax:

```
InvalidLiteral ::
        invalid
```

2.1.6 Identifiers

Identifiers are used to name and refer to three different elements of NIJA Script: variables (see 3.2) and functions (see 3.4) . Identifiers cannot start with a digit but can start with an underscore (_).

Syntax:

Identifier ::

    IdentifierName **but not** ReservedWord

IdentifierName ::

IdentifierLetter
IdentifierName IdentifierLetter
IdentifierName DecimalDigit

IdentifierLetter :: **one of**
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
_

DecimalDigit :: **one of**
0 1 2 3 4 5 6 7 8 9

Examples of legal identifiers are:

timeOfDay speed quality HOME_ADDRESS var0 _myName ____

The compiler looks for the longest string of characters make up a valid identifier. Identifiers cannot contain any special characters except underscore (_). NIJA Script keywords and reserved words cannot be used as identifiers. Examples of illegal identifiers are:

while for if my~name $sys 123 3pieces take.this

Uppercase and lowercase letters are distinct which means that the identifiers speed and Speed are different.

2.1.7 Reserved Words

NIJA Script specifies a set of reserved words that have a special meaning in programs and they cannot be used as identifiers. Examples of such words are (full list can be found from the NIJA Script grammar specification, see chapter 5):

break continue false true while

2.1.8 Name Spaces

NIJA Script supports name spaces for identifiers that are used for different purposes. The following name spaces are supported:

· Function names (see 6.4)
· Function parameters (see 6.4) and variables (see 6.2)
· Pragmas (see 6.7)

Thus, the same identifiers can be used to specify a function name, variable/parameter name or a name for a pragma within the same compilation unit

2.2 Variables and Data Types

This section describes the two important concepts of NIJA Script language: variables and internal data types. A variable is a name associated with a data value. Variables can be used to store and manipulate program data. NIJA Script supports local variables declared inside functions or passed as function parameters (see 3.4). NIJA Script also supports global variables. Global variables must be declared before main function.

2.2.1 Variable Declaration

Variable declaration is compulsory in NIJA Script. Variable declaration is done simply by using the var keyword and a variable name (see section 6.5.4 for information about variable statements).
Variable names follow the syntax defined for all identifiers (see section 3.1.6):

```
var x;
var price;
var x,y;
var size = 3;
```

Variables must be declared before they can be used. Initialisation of variables is optional. Uninitialised variables are automatically initialised to contain an empty string ("").

2.2.2 Variable Scope and Lifetime

The scope of NIJA Script variables is the remainder of the function (see 3.4) in which they have been declared (If they are not global variables). All variable names within a function must be unique. Block statements (see 3.5.3) are not used for scoping.

```
function priceCheck(givenPrice) {
if (givenPrice > 100) {
var newPrice = givenPrice;
} else {
newPrice = 100;
};
return newPrice;
};
```

The lifetime of a variable is the time between the variable declaration and the end of the function. (If it is not a global variable)

```
function foo() {
x = 1; // Error: usage before declaration
var x,y;
if (x) {
var y; // Error: redeclaration
};
};
```

2.2.3 Variable Access

Variables are accessible only within the function in which they have been declared. Accessing the content of a variable is done by using the variable name:

```
var myAge = 37;
var yourAge = 63;
var ourAge = myAge + yourAge;
```

2.2.4 Variable Type

NIJA Script is a weakly typed language. The variables are not typed but internally the following basic data types are supported: boolean, integer, floating-point and string. In addition to these, a fifth data type invalid is specified to be used in cases an invalid data type is needed to separate it from the other internal data types. Since these data types are supported only internally, the programmer does not have to specify variable types and any variable can contain any type of data at any given time. NIJA script will attempt automatically convert between the different types as needed.

```
var flag = true;            // Boolean
var number = 12;            // Integer
var temperature = 37.7;     // Float
number = "XII";             // String
var except = invalid;       // Invalid
```

2.2.5 L-Values

Some operators (see 3.3.1 for more information about assignment operators) require that the left operand is a reference to a variable (L-value) and not the variable value. Thus, in addition to the five data types supported by NIJA Script, a sixth type variable is used to specify that a variable name must be provided.

```
result += 111;        // += operator requires a variable
```

2.2.6 Type Equivalency

NIJA Script supports operations on different data types. All operators (see section 3.3) specify the accepted data types for their operands. Automatic data type conversions (see chapter 4) are used to convert operand values to required data types.

2.2.7 Numeric Values

NIJA Script supports two different numeric variable values: integer and floating-point values. Variables can be initialised with integer and floating-point literals and several operators can be used to modify their values during the run-time. Conversion rules between integer and floating-point values are specified in chapter 4.

```
var pi = 3.14;
var length = 0;
var radius = 2.5;
length = 2*pi*radius;
```

2.2.7.1 Integer Size

The size of the integer is 32 bits (two's complement). This means that the supported value range for integer values is: -2147483648 and 2147483647. Lang [NIJALibs] library functions can be used to get these values during the run-time:

Lang.maxInt() Maximum representable integer value Lang.minInt() Minimum representable integer value.

2.2.7.2 Floating-point Size

The minimum/maximum values and precision for floating-point values are specified by [IEEE754].
NIJA Script supports 32-bit single precision floating-point format:

- Maximum value: 3.40282347E+38

- Minimum positive nonzero value (at least the normalised precision must be supported):
  1.17549435E-38 or smaller

The Float [NIJALibs] library can be used to get these values during the run-time:

Float.maxFloat()	Maximum representable floating-point value supported.
Float.minFloat()	Smallest positive nonzero floating-point value supported.

The special floating-point number types are handled by using the following rules:

- If an operation results in a floating-point number that is not part of the set of finite real numbers (not a number, positive infinity etc.) supported by the single precision floating-point format then the result is an invalid value.
- If an operation results in a floating-point underflow the result is zero (0.0).
- Negative and positive zero are equal and undistinguishable.

2.2.8 String Values

NIJA Script supports strings that can contain letters, digits, special characters etc. Variables can be initialised with string literals and string values can be manipulated both with NIJA Script operators and functions specified in the standard String library [NIJAsLibs].

```
var msg = "Hello";
var len = Str.length(msg);
msg = msg + ' Worlds!';
```

2.2.9 Boolean Values

Boolean values can be used to initialise or assign a value to a variable or in statements which require a boolean value as one of the parameters. Boolean value can be a literal or the result of a logical expression evaluation (see 3.3.3 for more information).

```
var truth = true;
var lie = !truth;
```

2.3 Operators and Expressions

The following sections describe the operators supported by NIJA Script and how they can be used to form complex expressions.

2.3.1 Assignment Operators

NIJA Script supports several ways to assign a value to a variable. The simplest one is the regular assignment (=) but assignments with operation are also supported:

Operator	Operation
=	Assign
+=	add (numbers)/concatenate (strings) and assign
-=	subtract and assign
*=	multiply and assign
/=	Divide and assign
Div=	Divide (integer division) and assign
%=	remainder (the sign of the result equals the sign of the dividend) and assign
<<=	Bitwise left shift and assign
>>=	Bitwise right shift with sign and assign
>>>=	Bitwise right shift zero fill and assign
&=	Bitwise AND and assign
^=	Bitwise XOR and assign
|=	Bitwise OR and assign

Assignment does not necessarily imply sharing of structure nor does assignment of one variable change the binding of any other variable.

```
var a = "abc";
var b = a;
b = "def"; // Value of a is "abc"
```

2.3.2 Arithmetic Operators

NIJA Script supports all the basic binary arithmetic operations:

Operator	Operation
+	Add (numbers)/concatenation (strings)
-	Subtract
*	Multiply
/	Divide
Div	integer division

In addition to these, a set of more complex binary operations are supported, too:

Operator	Operation
%	remainder, the sign of the result equals the sign of the dividend
<<	Bitwise left shift
>>	Bitwise right shift with sign
>>>	Bitwise shift right with zero fill
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

The basic unary operations supported are:

Operator	Operation
+	Plus
-	Minus
--	Pre-or-post decrement
++	pre-or-post increment
~	bitwise NOT

Examples:

```
var y = 1/3;
var x = y*3+(++b);
```

2.3.3 Logical Operators

NIJA Script supports the basic logical operations:

Operator	Operation
&&	logical AND
||	logical OR
!	logical NOT (unary)

Logical AND operator evaluates the first operand and tests the result. If the result is false, the result of the operation is false and the second operand is not evaluated. If the first operand evaluates to true, the result of the operation is the result of the evaluation of the second operand.

If the first operand evaluates to invalid, the second operand is not evaluated and the result of the operation is invalid.

Similarly, the logical OR evaluates the first operand and tests the result. If the result is true, the result of the operation is true and the second operand is not evaluated. If the first operand evaluates to false, the result of the operation is the result of the evaluation of the second operand. If the first operand evaluates to invalid, the second operand is not evaluated and the result of the operation is invalid.

```
weAgree = (iAmRight && youAreRight) ||
(!iAmRight && !youAreRight);
```

NIJA Script requires a value of boolean type for logical operations. Automatic conversions from other types to boolean type and vice versa are supported (see 4).

Notice: If the value of the first operand for logical AND or OR is invalid, the second operand is not evaluated and the result of the operand is invalid:

```
var a = (1/0) || foo(); // result: invalid, no call to foo()
var b = true || (1/0); // true
var c = false || (1/0); // invalid
```

2.3.4 String Operators

NIJA Script supports string concatenation as a built-in operation. The + and += operators used with strings perform a concatenation on the strings. Other string operations are supported by a standard String library (see [NIJALibs]).

```
var str = "Beginning" + "End";
var chr = Str.charAt(str,10);  //  chr = "E"
```

2.3.5 Comparison Operators

NIJA Script supports all the basic comparison operations:

Operator	Operation
<	less than
<=	Minus
==	Equal
>=	greater or equal
>	greater than
!=	İnequality

Comparison operators use the following rules:

- Boolean: true is larger than false
- Integer: Comparison is based on the given integer values
- Floating-point: Comparison is based on the given floating-point values
- String: Comparison is based on the order of character codes of the given string values.

  Character codes are defined by the character set supported by the NIJAScript Interpreter
- Invalid: If at least one of the operands is invalid then the result of the comparison is invalid

Examples:

```
var res = (myAmount > yourAmount);
var val = ((1/0) == invalid); // val = invalid
```

2.3.6 Array Operators

NIJA Script does not support arrays as such. However, the standard String library (see[NIJASLibs]) supports functions by which array like behaviour can be implemented by using strings.
A string can contain elements that are separated by a separator specified by the application programmer. For this purpose, the String library contains functions by which creation and management of string arrays can be done.

```
function dummy() {
var str = "Mary had a little lamb";
var word = Str.elementAt(str,4," ");
};
```

2.3.7 Comma Operator

NIJA Script supports the comma (,) operator by which multiple evaluations can be combined into one expression. The result of the comma operator is the value of the second operand:

```
for (a=1, b=100; a < 10; a++,b++) {
… do something …
};
```

Commas used in the function call to separate parameters and in the variable declarations to separate multiple variable declarations are not comma operators. In these cases, the comma operator must be placed inside the parenthesis:

```
var a=2;
var b=3, c=(a,3);
myFunction("Name", 3*(b*a,c));  //  Two parameters: "Name",9
```

2.3.8 Conditional Operator

NIJA Script supports the conditional (?:) operator which takes three operands. The operator selectively evaluates one of the given two operands based on the boolean value of the first operand. If the value of the first operand (condition) is true then the result of the operation is the result of the evaluation of the second operand. If the value of the first operand is false or invalid then the result of the operation is the result of the evaluation of the third operand.

```
myResult = flag ? "Off" : "On (value=" + level + ")";
```

Notice: This operator behaves like an if statement (see 3.5.5). The third operand is evaluated if the evaluation of the condition results in false or invalid.

2.3.9 typeof Operator

Although NIJAScript is a weakly typed language, internally the following basic data types are supported: boolean, integer, floating-point, string and invalid. Typeof (typeof) operator returns an integer value12 that describes the type of the given expression. The possible results are:

Type	**Code**
Integer:	0
Floatingpoint:	1
String:	2
Boolean:	3
Invalid:	4

Typeof operator does not try to convert the result from one type to another but returns the type as it is after the evaluation of the expression.

```
var str = "123";
var myType = typeof str; // myType = 2
```

2.3.10 isvalid Operator

This operator can be used to check the type of the given expression. It returns a boolean value false if the type of the expression is invalid, otherwise true is returned. isvalid operator does not try to convert the result from one type to another but returns the type as it is after the evaluation of the expression.

```
var str = "123";
var ok = isvalid str;  //  true
var tst = isvalid (1/0);  //  false
```

2.3.11 Expressions

NIJA Script supports most of the expressions supported by other programming languages. The simplest expressions are constants and variable names, which simply evaluate to either the value of the constant or the variable.

```
567
66.77
"This is too simple"
'This works too'
true
myAccount
```

Expressions that are more complex can be defined by using simple expressions together with operators and function calls.

```
myAccount + 3
(a + b)/3
initialValue + nextValue(myValues);
```

2.3.12 Expression Bindings

The following table contains all operators supported by NIJA Script. The table also contains information about operator precedence (the order of evaluation) and the operator associativity (leftto-right (L) or right-to-left (R)):

Prece Dence	Associativity	Operator	Operand types	Result type	Operation performed
1	R	++	number	number*	pre- or post-increment (unary)
1	R	--	number	number*	pre- or post-increment (unary)
1	R	+	number	number*	unary plus
1	R	-	number	number*	unary minus (negation)
1	R	~	integer	integer*	bitwise NOT (unary)

1	R	!	boolean	boolean*	logical NOT (unary)
1	R	typeof	any	integer	return internal data type (unary)
1	R	isvalid	any	boolean	check for validity (unary)
2	L	*	numbers	number*	multiplication
2	L	/	numbers	floatingpoint*	division
2	L	div	İntegers	integer*	integer division
2	L	%	integers	integer*	remainder
3	L	-	numbers	number*	subtraction
3	L	+	numbers or strings	number or string*	addition (numbers) or string concatenation
4	L	<<	integers	integer*	bitwise left shift
4	L	>>	integers	integer*	bitwise right shift with sign
4	L	>>>	integers	integer*	bitwise right shift with zero fill
5	L	<, <=	numbers or strings	Boolean*	less than,less than or equal
5	L	>, >=	numbers or strings	Boolean*	greater than, greater or equal
6	L	==	numbers or strings	Boolean*	equal (identical values)
6	L	!=	numbers or strings	Boolean*	not equal (different values)
7	R	&	integers	integer*	bitwise AND
8	R	^	integers	integer*	bitwise XOR
9	R	|	integers	integer*	bitwise OR
10	R	&&	booleans	boolean*	logical AND
11	R	||	booleans	boolean*	logical OR
12	R	? :	Boolean , any, any	any*	conditional expression
13	R	=	variable, any	any	Assignment
13	R	*=, -=	variable, number	number*	assignment with numeric operation
13	R	/=	variable, number	floatingpoint*	assignment with numeric operation
13	R	%=,div=	variable, integer	integer*	assignment with integer operation

13	R	+=	variable, number or string	number or string*	assignment with addition or concatenation
13	R	<<=, >>=, >>>=, &=, ^=, |=	variable, integer	integer*	assignment with bitwise operation
14	L	,	any	any	multiple evaluation

* The operator can return an invalid value in case the data type conversions fail (see chapter 7 for more information about conversion rules) or one of the operands is invalid.

2.4 Functions

A NIJA Script function is a named part of the NIJA Script compilation unit that can be called to perform a specific set of statements and to return a value. The following sections describe how NIJA Script functions can be declared and used.

2.4.1 Declaration

Function declaration can be used to declare a NIJA Script function name (Identifier) with the optional parameters (FormalParameterList) and a block statement that is executed when the function is called. All functions have the following characteristics:

- Function declarations cannot be nested.
- Function names must be unique within one compilation unit.
- All parameters to functions are passed by value.

- Function calls must pass exactly the same number of arguments to the called function as specified in the function declaration.
- Function parameters behave like local variables that have been initialised before the function body (block of statements) is executed.
- A function always returns a value. By default it is an empty string (""). However, a return statement can be used to specify other return values.

Functions in NIJA Script are not data types but a syntactical feature of the language.

Syntax:

```
FunctionDeclaration :
        extern_opt function Identifier ( FormalParameterListopt ) Block ;_opt
FormalParameterList :
        Identifier
        FormalParameterList , Identifier
```

Arguments: The optional extern keyword can be used to specify a function to be externally accessible. Such functions can be called from outside the compilation unit in which they are defined. There must be at least one externally accessible function in a compilation unit. Identifier is the name specified for the function. FormalParameterList (optional) is a comma-separated list of argument names. Block is the body of the function that is executed when the function is called and the parameters have been initialised by the passed arguments.

Examples:

```
function currencyConverter(currency, exchangeRate) {
  return currency*exchangeRate;
};

extern function testIt() {
  var UDS = 10;
  var FIM = currencyConverter(USD, 5.3);
};
```

2.4.2 Function Calls

The way a function is called depends on where the called (target) function is declared. The following sections describe the three function calls supported by NIJA Script: local script function call, external function (in the next version) call and library function call.

2.4.2.1 Local Script Functions

Local script functions (defined inside the same compilation unit) can be called simply by providing the function name and a comma separated list of arguments (number of arguments must match the number of parameters accepted by the function).

Syntax:

```
LocalScriptFunctionCall :
        FunctionName Arguments

FunctionName :
        Identifier

Arguments :
        ( )
        ( ArgumentList )
ArgumentList :
        AssignmentExpression
        ArgumentList , AssignmentExpression
```

Functions inside the same compilation unit can be called before the function has been declared:

```
function test2(param) {
  return test1(param+1);
};
function test1(val) {
  return val*val;
};
```

2.4.2.2 Library Functions

Library function calls must be used when the called function is a NIJA Script standard library function [NIJALibs].

Syntax:

```
LibraryFunctionCall :
        LibraryName . FunctionName Arguments

LibraryName :
        Identifier
```

A library function can be called by prefixing the function name with the name of the library (see 6.6 for more information) and the dot symbol (.):

```
function test4(param) {
return Float.sqrt(Lang.abs(param)+1);
};
```

2.4.3 Default Return Value

The default return value for a function is an empty string (""). Return values of functions can be ignored (ie, function call as a statement):

```
function test5() {
test4(4);
};
```

2.5 Statements

NIJA Script statements consist of expressions and keywords used with the appropriate syntax. A single statement may span multiple lines. Multiple statements may occur on a single line.

The following sections define the statements available in NIJA Script: empty statement, expression statement, block statement, break, continue, for, if...else, return, var, while.

2.5.1 Empty Statement

Empty statement is a statement that can be used where a statement is needed but no operation is required.

Syntax:

```
    EmptyStatement :
        ;
```

Examples:

```
    while (!poll(device)) ;  //  Wait until poll() is true
```

2.5.2 Expression Statement

Expression statements are used to assign values to variables, calculate mathematical expressions, make function calls etc.

Syntax:

```
    ExpressionStatement :
        Expression ;

    Expression :
        AssignmentExpression
        Expression , AssignmentExpression
```

Examples:

```
    str = "Hey " + yourName;
    val3 = prevVal + 4;
    counter++;
    myValue1 = counter, myValue2 = val3;
    alert("Watch out!");
    retVal = 16*Lang.max(val3,counter);
```

2.5.3 Block Statement

A set of statements enclosed in the curly brackets is a block statement. It can be used anywhere a single statement is needed.

Syntax:

```
    Block :
        { StatementListopt }
```

```
StatementList :
        Statement
        StatementList Statement
```

Example:

```
{
var i = 0;
var x = Lang.abs(b);
popUp("Remember!");
}
```

2.5.4 Variable Statement

This statement declares variables with initialisation (optional, variables are initialised to empty string ("") by default). The scope of the declared variable is the rest of the current function (see section 3.2.2 for more information about variable scoping).

Syntax:

```
VariableStatement :
        var VariableDeclarationList ;

VariableDeclarationList :
        VariableDeclaration
        VariableDeclarationList , VariableDeclaration

VariableDeclaration :
        Identifier VariableInitializeropt
VariableInitializer :
        = ConditionalExpression
```

Arguments: Identifier is the variable name. It can be any legal identifier. ConditionalExpression is the initial value of the variable and can be any legal expression. This expression (or the default initialisation to an empty string) is evaluated every time the variable statement is executed. Variable names must be unique within a single function.

Examples:

```
function count(str) {
  var result = 0; // Initialized once
  while (str != "") {
  var ind = 0; // Initialized every time
  // modify string
  };
return result
};
function example(param) {
  var a = 0;
   if (param > a) {
   var b = a+1; // Variables a and b can be used
   } else {
   var c = a+2; // Variables a, b and c can be used
   };
   return a; // Variable a, b and c are accessible
};
```

2.5.5 If Statement

This statement is used to specify conditional execution of statements. It consists of a condition and one or two statements and executes the first statement if the specified condition is true. If the condition is false, the second (optional) statement is executed.

Syntax:

```
IfStatement :
        if ( Expression ) Statement else Statement
        if ( Expression ) Statement
```

Arguments: Expression (condition) can be any NIJA Script expression that evaluates (directly or after conversion) to a boolean or an invalid value. If condition evaluates to true, the first statement is executed. If condition evaluates to false or invalid, the second (optional) else statement is executed. Statement can be any NIJA Script statement, including another (nested) if statement.else is always tied to the closest if.

Example:

```
if (sunShines) {
  myDay = "Good";
  goodDays++;
} else
  myDay = "Oh well...";
```

2.5.6 While Statement

This statement is used to create a loop that evaluates an expression and, if it is true, execute a statement. The loop repeats as long as the specified condition is true.

Syntax:

```
WhileStatement :
        while ( Expression ) Statement
```

Arguments: Expression (condition) can be any NIJA Script expression that evaluates (directly or after the conversion) to a boolean or an invalid value. The condition is evaluated before each execution of the loop statement. If this condition evaluates to true, the Statement is performed.
When condition evaluates to false or invalid, execution continues with the statement following Statement. Statement is executed as long as the condition evaluates to true.

Example:

```
var counter = 0;
var total = 0;
while (counter < 3) {
  counter++;
  total += c;
};
```

2.5.7 For Statement

This statement is used to create loops. The statement consists of three optional expressions enclosed in parentheses and separated by semicolons followed by a statement executed in the loop.

Syntax:

```
ForStatement :
  for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
```

Arguments: The first Expression or VariableDeclarationList (initialiser) is typically used to initialise a counter variable. This expression may optionally declare new variables with the var keyword. The scope of the defined variables is the rest of the function (see section 6.2.2 for more information about variable scoping).

The second Expression (condition) can be any NIJA Script expression that evaluates (directly or after the conversion) to a boolean or an invalid value. The condition is evaluated on each pass through the loop. If this condition evaluates to true, the Statement is performed. This conditional test is optional. If omitted, the condition always evaluates to true.

The third Expression (increment-expression) is generally used to update or increment the counter variable. Statement is executed as long as the condition evaluates to true.

Example:

```
for (var index = 0; index < 100; index++) {
  count += index;
  myFunc(count);
};
```

2.5.8 Break Statement

This statement is used to terminate the current while or for loop and continue the program execution from the statement following the terminated loop. It is an error to use break statement outside a while or a for statement.

Syntax:

BreakStatement :

```
break ;
```

Example:

```
function testBreak(x) {
  var index = 0;
  while (index < 6) {
    if (index == 3) break;
    index++;
  };
  return index*x;
};
```

2.5.9 Continue Statement

This statement is used to terminate execution of a block of statements in a while or for loop and
continue execution of the loop with the next iteration. Continue statement does not terminate the
execution of the loop:

- In a while loop, it jumps back to the condition.
- In a for loop, it jumps to the update expression.

It is an error to use continue statement outside a while or a for statement.

Syntax:

```
ContinueStatement :
        continue ;
```

Example:

```
var index = 0;
var count = 0;
while (index < 5) {
  index++;
if (index == 3)
    continue;
  count += index;
};
```

2.5.10 Return Statement

This statement can be used inside the function body to specify the function return value. If no return statement is specified or none of the function return statements is executed, the function returns an empty string by default.

Syntax:

ReturnStatement :

return Expression$_{opt}$;

Example:

```
function square( x ) {
  if (!(Lang.isFloat(x))) return invalid;
  return x * x;
};
```

2.6 Libraries

NIJA Script supports the usage of libraries. Libraries are named collections of functions that belong logically together. These functions can be called by using a dot ('.') separtor with the library name and the function name with parameters:
An example of a library function call:

```
function dummy(str) {
  var i = Str.elementAt(str,3," ");
};
```

2.6.1 Standard Libraries

Standard libraries are specified in more detail in the NIJA Script Standard Libraries Specification
[NIJALibs].

3. AUTOMATIC DATA TYPE CONVERSION RULES

In some cases, NIJA Script operators require specific data types as their operands. NIJA Script supports automatic data type conversions to meet the requirements of these operators. The following sections describe the different conversions in detail.

3.1 General Conversion Rules

NIJA Script is a weakly typed language and the variable declarations do not specify a type. However, internally the language handles the following data types:

- Boolean: represents a boolean value true or false.
- Integer: represents an integer value
- Floating-point: represents a floating-point value
- String: represents a sequence of characters
- Invalid: represents a type with a single value invalid

A variable at any given time can contain a value of one of these types. NIJA Script provides an operator typeof, which can be used to determine what is the current type of a variable or any expression (no conversions are performed).
Each NIJA Script operator accepts a predefined set of operand types. If the provided operands are not of the right data type an automatic conversion must take place. The following sections specify the legal automatic conversions between two data types.

3.1.1 Conversions to String

Legal conversions from other data types to string are:

- Integer value must be converted to a string of decimal digits that follows the numeric string grammar rules for decimal integer literals. See section 4.4 for more information about the numeric string grammar.
- Floating-point value must be converted to an implementation-dependent string representation that follows the numeric string grammar rules for decimal floating-point literals (see section 4.4 for more information about the numeric string grammar). The resulting string representation must be equal to the original value (ie .5 can be represented as "0.5", ".5e0", etc.).
- The boolean value true is converted to string "true" and the value false is converted to string "false".
- Invalid can not be converted to a string value.

3.1.2 Conversions to Integer

Legal conversions from other data types to integer are:

- A string can be converted into an integer value only if it contains a decimal representation of an integer number (see section 8.4 for the numeric string grammar rules for a decimal integer literal).
- Floating-point value cannot be converted to an integer value.
- The boolean value true is converted to integer value 1, false to 0.
- Invalid can not be converted to an integer value.

3.1.3 Conversions to Floating-Point

Legal conversions from other data types to floating-point are:

- A string can be converted into a floating-point value only if it contains a valid representation of a floating-point number (see section 4.4 for the numeric string grammar rules for a decimal floating-point literal).
- An integer value is converted to a corresponding floating-point value.
- The boolean value true is converted to a floating-point value 1.0, false to 0.0.
- Invalid can not be converted to a floating-point value.

The conversions between a string and a floating-point type must be transitive within the ability of the data types to accurately represent the value. A conversion could result in loss of precision.

3.1.4 Conversions to Boolean

Legal conversions from other data types to boolean are:

- The empty string ("") is converted to false. All other strings are converted to true.
- An integer value 0 is converted to false. All other integer numbers are converted to true.
- A floating-point value 0.0 is converted to false. All other floating-point numbers are converted to true.
- Invalid can not be converted to a boolean value.

3.1.5 Conversions to Invalid

There are no legal conversion rules for converting any of the other data types to an invalid type.
Invalid is either a result of an operation error or a literal value. In most cases, an operator that has an invalid value as an operand evaluates to invalid (see the operators in sections 2.3.8, 2.3.9 and 2.3.10 for the exceptions to this rule).

3.1.6 Summary

The following table contains a summary of the legal conversions between data types:

Given \ Used as:	**Boolean**	**Integer**	**Floating-point**	**String**
Boolean true	-	1	1.0	"true"
Boolean false	-	0	0.0	"false"
Integer 0	false	-	0.0	"0"
Any other integer	True	-	floating-point value of number	string representation of a decimal integer
Floatingpoint 0.0	False	Illegal	-	implementationdependent string representation of a floating-point value, e.g. "0.0"
Any other floating-point	True	Illegal	-	implementationdependent string representation of a floating-point value
Empty string	False	Illegal	Illegal	-
Non-empty string	True	integer value of its string representation (if valid – see section 8.4 for numeric string grammar for	floating-point value of its string representation (if valid – see section 8.4 for numeric string	-

		decimal integer literals) or illegal	grammar for decimal floating-point literals) or illegal	
invalid	Illegal	Illegal	Illegal	Illegal

3.2 Operator Data Type Conversion Rules

The previous conversion rules specify when a legal conversion is possible between two data types. NIJA Script operators use these rules, the operand data type and values to select the operation to be performed (in case the type is used to specify the operation) and to perform the data type conversions needed for the selected operation. The rules are specified in the following way:

- The additional conversion rules are specified in steps. Each step is performed in the given
  order until the operation and the data types for its operands are specified and the return value defined.
- If the type of the operand value matches the required type then the value is used as such.
- If the operand value does not match the required type then a conversion from the current data type to the required one is attempted:
  Legal conversion: Conversion can be done only if the general conversion rules (see section 3.1) specify a legal conversion from the current operator data type to the required one.
  Illegal conversion: Conversion can not be done if the general conversion rules (see section 3.1) do not specify a legal conversion from the current type to the required type.
- If a legal conversion rule is specified for the operand (unary) or for all operands then theconversion is performed, the operation performed on the converted values and the result returned as the value of the operation. If a legal conversion results in an invalid value then the operation returns an invalid value.
- If no legal conversion is specified for one or more of the operands then no conversion isperformed and the next step in the additional conversion rules is performed.

The following table contains the operator data type conversion rules based on the given operand data types:

Operand Types	Additional conversion rules	**Examples**
Boolean(s)	▪ If the operand is of type boolean or canbe converted into a boolean value18then perform a boolean operation and return its value, otherwise ▪ return invalid	true && 3.4 => boolean 1 && 0 => boolean "A" || "" => boolean !42 => boolean !invalid => invalid 3 && invalid => invalid
Integer(s)	▪ If the operand is of type integer or can be converted into an integer value18then perform an integer operation and return its value, otherwise ▪ return invalid	"7" << 2 => integer true << 2 => integer 7. 2 >> 3 => invalid 2.1 div 4 => invalid
Floatingpoint(s)	▪ If the operand is of type floating-point or can be converted into a floating-pointvalue18 then perform a floating-point operation and return its value, otherwise ▪ return invalid	-
String(s)	▪ If the operand is of type string or can beconverted into a string value18 then perform a string operation and return its value, otherwise ▪ return invalid	-
Integer or floating-point (unary)	▪ If the operand is of type integer or canbe converted into an integer value thenperform an integer operation and return its value, otherwise ▪ if the operand is of type floating-point or can be converted into a	+10 => integer -10.3 => float -"33" => integer +"47.3" => float +true => integer 1 -false => integer 0 "ABC" => invalid

	floating-point value then perform a floating-point operation and return its value, otherwise ▪ return invalid	-"9e9999" => invalid

Operand types	Additional conversion rules	Examples
Integers or floatingpoints	▪ If at least one of the operands is of type floating-point then convert the remaining operand to a floating-point value, perform a floating-point operation and return its value, otherwise ▪ if the operands are of type integer or can be converted into integer values then perform an integer operation and return its value, otherwise ▪ if the operands can be converted into floating-point values then perform a floating-point operation and return its value, otherwise ▪ return invalid	100/10.3 => float 33*44 => integer "10"*3 => integer 3.4*"4.3" => float "10"-"2" => integer "2.3"*"3" => float 3.2*"A" => invalid .9*"9e999" => invalid invalid*1 => invalid
Integers,floatingpoints or strings	▪ If at least one of the operands is of type string then convert the remaining operand to a string value, perform a string operation and return its value, otherwise ▪ if at least one of the operands is of type floating-point then convert the remaining operand to a floating-point value,	12+3 => integer 32.4+65 => float "12"+5.4 => string 43.2<77 => float "Hey"<56 => string 2.7+"4.2" => string 9.9+true => float 3<false => integer "A"+invalid => invalid

	perform a floating-point operation and return its value, otherwise ▪ if the operands are of type integer or can be converted into integer values then perform an integer operation and return its value, otherwise ▪ return invalid	

Any	Any type is accepted	a = 37.3 => float
		b = typeof "s" => string

3.3 Summary of Operators and Conversions

The following sections contain a summary on how the conversion rules are applied to NIJA Script operators and what are their possible return value types.

3.3.1 Single-Typed Operators

Operators that accept operands of one specific type use the general conversion rules directly. The following list contains all single type NIJA Script operators:

Operator	**Operand types**	**Result type**	**Operation performed**
!	Boolean	boolean	logical NOT (unary)
&&	Booleans	boolean	logical AND
||	Booleans	boolean	logical OR
~	integer	integer	bitwise NOT (unary)
<<	integers	integer	bitwise left shift
>>	integers	integer	bitwise right shift with sign
>>>	integers	integer	bitwise right shift with zero fill

&	integers	integer	bitwise AND
^	integers	integer	bitwise XOR
|	integers	integer	bitwise OR
%	integers	integer	remainder
<<=, >>=, >>>=, &=, ^=, |=	first operand: variable second operand: integer	integer	assignment with bitwise operation
%=, div=	first operand: variable second operand: integer	integer	assignment with numeric operation

3.3.2 Multi-Typed Operators

The following sections contain the operators that accept multi-typed operands:

Operator	**Operand types**	**Result type**	**Operation performed**
++	integer or floating-point	integer / floating-point	pre-or post-increment
--	integer or floating-point	integer / floating-point	pre-or post-decrement
+	integer or floating-point	integer / floating-point	unary plus
-	integer or floating-point	integer / floating-point	unary minus(negation)
*	integer or floating-point	integer / floating-point	multiplication
/	integer or floating-point	integer / floating-point	Division
-	integer or floating-point	integer / floating-point	substraction
+	integers, floatingpoints or strings	integer/floatin g-point/string	addition or string concatenation

<, <=	integers, floatingpoints or strings	boolean	less than, less than or equal
>, >=	integers, floatingpoints or strings	boolean	greater than, greater or equal
==	integers, floatingpoints or strings	boolean	equal (identical values)
!=	integers, floatingpoints or strings	boolean	not equal (different values)
*=, -=	first operand:variable second operand: integer or floatingpoint	integer/floating-point	assignment with numeric operation
/=	first operand: variable second operand: integer or floatingpoint	floating-point	assignment with division
+=	first operand:variable second operand: integer, floatingpoint or string	integer/floating-point/string	assignment with addition or concatenation
typeof	any	İnteger	return internal data type (unary)
isvalid	any	Boolean	check for validity (unary)
? :	first operand: boolean second operand: any third operand: any	any	conditional expression
=	first operand: variable second operand: any	any	assignment
,	first operand: any second operand: any	any	multiple evaluation

4. NIJA SCRIPT GRAMMAR

The grammars used in this specification are based on [ECMA262]. Since NIJA Script is not compliant with ECMAScript, the standard has been used only as the basis for defining NIJA Script language.

4.1 Context-Free Grammars

This section describes the context-free grammars used in this specification to define the lexical and syntactic structure of a NIJA Script program.

4.1.1 General

A context-free grammar consists of a number of productions. Each production has an abstract symbol called a nonterminal as its left-hand side and a sequence of one or more nonterminal and terminal symbols as its right-hand side. For each grammar, the terminal symbols are drawn from a specified alphabet.
A given context-free grammar specifies a language. It begins with a production consisting of a single distinguished nonterminal called the goal symbol followed by a (perhaps infinite) set of possible sequences of terminal symbols. They are the result of repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

4.1.2 Lexical Grammar

A lexical grammar for NIJA Script is given in section 8.2. This grammar has as its terminal symbols the characters of the Universal Character set of ISO/IEC-10646 ([ISO10646]). It defines a set of productions, starting from the goal symbol Input that describes how sequences of characters are translated into a sequence of input elements. Input elements other than white space and comments form the terminal symbols for the syntactic grammar for NIJA Script and are called NIJA Script tokens. These tokens are the reserved words, identifiers, literals and punctuators of the NIJA Script language. Simple white space and single-line comments are simply discarded and do not appear in the stream of input elements for the syntactic grammar. Likewise, a multi-line comment is simply discarded if it contains no line terminator; but if a multi-line comment contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.
Productions of the lexical grammar are distinguished by having two colons "::" as separating punctuation.

4.1.3 Syntactic Grammar

The syntactic grammar for NIJA Script is given in section 4.3. This grammar has NIJA Script tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions, starting from the goal symbol CompilationUnit, that describe how sequences of tokens can form syntactically correct NIJA Script programs.
When a stream of Unicode characters is to be parsed as a NIJA Script, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntax grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal CompilationUnit, with no tokens left over.
Productions of the syntactic grammar are distinguished by having just one colon ":" as punctuation.

4.1.4 Numeric String Grammar

A third grammar is used for translating strings into numeric values. This grammar is similar to the translating of the lexical grammar having to do with numeric literals and has as its terminal symbols the characters of the Unicode character set. This grammar appears in section 4.4. Productions of the numeric string grammar are distinguished by having three colons “:::” as punctuation.

4.1.5 Grammar Notation

Terminal symbols of the lexical and string grammars and some of the terminal symbols of the syntactic grammar, are shown in fixed width font, both in the productions of the grammars and throughout this specification whenever the text directly refers to such a terminal symbol. These are to appear in a program exactly as written.
Nonterminal symbols are shown in italic type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by one or more colons. (The number of colons indicates to which grammar the production belongs.) One or more alternative right-

hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

```
    WhileStatement :
        while ( Expression ) Statement
```

states that the nonterminal WhileStatement represents the token while, followed by a left parenthesis token, followed by an Expression, followed by a right parenthesis token, followed by a Statement. The occurrences of Expression and Statement are themselves nonterminals. As another example, the syntactic definition:

ArgumentList :

AssignmentExpression

ArgumentList , AssignmentExpression

states that an ArgumentList may represent either a single AssignmentExpression or an ArgumentList, followed by a comma, followed by an ssignmentExpression. This definition of ArgumentList is recursive, that is to say, it is defined in terms of itself. The result is that an ArgumentList may contain any positive number of arguments, separated by commas, where each argument expression is an AssignmentExpression. Such recursive definitions of nonterminals are common.

The subscripted suffix "opt", which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it. This means that:

```
    VariableDeclaration :
        Identifier VariableInitializeropt
```

is a convenient abbreviation for:

```
    VariableDeclaration :
        Identifier
        Identifier VariableInitializer
```

and that:

```
    IterationStatement :
        for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
```

is a convenient abbreviation for:

```
    IterationStatement :
        for ( ; Expressionopt ; Expressionopt ) Statement
        for ( Expression ; Expressionopt ; Expressionopt ) Statement
```

which in turn is an abbreviation for:

```
    IterationStatement :
```

for (; ; Expressionopt) Statement

for (; Expression ; Expressionopt) Statement

for (Expression ; ; Expressionopt) Statement

for (Expression ; Expression ; Expressionopt) Statement

which in turn is an abbreviation for:

IterationStatement :

for (; ;) Statement

for (; ; Expression) Statement

for (; Expression ;) Statement

for (; Expression ; Expression) Statement

for (Expression ; ;) Statement

for (Expression ; ; Expression) Statement

for (Expression ; Expression ;) Statement

for (Expression ; Expression ; Expression) Statement

therefore, the nonterminal IterationStatement actually has eight alternative right-hand sides.

Any number of occurrences of LineTerminator may appear between any two consecutive tokens in the stream of input elements without affecting the syntactic acceptability of the program.

When the words "one of" follow the colon(s) in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar for NIJA Script contains the production:

ZeroToThree :: one of

0 1 2 3

which is merely a convenient abbreviation for:

ZeroToThree ::

0

1

2

3

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multicharacter token, it represents the sequence of characters that would make up such a token.

The right-hand side of a production may specify that certain expansions are not permitted by using the phrase "but not" and then indicating the expansions to be excluded. For example, the
production:

    Identifier ::
        IdentifierName but not ReservedWord

means that the nonterminal Identifier may be replaced by any sequence of characters that could
replace IdentifierName provided that the same sequence of characters could not replace ReservedWord.
Finally, a few nonterminal symbols are described by a descriptive phrase in roman type in cases
where it would be impractical to list all the alternatives:

    SourceCharacter:
        any Unicode character

4.1.6 Source Text

NIJA Script source text is represented as a sequence of characters representable using the Universal Character set of ISO/IEC-10646 ([ISO10646]). Currently, this character set is identical to Unicode 2.0 ([UNICODE]). Within this document, the terms ISO10646 and Unicode are used
interchangeably and will indicate the same document character set.

    SourceCharacter ::
        any Unicode character

There is no requirement that NIJA Script documents be encoded using the full Unicode encoding (e.g. UCS-4). Any character encoding ("charset") that contains an inclusive subset of the characters in Unicode may be used (e.g. US-ASCII, ISO-8859-1, etc.).

Every NIJA Script program can be represented using only ASCII characters (which are equivalent to the first 128 Unicode characters). Non-ASCII Unicode characters may appear only within comments and string literals. In string literals, any Unicode character may also be expressed as a Unicode escape sequence consisting of six ASCII characters, namely \u plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal, the Unicode escape sequence contributes one character to the string value of the literal.

4.1.7 Character Set Resolution

When a Nİ-JA Script document is accompanied by external information (e.g. HTTP or MIME) there may be multiple sources of information available to determine the character encoding. In this case, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol. See, for example, the documentation of the "text/vnd.wap.NIJA Script" and "application/vnd.wap. NIJA Scriptc" MIME media types.
The pragma meta http equiv (see section 2.7.3.2), if present in the document, is never used to determine the character encoding.

If a NIJA Script document is transformed into a different format - for example, into the NIİJAScript bytecode (see chapter 10) - then the rules relevant for that format are used to determine the character encoding.

4.2 NIJA Script Lexical Grammar

The following contains the specification of the lexical grammar for NIJA Script:

```
        SourceCharacter ::
                any Unicode character
        WhiteSpace ::
        <TAB>
        <VT>
        <FF>
        <SP>
        <LF>
        <CR>
LineTerminator ::
        <LF>
        <CR>
        <CR><LF>
Comment ::
        MultiLineComment
        SingleLineComment
        MultiLineComment ::
```

/* MultiLineCommentChars$_{opt}$ */

MultiLineCommentChars ::

MultiLineNotAsteriskChar MultiLineCommentCharsopt

* PostAsteriskCommentCharsopt

PostAsteriskCommentChars ::

MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentCharsopt

* PostAsteriskCommentCharsopt

MultiLineNotAsteriskChar ::

SourceCharacter but not asterisk *

MultiLineNotForwardSlashOrAsteriskChar ::

SourceCharacter but not forward-slash / or asterisk *

SingleLineComment ::

// SingleLineCommentCharsopt

SingleLineCommentChars ::

SingleLineCommentChar SingleLineCommentCharsopt

SingleLineCommentChar ::

SourceCharacter but not LineTerminator

Token ::

ReservedWord

Identifier

Punctuator

Literal

ReservedWord ::

Keyword

KeywordNotUsedByNIJAScript

FutureReservedWord

BooleanLiteral

InvalidLiteral

Keyword :: one of

access	equiv	meta	while
agent	extern	name	url
break	for	path	
continue	function	return	
div	header	typeof	
div=	http	use	
domain	if	user	

else	isvalid	var	

KeywordNotUsedByNIJA Script :: one of

delete	null
in	this
lib	void
new	with

FutureReservedWord :: one of

case	default	finally	struct
catch	do	import	super
class	enum	private	switch
const	export	public	throw
debugger	extends	sizeof	try

Identifier ::

IdentifierName but not ReservedWord

IdentifierName ::

IdentifierLetter

IdentifierName IdentifierLetter

IdentifierName DecimalDigit

IdentifierLetter :: one of

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

_

DecimalDigit :: **one of**

0 1 2 3 4 5 6 7 8 9

Punctuator :: one of

=	>	<	==	<=	>=
!=	,	!	~	?	:
.	&&	||	++	--	+
-	*	/	&	|	^
%	<<	>>	>>>	+=	-=
*=	/=	&=	|=	^=	%=
<<=	>>=	>>>=	(	)	{
}	;	#			

```
Literal ::
        InvalidLiteral
        BooleanLiteral
        NumericLiteral
        StringLiteral

InvalidLiteral ::
        invalid
BooleanLiteral ::
        true
        false
NumericLiteral ::
        DecimalIntegerLiteral
        HexIntegerLiteral
        OctalIntegerLiteral
        DecimalFloatLiteral
DecimalIntegerLiteral ::
        0
        NonZeroDigit DecimalDigitsopt
NonZeroDigit :: one of
1       2       3       4       5       6       7       8       9

HexIntegerLiteral ::
        0x HexDigit
        0X HexDigit
        HexIntegerLiteral HexDigit
        HexDigit :: one of
        0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
OctalIntegerLiteral ::
        0 OctalDigit
        OctalIntegerLiteral OctalDigit

OctalDigit :: one of
        0       1       2       3       4       5       6       7
DecimalFloatLiteral ::
        DecimalIntegerLiteral . DecimalDigitsopt ExponentPartopt
        . DecimalDigits ExponentPartopt
```

DecimalIntegerLiteral ExponentPart

DecimalDigits ::

DecimalDigit

DecimalDigits DecimalDigit

ExponentPart ::

ExponentIndicator SignedInteger

ExponentIndicator :: one of

e E

SignedInteger ::

DecimalDigits

+ DecimalDigits

- DecimalDigits

StringLiteral ::

" DoubleStringCharacters$_{opt}$ "

' SingleStringCharacters$_{opt}$ '

DoubleStringCharacters ::

DoubleStringCharacter DoubleStringCharactersopt

SingleStringCharacters ::

SingleStringCharacter SingleStringCharactersopt

DoubleStringCharacter ::

SourceCharacter but not double-quote "or backslash \ or LineTerminator

EscapeSequence

SingleStringCharacter ::

SourceCharacter but not single-quote 'or backslash \ or LineTerminator

EscapeSequence

EscapeSequence ::

CharacterEscapeSequence

OctalEscapeSequence

HexEscapeSequence

UnicodeEscapeSequence

CharacterEscapeSequence ::

\ SingleEscapeCharacter

SingleEscapeCharacter :: one of

' " \ / b f n r t

HexEscapeSequence ::

\x HexDigit HexDigit

OctalEscapeSequence ::

```
        \ OctalDigit
        \ OctalDigit OctalDigit
        \ ZeroToThree OctalDigit OctalDigit
ZeroToThree :: one of
        0       1       2       3
UnicodeEscapeSequence ::
        \u HexDigit HexDigit HexDigit HexDigit
```

4.3 NIJA Script Syntactic Grammar

The following contains the specification of the syntactic grammar for NIJA Script:

```
PrimaryExpression :27
        Identifier
        Literal
        ( Expression )
CallExpression :
        PrimaryExpression
        LocalScriptFunctionCall
        ExternalScriptFunctionCall
        LibraryFunctionCall

LocalScriptFunctionCall :
        FunctionName Arguments
ExternalScriptFunctionCall :
        ExternalScriptName # FunctionName Arguments
LibraryFunctionCall :
        LibraryName . FunctionName Arguments
FunctionName :
        Identifier
ExternalScriptName :
        Identifier
LibraryName :
        Identifier
Arguments :
        ( )
        ( ArgumentList )
ArgumentList :
```

AssignmentExpression
ArgumentList , AssignmentExpression

PostfixExpression :
CallExpression
Identifier ++
Identifier --

UnaryExpression :
PostfixExpression
typeof UnaryExpression
isvalid UnaryExpression
++ Identifier
-- Identifier
+ UnaryExpression
- UnaryExpression
~ UnaryExpression
! UnaryExpression

MultiplicativeExpression :
UnaryExpression
MultiplicativeExpression * UnaryExpression
MultiplicativeExpression / UnaryExpression
MultiplicativeExpression div UnaryExpression
MultiplicativeExpression % UnaryExpression
AdditiveExpression :
MultiplicativeExpression
AdditiveExpression + MultiplicativeExpression
AdditiveExpression - MultiplicativeExpression

ShiftExpression :
AdditiveExpression
ShiftExpression << AdditiveExpression
ShiftExpression >> AdditiveExpression
ShiftExpression >>> AdditiveExpression

RelationalExpression :
ShiftExpression
RelationalExpression < ShiftExpression
RelationalExpression > ShiftExpression
RelationalExpression <= ShiftExpression

```
        RelationalExpression >= ShiftExpression
EqualityExpression :
        RelationalExpression
        EqualityExpression == RelationalExpression
        EqualityExpression != RelationalExpression
BitwiseANDExpression :
        EqualityExpression
        BitwiseANDExpression & EqualityExpression
BitwiseXORExpression :
        BitwiseANDExpression
        BitwiseXORExpression ^ BitwiseANDExpression
BitwiseORExpression :
        BitwiseXORExpression
        BitwiseORExpression | BitwiseXORExpression
LogicalANDExpression :
        BitwiseORExpression
        LogicalANDExpression && BitwiseORExpression
LogicalORExpression :
        LogicalANDExpression
        LogicalORExpression || LogicalANDExpression
ConditionalExpression :
        LogicalORExpression
        LogicalORExpression ? AssignmentExpression : AssignmentExpression
AssignmentExpression :
        ConditionalExpression
        Identifier AssignmentOperator AssignmentExpression
AssignmentOperator :: one of
        = *= /= %= += -= <<= >>= >>>= &= ^= |= div=
Expression :
        AssignmentExpression
        Expression , AssignmentExpression
Statement :
        Block
        VariableStatement
        EmptyStatement
        ExpressionStatement
        IfStatement
        IterationStatement
        ContinueStatement
        BreakStatement
        ReturnStatement

Block :
        { StatementListopt }

StatementList :
        Statement
        StatementList Statement
VariableStatement :
        var VariableDeclarationList ;
VariableDeclarationList :
        VariableDeclaration
        VariableDeclarationList , VariableDeclaration
VariableDeclaration :
        Identifier VariableInitializeropt
VariableInitializer :
        = ConditionalExpression
EmptyStatement :
        ;
ExpressionStatement :
        Expression ;
IfStatement 2
        if ( Expression ) Statement else Statement
        if ( Expression ) Statement
IterationStatement :
        WhileStatement
        ForStatement
WhileStatement :
        while ( Expression ) Statement
ForStatement :
        for ( Expressionopt ; Expressionopt ; Expressionopt ) Statement
        for ( var VariableDeclarationList ; Expressionopt ; Expressionopt ) Statement
ContinueStatement :
        continue ;
```

BreakStatement :

    break ;

ReturnStatement :

    return Expression$_{opt}$;

FunctionDeclaration :

    externopt function Identifier (FormalParameterListopt) Block ;opt

FormalParameterList :

    Identifier

    FormalParameterList , Identifier

CompilationUnit :

    Pragmasopt FunctionDeclarations

Pragmas :

    Pragma

    Pragmas Pragma

Pragma :

    use PragmaDeclaration ;

PragmaDeclaration :

    ExternalCompilationUnitPragma

    AccessControlPragma

    MetaPragma

ExternalCompilationUnitPragma :

    url Identifier StringLiteral

AccessControlPragma :

    access AccessControlSpecifier

AccessControlSpecifier :

    domain StringLiteral

    path StringLiteral

    domain StringLiteral path StringLiteral

MetaPragma :

    meta MetaSpecifier

MetaSpecifier :

    MetaName

    MetaHttpEquiv

    MetaUserAgent

MetaName :

    name MetaBody

MetaHttpEquiv :

http equiv MetaBody

MetaUserAgent :

user agent MetaBody

MetaBody :

MetaPropertyName MetaContent MetaSchemeopt

MetaPropertyName :

StringLiteral

MetaContent :

StringLiteral

MetaScheme :

StringLiteral

FunctionDeclarations :

FunctionDeclaration

FunctionDeclarations FunctionDeclaration

4.4 Numeric String Grammar

The following contains the specification of the numeric string grammar for NIJA Script. This grammar is used for translating strings into numeric values. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols the characters of the US-ASCII character set.

The following grammar can be used to convert strings into the following numeric literal values:

- Decimal Integer Literal: Use the following productions starting from the goal symbol StringDecimalIntegerLiteral.
- Decimal Floating-Point Literal: Use the following productions starting from the goal symbol StringDecimalFloatingPointLiteral.

StringDecimalIntegerLiteral :::

StrWhiteSpaceopt StrSignedDecimalIntegerLiteral StrWhiteSpaceopt

StringDecimalFloatingPointLiteral :::
StrWhiteSpaceopt StrSignedDecimalIntegerLiteral StrWhiteSpaceopt
StrWhiteSpaceopt StrSignedDecimalFloatingPointLiteral StrWhiteSpaceopt

StrWhiteSpace :::
StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar :::
<TAB>
<VT>
<FF>
<SP>
<LF>
<CR>

StrSignedDecimalIntegerLiteral :::
StrDecimalDigits
+ StrDecimalDigits
- StrDecimalDigits

StrSignedDecimalFloatingPointLiteral :::
StrDecimalFloatingPointLiteral
+ StrDecimalFloatingPointLiteral
- StrDecimalFloatingPointLiteral

StrDecimalFloatingPointLiteral :::
StrDecimalDigits . StrDecimalDigitsopt StrExponentPartopt
. StrDecimalDigits StrExponentPartopt
StrDecimalDigits StrExponentPart

StrDecimalDigits :::
StrDecimalDigit
StrDecimalDigits StrDecimalDigit

StrDecimalDigit ::: one of
0 1 2 3 4 5 6 7 8 9

StrExponentPart :::
StrExponentIndicator StrSignedInteger

StrExponentIndicator ::: one of
e E

StrSignedInteger :::
StrDecimalDigits
+ StrDecimalDigits
- StrDecimalDigits

5. NIJA SCRIPT BYTECODE INTERPRETER

The textual format of NIJA Script language must be compiled into a binary format before it can be interpreted by the NIJA Script bytecode interpreter. NIJA Script compiler encodes one NIJA Script compilation unit into NIJA Script bytecode using the encoding format presented in the chapter 10. A NIJA Script compilation unit (see section .1.3) is a unit containing pragmas and any number of NIJA Script functions. NIJA Script compiler takes one compilation unit as input and generates the NIJA Script bytecode as its output.

5.1 Interpreter Architecture

NIJA Script interpreter takes NIJA Script bytecode as its input and executes encoded functions as they are called. The following figure contains the main parts related to NIJA Script bytecode interpretation:

The NIJA Script interpreter can be used to call and execute functions in a compilation unit encoded as Nİ-JA Script bytecode. Each function specifies the number of parameters it accepts and the instructions used to express its behaviour. Thus, a call to a NIJA Script function must specify the function, the function call arguments and the compilation unit in which the function is declared. Once the execution completes normally, the NIJA Script interpreter returns the control and the return value back to the caller.
Execution of a NIJA Script function means interpreting the instructions residing in the NIJA Script bytecode. While a function is being interpreted, the NIJA Script interpreter maintains the following state information:

- IP (Instruction Pointer): This points to an instruction in the bytecode that is being interpreted.
- Variables: Maintenance of function parameters and variables.
- Operand stack: It is used for expression evaluation and passing arguments to called functions and back to the caller.
- Function call stack: NIJA Script function can call other functions in the current or separatecompilation unit or make calls to library functions. The function call stack maintains the information about functions and their return addresses.

5.2 Character Set

The NIJA Script Interpreter must use only one character set (native character set) for all of its string operations. Transcoding between different character sets and their encodings is allowed as long as the NIJA Script string operations are performed using only the native character set. The native character set can be requested by using the Lang library function Lang.characterSet() (see
[NIJASLibs])

5.3 Bytecode Semantics

The following sections describe the general encoding rules that must be used to generate NI-JA Script bytecode. These rules specify what the NIJA Script compiler can assume from the behaviour of the NIJA Script interpreter.

5.3.1 Passing of Function Arguments

Arguments must be present in the operand stack in the same order as they are presented in a NIJA Script function declaration at the time of a NIJA Script or library function call. Thus, the first argument is pushed into the operand stack first, the second argument is pushed next, etc. The instruction executing the call must pop the arguments from the operand stack and use them to initialise the appropriate function variables.

5.3.2 Allocation of Variable Indexes

A NIJA Script function refers to variables by using unique variable indexes. These indexes must match with the information specified for each called NIJA Script function: the number of arguments the function accepts and the number of local variables used by the function. Thus, the variable index allocation must be done using the following rules:

1) Function Arguments: Indexes for function arguments must be allocated first. The allocation must be done in the same order as the arguments are pushed into the operand stack (0 is allocated for the first argument, 1 for the second argument, etc.). The number of indexes allocated for function arguments must match the number of arguments accepted by the function. Thus, if the function accepts N arguments then the last variable index must be N-1. If the function does not accept any arguments (N = 0) then no variable indexes are allocated.

2) Local variables: Indexes for local variables must be allocated subsequently from the first variable index (N) that is not used for function arguments. The number of indexes allocated for local variables must match the number of local variables used by the function.

5.3.3 Automatic Function Return Value

NIJA Script function must return an empty string in case the end of the function is encountered without a return statement. The compiler can rely on the NIJA Script interpreter to automatically return an empty string every time the interpreter reaches the end of the function without encountering a return instruction.

5.3.4 Initialisation of Variables

The NIJA Script compiler should rely on the NIJA Script interpreter to initialise all function local variables initially to an empty string. Thus, the compiler does not have to generate initialization code for variables declared without initialisation.

5.4 Access Control

NIJA Script provides two mechanisms for controlling the access to the functions in the NIJA Script compilation unit: external keyword and a specific access control pragma. Thus, the NIJA Script interpreter must support the following behaviour:

- External functions: Only functions specified as external can be called from other compilation
- units (see 2.4).
- Access control: Access to the external functions defined inside a compilation unit is allowed from other compilation units that match the given access domain and access path definitions (see 2.7.2).

6. NIJA SCRIPT BINARY FORMAT

The following sections contain the specifications for the NIJA Script bytecode, a compact binary representation for compiled NIJA Script functions. The format was designed to allow for compact transmission over narrowband channels, with no loss of functionality or semantic information.

6.1 Conventions

The following sections describe the general encoding conventions and data types used to generate NIJA Script bytecode.

6.1.1 Used Data Types

The following data types are used in the specification of the NIJA Script Bytecode:

Data Type	Definition
bit	1 bit of data
byte	8 bits of opaque data
int8	8 bit signed integer (two's complement encoding)
U_int8	8 bit unsigned integer
int16	16 bit signed integer (two's complement encoding)
U_int16	16 bit unsigned integer
mb_u_int16	16 bit unsigned integer, in multi-byte integer format. See 6.1.2 for more information.
int32	32 bit signed integer (two's complement encoding)
U_int32	32 bit unsigned integer
mb_u_int32	32 bit unsigned integer, in multi-byte integer format. See 10.1.2 for more information.
float32	32 bit signed floating-point value in ANSI/IEEE Std 754-1985 [IEEE754] format.

Network byte order for multi-byte integer values is "big-endian". In other words, the most significant byte is transmitted on the network first followed subsequently by the less significant bytes. Network bit ordering for bit fields within a byte is "big-endian". In other words, bit fields described first are placed in the most significant bits of the byte.

6.1.2 Multi-byte Integer Format

This encoding uses a multi-byte representation for integer values. A multi-byte integer consists of a series of octets, where the most significant bit is the continuation flag and the remaining seven bits are a scalar value. The continuation flag is used to indicate that an octet is not the end of the multibyte sequence. A single integer value is encoded into a sequence of N octets. The first N-1 octets have the continuation flag set to a value of one (1). The final octet in the series has a continuation flag value of zero.
The remaining seven bits in each octet are encoded in a big-endian order, e.g., most significant bit first. The octets are arranged in a big-endian order, e.g. the most significant seven bits are transmitted first. In the situation where the initial octet has less than seven bits of value, all unused bits must be set to zero (0).
For example, the integer value 0xA0 would be encoded with the two-byte sequence 0x81 0x20. The integer value 0x60 would be encoded with the one-byte sequence 0x60.

6.1.3 Character Encoding

NIJA Script bytecode supports the following character encoding:

- UTF-8 (see [RFC2279])

Other character sets and their encodings are supported by a special string type (string with external character encoding definition, see 10.4.1) that does not explicitly specify the used character set or its encoding but assumes that this information is provided as part of the compilation unit itself (constant pool). The following rules must be applied when defining the used character encoding forthese special strings:

- If the value of the character set number in the constant pool is non-zero then this numberdefines the used character encoding (the number denotes the MIBEnum value assigned by the IANA for all character sets).
- If the value of the character set number in the constant pool is zero (0) then the character setis unknown.

The compiler must select one of these encodings to encode character strings in the NIJA Script bytecode.

NIJA Script language constructs, such as function names in NIJA Script, are written by using only a subset of Unicode character set i.e, a subset of US-ASCII characters. Thus, function names in the NIJA Script bytecode must use a fixed UTF-8 encoding.

6.1.4 Notational Conventions

NIJA Script bytecode is a set of bytes that represent NIJA Script functions in a binary format. It contains all the information needed by the NIJA Script interpreter to execute the encoded functions as specified. The bytecode can be divided into sections and subsections each of which containing a binary representation of a logical NIJA Script unit.

The NIJA Script bytecode structure and content is presented using the following table based notation:

Name	**Data type and size**	**Comment**
This is a name of a section inside the bytecode.	This specifies a data type and its size reserved for a section in case it cannot be divided into smaller subsections. Subsection specification is given in a separate table. Reference to the table is provided.	This gives a general overview of the meaning of this section.

The name of the next section. Any number of sections can be presented in one table.		
...		

The following conventions apply:

- Sections of bytecode are represented as rows in a table.
- Each section may be divided into subsections and represented in separate tables. In such case a reference to the subsection table is provided.
- Repetitive sections are denoted by section name followed by three dots (...).

6.2 NIJA Script Bytecode

The NIJA Script encoding contains two major elements: constant literals and the constructs needed to describe the behaviour of each NI-JA Script function. Thus, the NIJA Script bytecode consists of the following sections:

Name	**Data type and size**	**Comment**
HeaderInfo	See 10.3	Contains general information related to the bytecode.
ConstantPool	See 10.4	Contains the information of all constants specified as part of the NIJA Script compilation unit that are encoded into bytecode.
PragmaPool	See 10.5	Contains the information related to pragmas specified as part of the NIJA Script compilation unit that are encoded into bytecode.
FunctionPool	See 10.6	Contains all the information related to the encoding of functions and their behaviour.

The following sections define the encoding of these sections and their subsections in detail.

6.3 Bytecode Header

The header of the NIJA Script bytecode contains the following information:

Name	Data type and size	Comment
VersionNumber	Byte	Version number of the NIJA Script bytecode. The version byte contains the major version minus one in the upper 4 bits and the minor version in the lower 4 bits. The current version is 1.1. Thus, the version number must be encoded as 0x01.
CodeSize	mb_u_int32	The size of the rest of the bytecode (not including the version number and this variable) in bytes

6.4 Constant Pool

Constant pool contains all the constants used by the NIJA Script functions. Each of the constants has an index number starting from zero that is defined by its position in the list of constants. The instructions use this index to refer to specific constants.

Name	Data type and size	Comment
NumberOfConstants	mb_u_int16	Specifies how many constants are encoded in this pool.
CharacterSet	mb_u_int16	Specifies the character set used by the string constants in the constant pool. The character set is specified as an integer that denotes a MIBEnum value

		assigned by the IANA for all character sets (see [WSP] for more information).
Constants...	See 10.4.1	Contains the definitions for each constant in the constant pool. The number of constants is specified by NumberOfConstants.

6.4.1 Constants

Constants are stored into the bytecode one after each other. Encoding of each constant starts with the definition of its type (integer, floating-point, string etc.). It is being followed by constant type specific data that represents the actual value of the constant:

Name	**Data type and size**	**Comment**
ConstantType	u_int8	The type of the constant.
ConstantValue	See 10.4.1.1, 10.4.1.2 and 10.4.1.3	Type specific value definition

The following encoding for constant types is used:

Code	**Type**	**Encoding**
0	8 bit signed integer	6.4.1.1.1
1	1 16 bit signed integer	6.4.1.1.2
2	32 bit signed integer	6.4.1.1.3
3	32 bit signed floating-point	6.4.1.2
4	UTF-8 String	6.4.1.3.1
5	Empty String	6.4.1.3.2
6	String with external character encoding definition	6.4.1.3.3

7-255	Reserved for future use	

6.4.1.1 Integers

NI-JA Script bytecode supports 8 bit, 16 bit and 32 bit signed integer constants. The compiler can optimise the NIJA Script bytecode size by selecting the smallest integer constant type that can still hold the integer constant value.

6.4.1.1.1 8 Bit Signed Integer

8 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger8	int8	The value of the 8 bit signed integer constant.

6.4.1.1.2 16 Bit Signed Integer

16 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger16	int16	The value of the 16 bit signed integer constant.

6.4.1.1.3 32 Bit Signed Integer

32 bit signed integer constants are represented in the following format:

Name	Data type and size	Comment
ConstantInteger32	int32	The value of the 32 bit signed integer

		constant.

6.4.1.2 Floats

Floating-point constants are represented in 32-bit ANSI/IEEE Std 754-1985 [IEEE754] format:

Name	Data type and size	Comment
ConstantFloat32	float32	The value of the 32 bit floating point constant.

6.4.1.3 Strings

NIJA Script bytecode supports several ways to encode string constants into the constant pool. The compiler can select the most suitable character encoding supported by the client and optimise the NIJA Script bytecode size by selecting the smallest string constant type that can still hold the string constant value.

6.4.1.3.1 UTF-8 Strings

Strings that use UTF-8 encoding are encoded into the bytecode by first specifying their length and then the content:

Name	Data type and size	Comment

StringSizeUTF8	Mb_u_int32	The size of the following string in bytes (not containing this variable).
ConstantStringUTF8	StringSizeUTF8 Bytes	The value of the Unicode string (nonnull terminated) constant encoded using UTF-8. See 10.1.3 for more information about transfer encoding of strings.

6.4.1.3.2 Empty Strings

Empty strings do not need any additional encoding for their value.

6.4.1.3.3 Strings with External Character Encoding Definition

Strings that use external character encoding definition are encoded into the bytecode by first specifying their length and then the content:

Name	**Data type and size**	**Comment**
StringSizeExt	mb_u_int32	The size of the following string in bytes (not containing this field).
ConstantStringExt	StringSizeExt bytes	The value of the string (non-null terminated) constant using external character encoding definition. See 6.1.3 for more information about transfer encoding of strings.

6.6 Function Pool

The function pool contains the function definitions. Each of the functions has an index number starting from zero that is defined by its position in the list of functions. The instructions use this index to refer to specific functions.

Name	Data type and size	Comment
NumberOfFunctions	u_int8	The number of functions specified in this function pool.
FunctionNameTable	See 6.6.1	Function name table contains the names of all external functions present in the bytecode.
Functions...	See 6.6.2	Contains the bytecode for each function.

6.6.1 Function Name Table

The names of the functions that are specified as external (extern) are stored into a function name table.The names must be presented in the same order as the functions are represented in the function pool. Functions that are not specified as external are not represented in the function name table. The format of the table is the following:

Name	Data type and size	Comment
NumberOfFunctionNames	u_int8	The number of function names stored into the following table.
FunctionNames...	See 10.6.1.1	Each external function name represented in the same order as the functions are stored into the function pool.

6.6.1.1 Function Names

Function name is provided only for functions that are specified as external in NIJA Script. Each name is represented in the following manner:

Name	Data type and size	Comment
FunctionIndex	u_int8	The index of the function for which the following name is provided.
FunctionNameSize	u_int8	The size of the following function name in bytes (not including this variable).
FunctionName	FunctionNameSize Bytes	The characters of the function name encoded by using UTF-8. See 10.1.3 for more information about function name encoding.

6.6.2 Functions

Each function is defined by its prologue and code array:

Name	Data type and size	Comment
NumberOfArguments	u_int8	The number of arguments accepted

		by the function.
NumberOfLocalVariables	u_int8	The number of local variables used by the function (not including arguments).
FunctionSize	mb_u_int32	Size of the following CodeArray (not including this variable) in bytes.
CodeArray	See 6.6.2.1	Contains the code of the function.

6.6.2.1 Code Array

Code array contains all instructions that are needed to implement the behaviour of a NIJA Script function. See 7 for more information about NIJA Script instruction set.

Name	**Data type and size**	**Comment**
Instructions…	See chapter 7	The encoded instructions.

6.7 Limitations

The following table contains the limitations inherent in the selected bytecode format and instructions:

Maximum size of the bytecode	4294967295 bytes
Maximum number of constants in the constant pool	65535
Maximum number of different constant types	256
Maximum size of a constant string	4294967295

	bytes
Maximum length of function name	255
Maximum number of different pragma types	256
Maximum number of pragmas in the pragma pool	65536
Maximum number of functions in the function pool	255
Maximum number of function parameters	255
Maximum number of local variables / function	255
Maximum number of local variables and function Parameters	256
Maximum number of libraries	65536
Maximum number of functions / library	256

7. NIJA SCRIPT INSTRUCTION SET

The NIJA Script instruction set specifies a set of assembly level instructions that must be used to encode all NIJA Script language constructs and operations. These instructions are defined in such a way that they are easy to implement efficiently on a variety of Platforms.

7.1 Conversion Rules

The following table contains a summary of the conversion rules specified for the NIJA Script interpreter:

Rule – Operand type(s)	**Conversion**
1 – Boolean(s)	See the conversion rules for Boolean(s) in section Operator Data Type Conversion

	Rules (3.2)
2 – Integer(s)	See the conversion rules for Integer(s) in section Operator Data Type Conversion Rules (3.2)
3 – Floating-point(s)	See the conversion rules for Floatingpoint(s) in section Operator Data Type Conversion Rules (3.2)
4 – String(s)	See the conversion rules for String(s) in section Operator Data Type Conversion Rules (3.2)
5 – Integer or floating-point (unary)	See the conversion rules for Integer or floating-point (unary) in section Operator Data Type Conversion Rules (3.2)
6 – Integers or floating-points	See the conversion rules for Integers or floating-points in section Operator Data Type Conversion Rules (3.2)
7 – Integers, floating-points or strings	See the conversion rules for Integers, floating-points or strings in section Operator Data Type Conversion Rules (3.2)
8 - Any	See the conversion rules for Any in section Operator Data Type Conversion Rules (3.2)

7.2 Fatal Errors

The following table contains a summary of the fatal errors specified for the NIJA Script interpreter:

Error code:	**Fatal Error:**

1 (Verification Failed)	See section Verification Failed (9.3.1.1) for details.
2 (Fatal Library Function Error)	See Section Fatal Library Function Error (9.3.1.2) for details.
3 (Invalid Function Arguments)	See Section Invalid Function Arguments (9.3.1.3) for details
4 (External Function Not Found)	See Section External Function Not Found (9.3.1.4) for details.
5 (Enable to Load Compilation Unit)	See Section Enable to Load Compilation Unit (9.3.1.5) for details.
6 (Access Violation)	See Section Access Violation (9.3.1.6) for details.
7 (Stack Under Flow)	See Section Stack Under Flow (9.3.1.7) for details.
8 (Programmed Abort)	See Section Programmed Abort (9.3.2.1) for details.
9 (Stack Over Flow)	See Section Stack Over Flow (9.3.3.1) for details*.
10 (Out Of Memory)	See Section Out of Memory (9.3.3.2) for details*.
11 (User Initiated)	See Section User Initiated (9.3.4.1) for details*.
12 (System Initiated)	See Section System Initiated (9.3.4.2) for details*.

* These fatals errors are not related to copputation but can be genareted as a result of memory exhaustion or external signals.

8. RUN-TIME ERROR DETECTION AND HANDLING

Since Nİ-JA Script functions are used to implement services for users that expect the terminals (in particular mobile phones) to work properly in all situations, error handling is of utmost importance.
This means that while the language does not provide, for example, an exception mechanism, it should provide tools to either prevent errors from happening or tools to notice them and take appropriate actions. Aborting a program execution should be the last resort used only in cases where nothing else is possible.
The following section lists errors that can happen when downloading bytecode and executing it. It does not contain programming errors (such as infinite loop etc.). For these cases a user controlled abortion mechanism is needed.

8.1 Error Detection

The goal of error detection is to give tools for the programmer to detect errors (if possible) that would lead to erroneous behaviour. Since NIJA Script is a weakly typed language, special functionality has been provided to detect errors that are caused by invalid data types :
· Check that the given variable contains the right value: Nİ-JA Script supports type validation library [NIJASLibs] functions such as Lang.isInt(), Lang.isFloat(), Lang.toInt() and Lang.toFloat().
· Check that the given variable contains a value that is of right type: NIJA Script supports the operators typeof and isvalid that can be used for this purpose.

8.2 Error Handling

Error handling takes place after an error has already happened. This is the case when the error could not be prevented by error detection (memory limits, external signals etc.) or it would have been too difficult to do so (overflow, underflow etc.). These cases can be divided into two classes:

· *Fatal errors:* These are errors that cause the program to abort. Since NIJA Script functions are always called from some other user agents, program abortion should always be signalled to the calling user agent. It is then its responsibility to take the appropriate actions to signal the user of errors.

· *Non-fatal errors:* These are errors that can be signalled back to the program as special return values and the program can decide on the appropriate action.

The following error descriptions are divided into sections based on their fatality.

8.3 Fatal Errors

8.3.1 Bytecode Errors

These errors are related to the bytecode and the instructions being executed by the NIJA Script
Bytecode Interpreter. They are indications of erroneous constant pool elements, invalid instructions, invalid arguments to instructions or instructions that cannot be completed.

8.3.1.1 Verification Failed

Description:	Reports that the specified bytecode for the called compilation unit did not pass the verification about bytecode verification.
Generated:	At any time when a program attempts to call an external function.
Example:	**var a = 3*OtherScript#doThis(param);**
Severity:	Fatal
Predictable:	Is detected during the bytecode verification.
Solution:	Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.1.2 Fatal Library Function Error

Description:	Reports that a call to a library function resulted in a fatal error.
Generated:	At any time when a call to a library function is used (CALL_LIB).

Typically, this is an unexpected error in the library function implementation.

Example: **var a = Str.Format(param);**

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script
interpreter.

8.3.1.3 Invalid Function Arguments

Description: Reports that the number of arguments specified for a function call do not match with the number of arguments specified in the called function.

Generated: At any time a call to an external function is used (CALL_URL).

Example: Compiler generates an invalid parameter to an instruction or the number of parameters in the called function has changed.

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.1.4 External Function Not Found

Description: Reports that a call to an external function could not be found from the specified compilation unit.

Generated: At any time, when a program attempts to call an external function (CALL_URL).

Example: **var a = 3*OtherScript#doThis(param);**

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.1.5 Unable to Load Compilation Unit

Description: Reports that the specified compilation unit could not be loaded dueto unrecoverable errors in accessing the compilation unit in the network server or the specified compilation unit does not exist in the network server.

Generated: At any time, when a program attempts to call an external function (CALL_URL).

Example: **var a = 3*OtherScript#doThis(param);**

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.1.6 Access Violation

Description: Reports an access violation. The called external function resides in a protected compilation unit.

Generated: At any time when a program attempts to call an external function (CALL_URL).

Example: **var a = 3*OtherScript#doThis(param);**

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.1.7 Stack Underflow

Description:	Indicates a stack underflow because of a program error (compiler generated bad code).
Generated:	At any time when a program attempts to pop an empty stack.
Example:	Only generated if compiler generates bad code.
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.2 Program Specified Abortion

This error is generated when a Nİ-JA Script function calls the library function Lang.Abort() (see [NIJASLibs]) to abort the execution.

8.10.2.1 13.3.2.1 Programmed Abort

Description:	Reports that the execution of the bytecode was aborted by a call to Lang.abort() function.
Generated:	At any time when a program makes a cal to Lang.abort() function.
Example:	**Lang.abort("Unrecoverable error");**
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.3 Memory Exhaustion Errors

These errors are related to the dynamic behaviour of the NIJA Script interpreter (see section 5.1 for more information) and its memory usage.

8.3.3.1 Stack Overflow

Description:	Indicates a stack overflow.
Generated:	At any time when a program recourses too deep or attempts to push too many variables onto the operand stack.
Example:	**function f(x) { f(x+1); };**
Severity:	Fatal.
Predictable:	No.
Solution:	Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.3.2 Out of Memory

Description: Indicates that no more memory resources are available to the nterpreter.

Generated: At any time when the operating system fails to allocate more space for the interpreter.

Example:

```
function f(x) {
   x=x+ "abcdefghijklmnopqrstuvzyxy";
   f(x);
};
```

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.4 External Exceptions

The following exceptions are initiated outside of the NIJA Script Bytecode Interpreter.

8.10.4.1 13.3.4.1 User Initiated

Description: Indicates that the user wants to abort the execution of the program (reset button etc.)

Generated: At any time.

Example: User presses reset button while an application is running.

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.3.4.2 System Initiated

Description: Indicates that an external fatal exception occurred while a program is running and it must be aborted. Exceptions can be originated from a low battery, power off, etc.

Generated: At any time.

Example: The system is automatically switching off due to a low battery.

Severity: Fatal.

Predictable: No.

Solution: Abort program and signal an error to the caller of the NIJA Script interpreter.

8.4 Non-Fatal Errors

8.4.1 Computational Errors

These errors are related to arithmetic operations supported by the NIJA Script.

8.4.1.1 Divide by Zero

Description: Indicates a division by zero.

Generated: At any time when a program attempts to divide by 0 (integer or floating-point division or remainder).

Example:

```
var a = 10;
var b = 0;
```

var x = a / b;

var y = a div b;

var z = a % b;

a /= b;

Severity: Non-fatal.

Predictable: Yes.

Solution: The result is an invalid value.

8.4.1.2 Integer Overflow

Description: Reports an arithmetic integer overflow.

Generated: At any time when a program attempts to execute an integer operation.

Example:

var a =Lang.maxInt();

var b = Lang.maxInt();

var c = a + b;

Severity: Non-fatal.

Predictable: Yes (but difficult in certain cases).

Solution: The result is an invalid value.

8.4.1.3 Floating-Point Overflow

Description: Reports an arithmetic floating-point overflow.

Generated: At any time when a program attempts to execute a floating-point operation.

Example:

var a = 1.6e308;

var b = 1.6e308;

var c = a * b;

Severity: Non-fatal.

Predictable: Yes (but difficult in certain cases).

Solution: The result is an invalid value.

8.4.1.4 Floating-Point Underflow

Description: Reports an arithmetic underflow.

Generated: At any time when the result of a floating-point operation is smaller than what can be represented.

Example:

```
var a = Float.precision();
var b = Float.precision();
var c = a * b;
```

Severity: Non-fatal.

Predictable: Yes (but difficult in certain cases).

Solution: The result is a floating-point value 0.0.

8.4.2 Constant Reference Errors

These errors are related to run-time references to constants in the constant pool.

8.4.2.1 Not a Number Floating-Point Constant

Description: Reports a reference to a floating-point literal in the constant pool that is Not a Number [IEEE754].

Generated: At any time when a program attempts to access a floating-point literal and the compiler has generated a Not a Number as a floatingpoint constant.

Example: A reference to a floating-point literal.

Severity: Non-fatal.

Predictable: Yes.

Solution:	The result is an invalid value.

8.4.2.2 Infinite Floating-Point Constant

Description:	Reports a reference to a floating-point literal in the constant pool that is either positive or negative infinity [IEEE754].
Generated:	At any time when a program attempts to access a floating-point literal and the compiler has generated a floating-point constant with a value of positive or negative infinity.
Example:	A reference to a floating-point literal.
Severity:	Non-fatal.
Predictable:	Yes.
Solution:	The result is an invalid value.

8.4.2.3 Illegal Floating-Point Reference

Description:	Reports an erroneous reference to a floating-point value in the constant pool.
Generated:	At any time when a program attempts to use floating-point values and the environments supports only integer values.
Example:	**var a = 3.14;**
Severity:	Non-fatal.
Predictable:	Can be detected during the run-time.
Solution:	The result is an invalid value.

8.4.3 Conversion Errors

These errors are related to automatic conversions supported by the NIJA Script.

8.4.3.1 Integer Too Large

Description: Indicates a conversion to an integer value where the integer value is too large (positive/negative).

Generated: At any time when an application attempts to make an automatic conversion to an integer value.

Example: **var a = -99999999999999999999999999999999999999999";**

Severity: Non-fatal.

Predictable: No.

Solution: The result is an invalid value.

8.4.3.2 Floating-Point Too Large

Description: Indicates a conversion to a floating-point value where the floating point value is too large (positive/negative).

Generated: At any time when an application attempts to make an automatic conversion to a floating-point value.

Example: **var a = -"9999999.9999999999e99999";**

Severity: Non-fatal.

Predictable: No.

Solution: The result is an invalid value.

8.4.3.3 Floating-Point Too Small

Description: Indicates a conversion to a floating-point value where the floatingpoint value is too small (positive/negative).

Generated: At any time when an application attempts to make an automatic conversion to a floating-point value.

Example: **var a = -"0.01e-99";**

Severity: Non-fatal.

Predictable: No.

Solution: The result is a floating-point value 0.0.

8.5 Library Calls and Errors

Since NIJA Script supports the usage of libraries, there is a possibility that errors take place inside the library functions. Design and the behaviour of the library functions are not part of the NIJA Script language specification. However, following guidelines should be followed when designing libraries:

· Provide the library users mechanisms by which errors can be detected before they happen.

· Use the same error handling mechanisms as NIJA Script operators in cases where error should be reported back to the caller.

· Minimise the possibility of fatal errors in all library functions.

9. SUPPORT FOR INTEGER ONLY DEVICES

The NIJA Script language has been designed to run also on devices that do not support floatingpoint operations. The following rules apply when NIJA Script is used with such devices:

- Variables can only contain the following internal data types:

  * Boolean
  * Integer
  * String
  * Invalid

- Any LOAD_CONST bytecode that refers to a floating point constant in the constant pool will push an invalid value on the operand stack instead of the constant value.
- Division (/) operation returns always an invalid value.
- Assignment with division (/=) operation always results in an invalid value.
- All conversion rules related to floating-points are ignored.

The programmer can use Lang.float() [NIJASLibs] to test (during the run-time) if floating-point operations are supported.

NIJAScript Standard Libraries
Specification

Approved Version 04-September-2002

Wireless Application Protocol
NIJAScript Standard Libraries Specification
Version 1.1

1. SCOPE

This document specifies the library interfaces for the standard set of libraries supported by NIJAScript [NIJAScript]
NIJAScript is compiled into bytecode *before* it is being sent to the client. This way the narrowband communication channels available today can be optimally utilized and the memory requirements for the client kept to the minimum. For the same reasons, many of the advanced features of the JavaScript language have been removed to make the language both optimal, easier to compile into bytecode and easier to learn.
Library support has been added to the NIJAScript to replace some of the functionality that has been removed from ECMAScript in accordance to make the NIJAScript more efficient. This feature provides access to built-in functionality and a means for future expansion without unnecessary overhead.
The following chapters describe the set of libraries defined to provide access to core functionality of a NIJAScript client. This means that all libraries, except *Floa*t, are present in the client's scripting environment. Float library is optional and only supported with clients that can support floating-point arithmetic operations.

1.1. DEFINITIONS AND ABBREVIATIONS

1.1 .1 Definitions

Bytecode - content encoding where the content is typically a set of low-level opcodes (ie, instructions) and operands for a targeted hardware (or virtual) machine.
Client - a device (or application) that initiates a request for connection with a server.

Content - subject matter (data) stored or generated at an origin server. Content is typically displayed or interpreted by a user agent in response to a user request.

Content Encoding - when used as a verb, content encoding indicates the act of converting a data
object from one format to another. Typically the resulting format requires less physical space than
the original, is easier to process or store and/or is encrypted. When used as a noun, content encoding specifies a particular format or encoding standard or process.

Content Format – actual representation of content.

Device - a network entity that is capable of sending and receiving packets of information and has a unique device address. A device can act as both a client or a server within a given context or across multiple contexts.

NIJAScript - a scripting language used to program the low level hardware device. NIJAScript is one of the originating technologies of ECMAScript.

1.1.2 Abbreviations

For the purposes of this specification, the following abbreviations apply:

API	Application Programming Interface
ECMA	European Computer Manufacturer Association
LSB	Least Significant Bits
MSB	Most Significant Bits
RFC	Request For Comments
UI	User Interface
WTP	Wireless Transport Protocol
WAE	Wireless Application Environment
WTA	Wireless Telephony Applications
WTAI	Wireless Telephony Applications Interface
WBMP	Wireless BitMaP

1. 2. NOTATIONAL CONVENTIONS

The libraries in this document are represented by providing the following information:
NAME: Library name. The syntax of the library name follows the syntax specified in the [NIJAScript] specification. Library names are case sensitive.

Examples: Lang, Str

LIBRARY ID: The numeric identifier reserved for the library to be used by the NIJAScript Compiler. The range of values reserved for this identifier is divided into the following two categories:
0 .. 32767 Reserved for standard libraries.
32768 … 65535
Reserved for future use.

DESCRIPTION: A short description of the library and used conventions.
Each function in the library is represented by providing the following information:

FUNCTION: Specifies the function name and the number of function parameters. The syntax of the function name follows the syntax specified in the [NIJAScript] specification. Function names are case sensitive.
Example: Abs(*value*)
Usage: var a = 3*Lang.Abs(length);

FUNCTION ID: The numeric identifier reserved for the function to be used by the NIJAScript Compiler. The range of values reserved for this identifier is: **0..255**.

DESCRIPTION:
Describes the function behaviour and its parameters.

PARAMETERS:
Specifies the function parameter types.
Example: *value* = Number

RETURN VALUE:
Specifies the type(s) of the return value.
Example: String or invalid.
EXCEPTIONS: Describes the possible special exceptions and error codes and the

corresponding return values. Standard errors, common to all functions, are not described here (see 6.3 for more information about error handling).

Example: If the *value1* <= 0 and *value2* < 0 and not an integer then invalid is returned.

EXAMPLE: **Gives a few examples of how the function could be used.**

var a = -3;

var b = Lang.Abs(a); // b = 3

1.3. NIJASCRIPT COMPLIANCE

NIJAScript standard library functions provide a mechanism to extend the NIJAScript language. Thus, the specified library functions must follow the NIJAScript conventions and rules.

1.3.1 Supported Data Type

The following NIJAScript types [NIJAScript] are used in the function definitions to denote the type of both the function parameters and return values:

*Boolea*n, *Intege*r, *Floa*t, *String* and *Invalid*

In addition to these, *number* can be used to denote a parameter type when both integer and floating-point parameter value types are accepted. *Any* can be used when the type can be any of the supported types.

1.3.2 Data Type Conversions

Since NIJAScript is a weakly typed language, the conversions between the data types are done automatically if necessary (see [NIJAScript] for more details about data type conversion rules). The library functions follow NIJAScript operator data type conversion rules except where explicitly stated otherwise.

1.3.3 Error Handling

Error cases are handled in the same way as in the NIJAScript language (see [NIJAScript] for more details):

An invalid function argument results in an invalid return value with no other side effects unless explicitly stated otherwise.

A function argument that cannot be converted to the required parameter type results in an invalid return value with no side effects. See 6.2 for more information about data type conversions.

Function dependent error cases are handled by returning a suitable error code specified in each function definition. These errors are documented as part of the function specification (exceptions).

1.3.4 Support for Integer-Only Devices

The NIJAScript language has been designed to run also on devices that do not support floating-point operations. The NIJAScript standard libraries have operations that require floating-point support. Thus, the following rules apply when the libraries are implemented for an integer-only device:

*Library functions accept arguments of the following type only: *boolea*n, *intege*r, *string* and *invali*d.

*All conversion rules related to floating-point data are ignored.

**Lang.toFloat()* function returns invalid.

**Str.Format()* function returns invalid when type *f* is specified in the format.

*All *Float* (see chapter 2) library functions return invalid.

2. LANG

NAME: Lang

LIBRARY ID: 0

DESCRIPTION:

This library contains a set of functions that are closely related to the
NIJAScript language core.

2.1 Abs

FUNCTION: Abs(*value*)

FUNCTION ID: 0

DESCRIPTIOSN:

Returns the absolute value of the given number. If the given number is of type integer then an integer value is returned. If the given number is of type floating-point then a floating-point value is returned.

PARAMETERS:

value = Number

RETURN VALUE:

Number or invalid.

EXCEPTIONS:

EXAMPLE: **var a = -3;**

var b = Lang.Abs(a); // b = 3

2.2 Min

FUNCTION: Min(*value1, value*2)

FUNCTION ID: 1

DESCRIPTION:

Returns the minimum value of the given two numbers. The value and type returned is the same as the value and type of the selected number. The selection is done in the following way:

- NIJAScript operator data type conversion rules for *integers and floating-points* (see [NIJAScript]) must be used to specify the data type (integer or floating-point) for comparison.

- Compare the numbers to select the smaller one.
- If the values are equal then the first value is selected.

PARAMETERS:

value1 = Number

value2 = Number

RETURN VALUE:

Number or invalid.

EXCEPTIONS:

-

EXAMPLE:

var c = Lang.Min(a,b); // c = -3

var d = Lang.Min(45, 76.3); // d = 45 (integer)

var e = Lang.Min(45, 45.0); // e = 45 (integer)

2.3 Max

FUNCTION: Max(*value1, value*2)

FUNCTION ID: 2

DESCRIPTION:

Returns the maximum value of the given two numbers. The value and type returned is the same as the value and type of the selected number. The selection is done in the following way:

- NIJAScript operator data type conversion rules for *integers and floating-points* (see [NIJAScript]) must be used to specify the data type (integer or floating-point) for comparison.

- Compare the numbers to select the larger one.
- If the values are equal then the first value is selected.

PARAMETERS:

value1 = Number

value2 = Number

RETURN VALUE:

Number or invalid.

EXCEPTIONS:

-

EXAMPLE:

var c = Lang.Max(a,b); // c = 3

var d = Lang.Max(45.5, 76); // d = 76 (integer)

var e = Lang.Max(45.0, 45); // e = 45.0 (float)

2.4 toInt

FUNCTION: toInt(*value*)

FUNCTION ID: 3

DESCRIPTION:

Returns an integer value defined by the string *value*. The legal integer syntax is specified by the NIJAScript (see [NIJAScript]) numeric string grammar for *decimal integer literals* with the following additional parsing rule:

- Parsing ends when the first character is encountered that is not a leading '+' or '-' or a decimal digit.

The result is the parsed string converted to an integer value.

PARAMETERS:

value = String

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

In case of a parsing error an invalid value is returned.

EXAMPLE:

var i = Lang.parseInt("1234"); // i = 1234

var j = Lang.parseInt(" 100 m/s"); // j = 100

2.5 toFloat

FUNCTION: toFloat(*value*)

FUNCTION ID: 4

DESCRIPTION:

Returns a floating-point value defined by the string *value*. The legal floating-point syntax is specified by the NIJAScript (see [NIJAScript]) numeric string grammar for *decimal floating-point literals* with the following additional parsing rule:

- Parsing ends when the first character is encountered that cannot be parsed as being part of the floating-point representation.

The result is the parsed string converted to a floating-point value.

PARAMETERS:

value = String

RETURN VALUE:

Floating-point or invalid.

EXCEPTIONS:

In case of a parsing error an invalid value is returned.

If the system does not support floating-point operations then an invalid value is returned.

EXAMPLE:

var a = Lang.parseFloat("123.7"); // a = 123.7

```
var b = Lang.parseFloat(" +7.34e2 Hz"); // b = 7.34e2
var c = Lang.parseFloat(" 70e-2 F"); // c = 70.0e-2
var d = Lang.parseFloat("-.1 C"); // d = -0.1
var e = Lang.parseFloat(" 100 "); // e = 100.0
var f = Lang.parseFloat("Number: 5.5"); // f = invalid
var g = Lang.parseFloat("7.3e meters"); // g = invalid
var h = Lang.parseFloat("7.3e- m/s"); // h = invalid
```

2.6 isInt

FUNCTION: isInt(*value*)

FUNCTION ID: 5

DESCRIPTION:

Returns a boolean value that is true if the given *value* can be converted into an integer number by using parseInt(*value*). Otherwise false is returned.

PARAMETERS:

value = Any

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Lang.isInt(" -123"); // true
var b = Lang.isInt(" 123.33"); // true
var c = Lang.isInt("string"); // false
var d = Lang.isInt("#123"); // false
var e = Lang.isInt(invalid); // invalid
```

2.7 isFloat

FUNCTION: isFloat(*value*)

FUNCTION ID: 6

DESCRIPTION:

Returns a boolean value that is true if the given *value* can be converted into a floating-point number using parseFloat(*value*). Otherwise false is returned.

WMLScript:

value = Any

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS: If the system does not support floating-point operations then an invalid value is returned.

EXAMPLE:

```
var a = Lang.isFloat(" -123"); // true
var b = Lang.isFloat(" 123.33"); // true
var c = Lang.isFloat("string"); // false
var d = Lang.isFloat("#123.33"); // false
var e = Lang.isFloat(invalid); // invalid
```

2.8 maxInt

FUNCTION: maxInt()

FUNCTION ID: 7

DESCRIPTION:

Returns the maximum integer value.

PARAMETERS:

-

RETURN VALUE:

Integer 2147483647.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Lang.maxInt();
```

2.9 minInt

FUNCTION: minInt()

FUNCTION ID: 8

DESCRIPTION:

Returns the minimum integer value.

PARAMETERS:

-

RETURN VALUE:

Integer –2147483648.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Lang.minInt();
```

2.10 Random

FUNCTION: Random(*value*)

FUNCTION ID: 11

DESCRIPTION:

Returns an integer value with positive sign that is greater than or equal to 0 but less than or equal to the given *value*. The return value is chosen randomly or pseudo-randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy.
If the *value* is of type floating-point, *Float.Int()* is first used to calculate the actual integer *value*.

PARAMETERS:

value = Number

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

If *value* is equal to zero (0), the function returns zero.

If *value* is less than zero (0), the function returns invalid.

EXAMPLE:

```
var a = 10;
var b = Lang.Random(5.1)*a; // b = 0..50
var c = Lang.Random("string"); // c = invalid
```

2.11 Seed

FUNCTION: seed(*value*)

FUNCTION ID: 12

DESCRIPTION:

Initializes the pseudo-random number sequence and returns an empty string. If the *value* is zero or a positive integer then the given *value* is used for initialization, otherwise a random, system dependent initialization value is used.
If the *value* is of type floating-point, *Float.Int()* is first used to calculate the

actual integer *value*.

PARAMETERS:

value = Number

RETURN VALUE:

String or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Lang.seed(123); // a = ""
var b = Lang.Random(20); // b = 0..20
var c = Lang.seed("seed"); // c = invalid (random seed left
// unchanged)
```

3. FLOAT

NAME: Float

LIBRARY ID: 1

DESCRIPTION:

This library contains a set of typical arithmetic floating-point functions that are frequently used by applications.
The implementation of these library functions is *optional* and implemented only by devices that can support floating-point operations (see 6.4). If floating-point operations are not supported, all functions in this library must return invalid.

3.1. int

FUNCTION: Int(*value*)

FUNCTION ID: 0

DESCRIPTION:

Returns the integer part of the given value. If the *value* is already an integer, the result is the *value* itself.

PARAMETERS:

value = Number

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = 3.14;
var b = Float.Int(a); // b = 3
var c = Float.Int(-2.8); // c = -2
```

3.2 Floor

FUNCTION: Floor(*value*)

FUNCTION ID: 1

DESCRIPTION:

Returns the greatest integer value that is not greater than the given *value*. If the *value* is already an integer, the result is the *value* itself.

PARAMETERS:

value = Number

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = 3.14;
var b = Float.Floor(a); // b = 3
var c = Float.Floor(-2.8); // c = -3
```

3.3 ceil

FUNCTION: Ceil(*value*)

FUNCTION ID: 2

DESCRIPTION:

Returns the smallest integer value that is not less than the given *value*. If the *value* is already an integer, the result is the *value* itself.

PARAMETERS:

value = Number

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = 3.14;
var b = Float.Ceil(a); // b = 4
var c = Float.Ceil(-2.8); // c = -2
```

3.4 pow

FUNCTION: Pow(*value1, value*2)

FUNCTION ID: 3

DESCRIPTION:

Returns an implementation-dependent approximation to the result of raising *value1* to the power of *value*2. If *value1* is a negative number then *value2* must be an integer.

PARAMETERS:

value1 = Number

value2 = Number

RETURN VALUE:

Floating-point or invalid.

EXCEPTIONS:

If *value1* == 0 and *value2* < 0 then invalid is returned.

If *value1* < 0 and *value2* is not an integer then invalid is returned.

EXAMPLE:

```
var a = 3;
var b = Float.Pow(a,2); // b = 9
```

3.5 round

FUNCTION: Round(*value*)

FUNCTION ID: 4

DESCRIPTION:

Returns the number value that is closest to the given *value* and is equal to a mathematical integer. If two integer number values are equally close to the *valu*e, the result is the larger number value. If the *value* is already an integer, the result is the *value* itself.

PARAMETERS:

value = Number

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Float.Round(6.7); // a = 7
var a = Float.Round(3.4); // a = 3
var a = Float.Round(3.5); // a = 4
var b = Float.Round(-3.5); // b = -3
var c = Float.Round(0.5); // c = 1
var d = Float.Round(-0.5); // b = 0
```

3.6 sqrt

FUNCTION: Sqrt(*value*)

FUNCTION ID: 5

DESCRIPTION:

Returns an implementation-dependent approximation to the square root of the given *value*.

PARAMETERS:

value = Floating-point

RETURN VALUE:

Floating-point or invalid.

EXCEPTIONS:

If *value* is a negative number then invalid is returned.

EXAMPLE:

```
var a = 4;
var b = Float.sqrt(a); // b = 2.0
var c = Float.sqrt(5); // c = 2.2360679775
```

3.7 maxFloat

FUNCTION: maxFloat()

FUNCTION ID: 6

DESCRIPTION:

Returns the maximum floating-point value supported by [IEEE754] single precision floating-point format.

PARAMETERS:

-

RETURN VALUE:

Floating-point 3.40282347E+38.

EXCEPTIONS:

-

EXAMPLE:

var a = Float.maxFloat();

3.8 minFloat

FUNCTION: minFloat()

FUNCTION ID: 7

DESCRIPTION:

Returns the smallest nonzero floating-point value supported by [IEEE754] single precision floating-point format.

PARAMETERS:

-

RETURN VALUE:

Floating-point. Smaller than or equal to the normalised minimum single precision floating-point value: 1.17549435E-38.

EXCEPTIONS:

-

EXAMPLE:

var a = Float.minFloat();   // 1.17549435E-38

4. STRING

NAME: String

LIBRARY ID: 2

DESCRIPTION:

This library contains a set of string functions. A string is an array of characters. Each of the characters has an index. The first character in a string has an index zero (0). The length of the string is the number of characters in the array.
The user of the String library can specify a special *separator* by which *elements* in a string can be separated. These elements can be accessed by specifying the separator and the element index. The first element in a string has an index zero (0). Each occurrence of the separator in the string separates two elements (no escaping of separators is allowed).
A *White space character* is one of the following characters:
- TAB: Horizontal Tabulation
- VT: Vertical Tabulation
- FF: Form Feed
- SP: Space
- LF: Line Feed
- CR: Carriage Return

4.1 length

FUNCTION: length(*string*)

FUNCTION ID: 0

DESCRIPTION:

Returns the length (number of characters) of the given *string*.

PARAMETERS:

string = String

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "ABC";
var b = Str.Length(a); // b = 3
var c = Str.Length(""); // c = 0
var d = Str.Length(342); // d = 3
```

4.2 isEmpty

FUNCTION: isEmpty(*string*)

FUNCTION ID: 1

DESCRIPTION:

Returns a boolean true if the string length is zero and boolean false otherwise.

PARAMETERS:

string = String

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "Hello";
var b = "";
var c = Str.isEmpty(a); // c = false;
var d = Str.isEmpty(b); // d = true
var e = Str.isEmpty(true); // e = false
```

4.3 charAt

FUNCTION: charAt(*string*, *index*)

FUNCTION ID: 2

DESCRIPTION:

Returns a new string of length one containing the character at the specified *index* of the given *string*.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual integer *index*.

PARAMETERS:

string = String

index = Number (the index of the character to be returned)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

If *index* is out of range then an empty string ("") is returned.

EXAMPLE:

```
var a = "My name is Joe";
var b = Str.charAt(a, 0); // b = "M"
var c = Str.charAt(a, 100); // c = ""
var d = Str.charAt(34, 0); // d = "3"
```

var d = Str.charAt(a, "first"); // e = invalid

4.4 subString

FUNCTION: subString(*string, startIndex, length*)

FUNCTION ID: 3

DESCRIPTION:

Returns a new string that is a substring of the given *string*. The substring begins at the specified *startIndex* and its length (number of characters) is the given *length*. If the *startIndex* is less than 0 then 0 is used for the *startIndex*. If the *length* is larger than the remaining number of characters in the string, the *length* is replaced with the number of remaining characters. If the *startIndex* or the *length* is of type floating-point, *Float.Int()* is first used to calculate the actual integer value.

PARAMETERS:

string = String

startIndex = Number (the beginning index, inclusive)

length = Number (the length of the substring)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

If *startIndex* is larger than the last index an empty string ("") is returned.

If *length* <= 0 an empty string ("") is returned.

EXAMPLE:

```
var a = "ABCD";
var b = Str.subString(a, 1, 2); // b = "BC"
var c = Str.subString(a, 2, 5); // c = "CD"
var d = Str.subString(1234, 0, 2); // d = "12"
```

4.5 find

FUNCTION: find(*string*, *subString*)

FUNCTION ID: 4

DESCRIPTION:

Returns the index of the first character in the *string* that matches the requested *subStrin*g. If no match is found integer value –1 is returned.

Two strings are defined to match when they are *identica*l. Characters with multiple possible representations match only if they have the same representation in both strings. No case folding is performed.

PARAMETERS:

string = String

subString = String

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

If *subString* is an empty string (""), an invalid value is returned.

EXAMPLE: **var a = "abcde";**

var b = Str.find(a, "cd"); // b = 2

var c = Str.find(34.2, "de"); // c = -1

var d = Str.find(a, "qz"); // d = -1

var e = Str.find(34, "3"); // e = 0

var f = Str.find(a, ""); // f = invalid

4.6 replace

FUNCTION: replace(*string, oldSubString, newSubStrin*g)

FUNCTION ID: 5

DESCRIPTION:

Returns a new string resulting from replacing all occurrences of *oldSubString* in this string with *newSubStrin*g.

Two strings are defined to match when they are *identica*l. Characters with multiple possible representations match only if they have the same representation in both strings. No case folding is performed.

PARAMETERS:

string = String

oldSubString = String

newSubString = String

RETURN VALUE:

String or invalid.

EXCEPTIONS:

If *oldSubString* is an empty string an invalid value is returned.

EXAMPLE:

```
var a = "Hello Joe. What is up Joe?";
var newName = "Don";
var oldName = "Joe";
var c = Str.replace(a, oldName, newName);
// c = "Hello Don. What is up Don?";
var d = Str.replace(a, newName, oldName);
// d = "Hello Joe. What is up Joe?"
```

4.7 elements

FUNCTION: elements(*string, separato*r)

FUNCTION ID: 6

DESCRIPTION:

Returns the number of elements in the given *string* separated by the given *separato*r. Empty string ("") is a valid element (thus, this function can never return a value that is less or equal to zero).

PARAMETERS:

string = String

separator = String (the first character of the string used as separator)

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

Returns invalid if the *separator* is an empty string.

EXAMPLE:

```
var a = "My name is Joe; Age 50;";
var b = Str.elements(a, " "); // b = 6
var c = Str.elements(a, ";"); // c = 3
var d = Str.elements("", ";"); // d = 1
var e = Str.elements("a", ";"); // e = 1
var f = Str.elements(";", ";"); // f = 2
var g = Str.elements(";;,;", ";,"); // g = 4 separator = ;
```

4.8 elementAt

FUNCTION: elementAt(*string*, *index, separato*r)

FUNCTION ID: 7

DESCRIPTION:

Search *string* for *index*'th element, elements being separated by *separator* and return the corresponding element. If the *index* is less than 0 then the first element is returned. If the *index* is larger than the number of elements then the last element is returned. If the *string* is an empty string then an empty string is returned.
If the *index* is of type floating-point, *Float.Int()* is first used to calculate the actual *index* value.

PARAMETERS:

string = String

index = Number (the index of the element to be returned)

separator = String (the first character of the string used as separator)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

Returns invalid if the *separator* is an empty string.

EXAMPLE:

```
var a = "My name is Joe; Age 50;";
var b = Str.elementAt(a, 0, " "); // b = " My"
var c = Str.elementAt(a, 14, ";"); // c = ""
var d = Str.elementAt(a, 1, ";"); // d = " Age 50"
```

4.9 removeAt

FUNCTION: removeAt(*string*, *index, separator*)

FUNCTION ID: 8

DESCRIPTION:

Returns a new string where the element and the corresponding *separator* (if existing) with the given *index* are removed from the given *string*. If the *index* is less than 0 then the first element is removed. If the *index* is larger than the number of elements then the last element is removed. If the *string* is empty, the function returns a new empty string.
If the *index* is of type floating-point, *Float.Int()* is first used to calculate the actual *index* value.

PARAMETERS:

string = String

index = Number (the index of the element to be deleted)

separator = String (the first character of the string used as separator)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

Returns invalid if the *separator* is an empty string.

EXAMPLE:

```
var a = "A A; B C D";
var s = " ";
var b = Str.removeAt(a, 1, s);
// b = "A B C D"
var c = Str.removeAt(a, 0, ";");
// c = " B C D"
var d = Str.removeAt(a, 14, ";");
// d = "A A"
```

4.10 replaceAt

FUNCTION: replaceAt(*string*, *element, index, separator*)

FUNCTION ID: 9

DESCRIPTION:

Returns a string with the current element at the specified *index* replaced with the given *element*. If the *index* is less than 0 then the first element is replaced. If the *index* is larger than the number of elements then the last element is replaced. If the *string* is empty, the function returns a new string with the given *element*.

If the *index* is of type floating-point, *Float.Int()* is first used to calculate the actual *index* value.

PARAMETERS:

string = String

element = String

index = Number (the index of the element to be replaced)

separator = String (the first character of the string used as separator)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

Returns invalid if the *separator* is an empty string.

EXAMPLE:

```
var a = "B C; E";
var s = " ";
var b = Str.replaceAt(a, "A", 0, s);
// b = "A C; E"
var c = Str.replaceAt(a, "F", 5, ";");
// c = "B C;F"
```

4.11 insertAt

FUNCTION: insertAt(*string*, *element, index, separator*)

FUNCTION ID: 10

DESCRIPTION:

Returns a string with the *element* and the corresponding *separator* (if needed) inserted at the specified element *index* of the original *string*. If the *index* is less than 0 then 0 is used as the *index*. If the *index* is larger than the number of elements then the element is appended at the end of the *string*. If the *string* is empty, the function returns a new string with the given *element*.

If the *index* is of type floating-point, *Float.int()* is first used to calculate the actual *index* value.

PARAMETERS:

string = String (original string)

element = String (element to be inserted)

index = Number (the index of the element to be added)

separator = String (the first character of the string used as separator)

RETURN VALUE:

String or invalid.

EXCEPTIONS:

Returns invalid if the *separator* is an empty string.

EXAMPLE:

```
var a = "B C; E";
var s = " ";
var b = Str.insertAt(a, "A", 0, s);
```

```
// b = "A B C; E"
var c = Str.insertAt(a, "X", 3, s);
// c = "B C; E X"
var d = Str.insertAt(a, "D", 1, ";");
// d = "B C;D; E"
var e = Str.insertAt(a, "F", 5, ";");
// e = "B C; E;F"
```

4.12 squeeze

FUNCTION: squeeze(*string*)

FUNCTION ID: 11

DESCRIPTION:

Returns a string where all consecutive series of white spaces within the *string* are reduced to single inter-word space.

PARAMETERS:

String = String

RETURN VALUE:

String or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "Hello";
var b = " Bye Jon . \r\n See you! ";
var c = Str.squeeze(a); // c = "Hello";
var d = Str.squeeze(b); // d = " Bye Jon . See you! ";
```

4.13 trim

FUNCTION: Trim(*string*)

FUNCTION ID: 12

DESCRIPTION:

Returns a string where all trailing and leading white spaces in the given *string* have been trimmed.

PARAMETERS:

String = String

RETURN VALUE:

String or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "Hello";
var b = " Bye Jon . See you! ";
var c = Str.trim(a); // c = "Hell o"
var d = Str.trim(b); // d = "Bye Jon . See you!"
```

4.14 compare

FUNCTION: compare(*string1, string*2)

FUNCTION ID: 13

DESCRIPTION:

The return value indicates the lexicographic relation of *string1* to *string*2. The relation is based on the relation of the character codes in the native character set. The return value is –1 if *string1* is less than *string*2, 0 if *string1* is identical to *string2* or 1 if *string1* is greater than *string*2.

PARAMETERS:

String1 = String

String2 = String

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "Hello";
var b = "Hello";
var c = Str.compare(a, b); // c = 0
var d = Str.compare("Bye", "Jon"); // d = -1
var e = Str.compare("Jon", "B ye"); // e = 1
```

4.15 toString

FUNCTION: toString(*valu*e)

FUNCTION ID: 14

DESCRIPTION:

Returns a string representation of the given *valu*e. This function performs exactly the same conversions as supported by the [NIJAScript] language (automatic conversion from boolean, integer and floating-point values to strings) except that invalid value returns the string "invalid".

PARAMETERS:

value = Any

RETURN VALUE:

String.

EXCEPTIONS:

-

EXAMPLE:

```
var a = Str.toString(12); // a = "12"
var b = Str.toString(true); // b = "true"
```

4.16 Format

FUNCTION: Format(*forma*t, *valu*e)

FUNCTION ID: 15

DESCRIPTION:

Converts the given *value* to a string by using the given formatting provided as a *format* string. The format string can contain only one format specifier, which can be located anywhere inside the string. If more than one is specified, only the first one (leftmost) is used and the remaining specifiers are replaced by an empty string. The format specifier has the following form:

% [width] [.precision] type

The **width** argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left until the minimum width is reached. The width argument never causes the

value to be truncated. If the number of characters in the output value is greater than the specified width or, if width is not given, all characters of the *value* are printed (subject to the precision argument).
The **precision** argument specifies a nonnegative decimal integer, preceded by a period (.), that can be used to set the precision of the output value. The interpretation of this value depends on the given type:
d Specifies the minimum number of digits to be printed. If the number of digits in the *value* is less than precision, the output value is padded on the left with zeroes. The value is not truncated when the number of digits exceeds precision. Default precision is 1. If precision is specified as 0 and the value to be converted is 0, the result is an empty string.
f Specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. Default precision is 6; if precision is 0 or if the period (.) appears without a number following it, no decimal point is printed.
s Specifies the maximum number of characters to be printed. By default, all characters are printed.
Unlike the width argument, the precision argument can cause either truncation of the output value or rounding of a floating-point value.
The **type** argument is the only required format argument; it appears after any optional format fields. The type character determines whether the given *value* is interpreted as integer, floating-point or string. The supported type arguments are:
d *Integer*: The output value has the form [-]dddd, where dddd is one or more decimal digits.
f *Floating-point*: The output value has the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number and the number of digits after the decimal point depends on the requested precision. When the number of digits after the decimal point in the *value* is less than the precision, letter 0 should be padded to fill columns (e.g. the result of Str.Format("%2.3f", 1.2) will -be "1.200")
s *String*: Characters are printed up to the end of the string or until the precision value is reached. When the width is larger than

precision, the width should be ignored.

Percent character (%) in the format string can be presented by preceding it with another percent character (%%).

PARAMETERS:

format = String

value = Any

RETURN VALUE:

String or invalid.

EXCEPTIONS:

Illegal format specifier results in an invalid return value.

If type **f** is specified in format argument and floating-point numbers are not supported, invalid is returned.

EXAMPLE:

```
var a = 45;
var b = -45;
var c = "now";
var d = 1.2345678;
var e = Str.Format("e: %6d", a); // e = "e: 45"
var f = Str.Format("%6d", b); // f = " -45"
var g = Str.Format("%6.4d", a); // g = " 00 45"
var h = Str.Format("%6.4d", b); // h = " –0045"
var i = Str.Format("Do it %s", c); // i = "Do it now"
var j = Str.Format("%3f", d); // j = "1.234568"
var k = Str.Format("%10.2f%%", d); // k = " 1.23%"
var l = Str.Format("%3f %2f.", d); // l = "1.234568 ."
var m = Str.Format("%.0d", 0); // m = ""
var n = Str.Format("%7d", "Int"); // n = invalid
var o = Str.Format("%s", true); // o = "true"
```

5. PROTEK

NAME: Pro

LIBRARY ID: 3

DESCRIPTION:

This library contains a set of conversion of string and hexadecimal functions that are frequently used by applications.

5.1 toAscii

FUNCTION: toAscii(*string*)

FUNCTION ID: 0

DESCRIPTION:

Returns the integer value of the given character.

PARAMETERS:

value = string

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

-

EXAMPLE:

```
var a = "A";
var b = Pro.toAscii(a);     // b = 65
var c = Pro.toAscii(1);     // c = 49
```

5.2 toChar

FUNCTION: toChar(*value*)

FUNCTION ID: 1

DESCRIPTION:

Returns the ascii value of the given integer value. Value must be integer type. If it is not, then call Float.Int(value) function to convert integer.

PARAMETERS:

value = Number

RETURN VALUE:

Character or invalid.

EXCEPTIONS:

EXAMPLE:

```
var a = 65;
var b = Pro.toChar(a);                      // b = A
var c = Pro.toChar(13)+ Pro.toChar(10);   // CRLF
```

5.3 StrToHex

FUNCTION: StrToHex(*string*)

FUNCTION ID: 2

DESCRIPTION:

Returns the hexadecimal number of the given each character in string. If the parameter is not string type returns *invalid.*

PARAMETERS:

value = String

RETURN VALUE:

String or invalid.

EXCEPTIONS:

- Returns invalid. if the parameter type is not string type.

EXAMPLE:

var a = "abcd";

var b = Pro.StrToHex(a);   // b = 61626364

5.4 HexToStr

FUNCTION: HexToStr(*string*)

FUNCTION ID: 3

DESCRIPTION:

Returns the string character of the given each two number in string. If the parameter is not string type returns *invalid.*

PARAMETERS:

value = String

RETURN VALUE:

String or invalid.

EXCEPTIONS:

- Returns invalid. if the parameter type is not string type.

EXAMPLE:

var a = "4142434445";

var b = Pro.HexToStr(a);   // b = ABCDE

5.5 IntToHex

FUNCTION: IntToHex(value1, value2)

FUNCTION ID: 4

DESCRIPTION:

IntToHex converts a number into a string containing the number's hexadecimal (base 16) representation. Value is the number to convert. Value2 indicates the minimum number of hexadecimal digits to return.

Value1 and value2 are converted to integer type using NIJAScript conversion rules. If the parameters are not integer type returns *invalid.*

PARAMETERS:

Value1 = Value

Value1 = Value

RETURN VALUE:

String or invalid.

EXCEPTIONS:

- Returns invalid. if the parameters type is not integer type.

EXAMPLE:

```
var a = “9999”;
var b = Pro.IntToHex(a,2);      // b = 270F
var c = Pro. IntToHex(64,2);         // c = 40
```

5.6 HexToInt

FUNCTION: HexToInt(string)

FUNCTION ID: 5

DESCRIPTION:

Returns the integer value of the given hexadecimal value in the string, If the parameter is not string type returns *invalid.*

PARAMETERS:

Value = String

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

- Returns invalid. if the parameters type is not string type.

EXAMPLE:

var a = 999;

var b = Pro.HexToInt(a);   // b = 2457

5.7 subShort

FUNCTION: subShort(string,value)

FUNCTION ID: 6

DESCRIPTION:

Returns two bytes with the reference of the value index. Every two element denote one byte.if the value is not interger then call Float.Int() otherwise type will be invalid. First index is 0 as usual.

PARAMETERS:

string = String

value = Value

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

- Returns invalid. if the string is not string type.

EXAMPLE:

var a = 234567436;

var b = Pro.subShort(a,1);   // b = 3456

var a = 234;

var b = Pro.subShort(a,1);   // b = 34

5.8 subLong

FUNCTION: subLong(string,value)

FUNCTION ID: 7

DESCRIPTION:

Returns four bytes with the reference of the value index. Every two element denote one byte.if the value is not interger then call Float.Int() otherwise type will be invalid. First index is 0 as usual.

PARAMETERS:

string = String

value = Value

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

- Returns invalid. if the string is not string type.

EXAMPLE:

```
var a = 23426789;
var b = Pro.subLong(a,2);      // b = 426789
```

5.9 swapInt

FUNCTION: swapInt(value)

FUNCTION ID: 8

DESCRIPTION:

Exchanges the first byte and second bytes of integer value and then set zero third and fourth bytes.

Returns swapped value.

PARAMETERS:

value = Value

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

- If value is not integer type return invalid

EXAMPLE:

```
var a = 16;
var b = Pro.swapInt(a);     // b = 4096
var c = Pro.swapInt(b);     // b = 16
```

5.10 swapLong

FUNCTION: swapLong(value)

FUNCTION ID: 9

DESCRIPTION:

Exchanges the first byte , the fourth byte, and the second byte , the third byte each other them.

Returns swapped value.

PARAMETERS:

value = Value

RETURN VALUE:

Integer or invalid.

EXCEPTIONS:

- If value is not integer type return invalid

EXAMPLE:

```
var a = 16;
var b = Pro.swapLong(a);     // b = 268435456
var c = Pro.swapLong(b);     // b = 16
```

6. Vps

NAME: Vps

LIBRARY ID: 133

DESCRIPTION:

This library contains a set of LCD, card and SAM(Security Access Module) functions.

6.1 Clear

FUNCTION: Clear(void)

FUNCTION ID: 0

DESCRIPTION:

Clears display and returns Boolean value.

PARAMETERS:

-

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.Clear();   // a = True

6.2 Print

FUNCTION: Print(value1, value2,string);

FUNCTION ID: 1

DESCRIPTION:

Prints the string on the screen with the given colon and the row coordinates which is given by value1 and value2 respectively.

Row and colon information starts from 1.

Value1 and value2 must be integer type then convert them by using conversion rules (For more information see NIJAScript conversion rules at the NIJAScript Specification).

PARAMETERS:

- Value1 = Value
- Value2 = Value
- String = string

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.Print(1,1,” Protekila ”); // a = True

6.3 Cursor

FUNCTION: Cursor(value1, value2,value3);

FUNCTION ID: 2

DESCRIPTION:

Decides the cursor position on the LCD screen via value1 and value2, and the value3 determines whether blink or not. For instance if value3 is 1, cursor blanks, if it’s 0 then stops blink.

PARAMETERS:

- Value1 = Value
- Value2 = Value
- Value3 = Value

RETURN VALUE:

Boolean or invalid.

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.Cursor(10,2,1); // a = True

6.4 CurHide

FUNCTION: CurHide (void);

FUNCTION ID: 3

DESCRIPTION:

-Hides cursor.

PARAMETERS:

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.CurHide();   // a = True

6.5 Backlight

FUNCTION: Backlight (value);

FUNCTION ID: 4

DESCRIPTION:

- Turns on back light of the LCD.

PARAMETERS:

- Value = value

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.BackLight(1);   // a = True

6.6 Contrast

FUNCTION: Contrast (value);

FUNCTION ID: 5

DESCRIPTION:

- Increases contrast adjustment.

PARAMETERS:

- Value = value

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

```
var a = Vps.Contrast(1);      // a = True
```

6.7 DayTime

FUNCTION: DayTime (value,string);

FUNCTION ID: 8

DESCRIPTION:

- Format day time.

PARAMETERS:

- value = value
- string = string

RETURN VALUE:

Return formatted value as string type.

EXCEPTIONS:

-value must be integer type and string must string type

If not return invalid.

EXAMPLE:

```
s=Vps.Clock();                      // s=12345678
n=Vps.DayTime(s,'d/m/y h:u:s');     // n="dd/mm/yy hh:uu:ss"
```

6.7 Keyboard

FUNCTION: Keyboard (value);

FUNCTION ID: 10

DESCRIPTION:

- Visible Keypad

Keys are enabled by the given parameter. Value must be integer.

If value = 0 Only Keypad is visible, not enabled.

If value = 1 Only first row of the keypad is enabled.

If value = 2 Only first two rows of the keypad is enabled.

If value = 3 Only first three rows of the keypad is enabled.

If value = 4 All keypad is enabled.

PARAMETERS:

- Value = value

RETURN VALUE:

Boolean

EXCEPTIONS:

- If value > 4 or value < 0 return Invalid.

EXAMPLE:

var a = Vps.Keyboard(1);   // a = True

6.8 KeyRead

FUNCTION: KeyRead (void);

FUNCTION ID: 11

DESCRIPTION:

- Read a key from keypad and return string value of pressed.

PARAMETERS:

-

RETURN VALUE:

String

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.KeyRead();   // a = pressed key

6.9 Remote

FUNCTION: Remote (value);

FUNCTION ID: 12

DESCRIPTION:

- Visible remote control.

If value = 1 remote control is enabled.

If value = 0 remote control is disabled.

PARAMETERS:

- Value = value

RETURN VALUE:

Empty string

EXCEPTIONS:

- Value must be 0 or 1 if it is not then returns invalid.

EXAMPLE:

var a = Vps.Remote(1); // a = “” Visible remote control at the simulator

var b = Vps.Remote(0); // b = “” Invisible remote control at the Simulator

6.10 RemRead

FUNCTION: RemRead (void);

FUNCTION ID: 13

DESCRIPTION:

Read a key from remote control and return string value of pressed.

PARAMETERS:

-

RETURN VALUE:

String

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.RemRead(); // a = pressed key

6.11 LedGreen

FUNCTION: LedGreen (value);

FUNCTION ID: 15

DESCRIPTION:

Turns on green led

If value > 0 then turns on green led.

PARAMETERS:

- value

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.LedGreen(1);   // a = True

6.12 LedRed

FUNCTION: LedRed (value);

FUNCTION ID: 16

DESCRIPTION:

Turns on red led.

If value > 0 then turns on red led.

PARAMETERS:

- value

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.LedRed(1);   // a = True   Turn on red led.

var b = Vps.LedRed(0);   // a = True   Turn off red led.

6.13 Relay

FUNCTION: Relay (value);

FUNCTION ID: 17

DESCRIPTION:

Switches on relay and turns on led at the same time with the given time value. Stays on during time value then off.

PARAMETERS:

- value

RETURN VALUE:

Boolean

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.Relay(1000);   // a = True

6.14 Buzzer

FUNCTION: Buzzer (value1, value2);

FUNCTION ID: 20

DESCRIPTION:

Ring buzzer notified by the value1 during value2

Value1: Frequency

Value2 : Duration

PARAMETERS:

- value1 : value
- value2: value

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Values are must be integer. If not first use Float.Int to convert integer data type.

EXAMPLE:

var a = Vps.Buzzer(600,1000);   // a = “” Empty

6.15 TimerSet

FUNCTION: TimerSet (value1, value2);

FUNCTION ID: 25

DESCRIPTION:

Value1 is number of timer, value2 is duration of timer which is assigned to value1

Duration of timer is milliseconds.

PARAMETERS:

- value1 : value   (between [1-10])
- value2: value   (milliseconds)

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Values are must be integer. If not first use Float.Int to convert integer data type.

EXAMPLE:

var a = Vps.TimerSet(1,1000);   // a = “” Empty

6.16 TimerGet

FUNCTION: TimerGet (value);

FUNCTION ID: 26

DESCRIPTION:

Get timer result that is already set TimerSet Library function.
TimerGet function returns remain seconds of the specified by TimerSet function
When timer is over returns zero.

PARAMETERS:

- Value: value

RETURN VALUE:

Integer (returns how many milliseconds left)

EXCEPTIONS:

- value must be integer, if value can not be convert to integer type returns invalid.

EXAMPLE:

var a = Vps.TimerGet(1);   // a = 20

6.17 TimerStop

FUNCTION: TimerStop (value);

FUNCTION ID: 27

DESCRIPTION:

Stop timer, which is specified by value parameter, that is already set TimerSet Library function.

PARAMETERS:

- Value: value
- 1-10 number of timer
- if value equals 0, stop all timers, which are already set by TimerSet library function.

RETURN VALUE:

If succeed returns empty string

EXCEPTIONS:

- value must be integer, if value can not be convert to integer type returns invalid.

EXAMPLE:

```
var a = Vps.TimerStop(1);     // a = “”
var b = Vps.TimerStop(0);     // a = “”  // stop all timer
```

6.18 Clock

FUNCTION: clock ();

FUNCTION ID: 28

DESCRIPTION:

Get real time clock. Before use this function, real time must be set from NIJA compiler Platform. For to this: select **run** from menu and select **Adjust Date/Time**.

Time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 2000, and stores that value in the return value of function

PARAMETERS:

-

RETURN VALUE:

If succeed returns time gives the current time, in seconds, elapsed since 00:00:00 GMT, January 1, 2000, and stores that value in the return value of function

EXCEPTIONS:

- Returns invalid if real time is not set.

EXAMPLE:

```
var a = Vps.Clock();     // a = 12354546
```

6.19 IsoCard

FUNCTION: IsoCard (void);

FUNCTION ID: 30

DESCRIPTION:

Checks if there is card inserted in card socket or not.

PARAMETERS:

-

RETURN VALUE:

If there is card in the card socket return 1 else return 0;

EXCEPTIONS:

-

EXAMPLE:

```
var a = Vps.IsoCard();      // a = 1 or 0
```

6.20 IsoResult

FUNCTION: IsoResult (void);

FUNCTION ID: 31

DESCRIPTION:

Checks Iso card standart of the inserted card.

Returns ISO result code .

PARAMETERS:

-

RETURN VALUE:

Integer

EXCEPTIONS:

-

EXAMPLE:

```
var a = Vps.IsoResult();      // a = 0x9000
```

6.21 IsoAtr

FUNCTION: IsoAtr (value);

FUNCTION ID: 35

DESCRIPTION:

Returns ATR(answer to reset).

If value= 0: Get request from inserted card

If value= 1: Get request from sam 1

If value= 2: Get request from sam 2

Otherwise returns 0.

PARAMETERS:

- Integer

RETURN VALUE:

- String

EXCEPTIONS:

- Returns invalid, if value is not integer data type. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

EXAMPLE:

var a = Vps.IsoAtr(0); // a = returns ATR of inserted card.

6.22 IsoIdt

FUNCTION: IsoIdt (value,string);

FUNCTION ID: 36

DESCRIPTION:

Reads card by the given command, which is given in string type. Value determines which card is going to be read.

If value= 0: Reads from inserted card

If value= 1:Reads from sam 1

If value= 2: Reads from sam 2

Otherwise returns 0.

String indicates that which address is read from card.

PARAMETERS:

- value = Value
- string = String

RETURN VALUE:

- String

EXCEPTIONS:

- Returns invalid, if value is not integer data type and string is not string data type. The legal integer and string syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

EXAMPLE:

var a = Vps.IsoIdt(0,command);   // a = “reading data from card”

6.23 IsoOdt

FUNCTION: IsoOdt (value,string1, string2);

FUNCTION ID: 37

DESCRIPTION:

Selects an address, where will be read in the card.

If value= 0: Selects inserted card

If value= 1: Selects sam 1

If value= 2: Selects sam 2

Otherwise returns 0.

String1 is command of selecting. String2 indicates that which address is selected from card.

PARAMETERS:

- value = Value
- string1 = String
- string2 = String

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Returns invalid, if value is not integer data type, string1 and string2 are not string data type. The legal integer and string syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

EXAMPLE:

var a = Vps.IsoOdt(0,command,data);   // a = “ ” Empty string

6.24 IsoOff

FUNCTION: IsoOff (value);

FUNCTION ID: 38

DESCRIPTION:

Close card reader to ATR

If value= 0: Close card’s ATR

If value= 1: Close SAM’s ATR

If value= 2: Close SAM’s ATR

Otherwise returns 0.

PARAMETERS:

- Integer

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Returns invalid, if value is not integer data type. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

EXAMPLE:

var a = Vps.IsoOff(0);   // a = “” Empty

6.25 SendQuery

FUNCTION: SendQuery(string1, string2);

FUNCTION ID: 50

DESCRIPTION:

Sends string2 to the specified node by string1. String1 is path of the network that where you send your message on the network. String2 is message that you will send to network.

PARAMETERS:

- string1: string
- string2: string

RETURN VALUE:

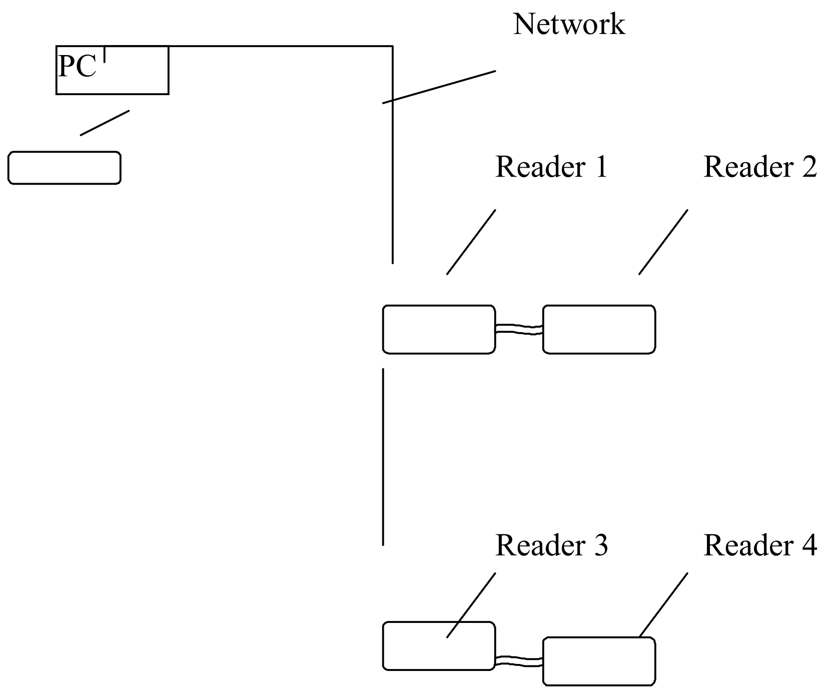
- Empty string

EXCEPTIONS:

- Returns invalid, if value is not string data type. The legal string syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

EXAMPLE:

var a = Vps.SendQuery("\xFE\x01\xFE","message");   // a = "" Empty



FE: Send data to serial port

01,02……0n : unique id of reader

If your source code run on reader 4 and then you will send "message" to Reader 2 as shown above example.

If you download your source code from your PC to Reader 2 You should write following

```
   a = Vps.SendQuery("\x01\xFE","message");      // a = "" Empty
```

If you send a query from Reader 4 to your PC You should write following

```
   a = Vps.SendQuery("\xFE\x01\FE","message");      // a = "" Empty
```

If you send a query from reader 1 to reader 3 You should write following

```
   a = Vps.SendQuery("\x03","message");      // a = "" Empty
```

When you send query using Vps.SendQuery library function, you can get your message from where you send via Vps.PollQuery library function.

6.26 PollAnswer

FUNCTION: PollAnswer(void);

FUNCTION ID: 51

DESCRIPTION:

Get message that will be sent. When a message is sent with Vps.SendAnswer library function, the message is taken by the PollAnswer function.

PARAMETERS:

-

RETURN VALUE:

- The message that will be sent

EXCEPTIONS:

- if there is no message return empty string

EXAMPLE:

var a = Vps.PollAnswer();   // a = message

6.27 PollQuery

FUNCTION: PollQuery(void);

FUNCTION ID: 52

DESCRIPTION:

Get query that will be sent. When a query is sent with Vps.SendQuery library function, the query is taken by the PollQuery function.

PARAMETERS:

-

RETURN VALUE:

- The query that will be sent

EXCEPTIONS:

- if there is no query return empty string

EXAMPLE:

```
var a = Vps.PollQuery();     // a = query
```

6.28 SendAnswer

FUNCTION: SendAnswer(string1);

FUNCTION ID: 53

DESCRIPTION:

When a query is taken by PollQuery library function. Answer of the query is replied by SendAnswer library function

PARAMETERS:

- string1: string

RETURN VALUE:

- Empty string

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.SendAnswer("message"); // a = "" Empty string

6.29 FileFree

FUNCTION: FileFree(void);

FUNCTION ID: 60

DESCRIPTION:

Returns useable total bytes. If return value is less than zero stop execution.

PARAMETERS:

-

RETURN VALUE:

- Integer

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.FileFree(); // a = Total empty bytes

6.30 FileSize

FUNCTION: FileSize(value);

FUNCTION ID: 61

DESCRIPTION:

Files are numerated from 0 to 29. Returns size of the file, which is given with parameter in the function.

Value parameter must be integer. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

PARAMETERS:

- value = Value

RETURN VALUE:

- Integer

EXCEPTIONS:

-

EXAMPLE:

var a = Vps.FileSize(0);   // a = size of file

6.31 FileErase

FUNCTION: FileErase(value);

FUNCTION ID: 62

DESCRIPTION:

Clears contents of the file, which is given by parameter.

Value parameter must be integer. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

If value less than 0 or greater than 29, Stop execution.

PARAMETERS:

- value = Value

RETURN VALUE:

- Emty String

EXCEPTIONS:

- if value is not integer data type return invalid.

EXAMPLE:

var a = Vps.FileErase(0);   // a = “” Empty

6.32 FileMove

FUNCTION: FileMove(value1, value2);

FUNCTION ID: 63

DESCRIPTION:

Moves first file contents, which is given with first parameter to second file which is given with second parameter. Clear contents of the first file. Value1 and value2 parameters must be integer. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

If value1 and value2 less than 0 or greater than 29, Stop execution.

PARAMETERS:

- value = Value

RETURN VALUE:

- Emty String

EXCEPTIONS:

- if value1 and value2 are not integer data type return invalid.

EXAMPLE:

var a = Vps.FileMove(0,1);   // a = “” Empty

6.33 FileFlush

FUNCTION: FileFlush(value1,value2);

FUNCTION ID: 64

DESCRIPTION:

Truncates file as byte as given value with the second parameter. Value1 indicates which file truncated.

Value1 and value2 must be integer. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

If value1 is less than zero or greater than 29, Stop execution.

If value2 is less than zero, Stop execution.

PARAMETERS:

- value1 = Value
- value2 = Value

RETURN VALUE:

- Emty String

EXCEPTIONS:

- if value1 and value2 are not integer data type return invalid.

EXAMPLE:

var a = Vps.FileFlush(0,1);   // a = “” Empty

6.34 FileRead

FUNCTION: FileRead(value1,value2, value3);

FUNCTION ID: 65

DESCRIPTION:

Value1 is file number.

Value2 is offset of the file.

Value3 denotes how many bytes will be read from the file.

Reads value3 bytes from given offset value2 of the file that is specified by value1.

Value1, value2, and value3 parameters must be integer. The legal integer syntax is specified by the NIJAScript (see [NIJAScript Specifications]).

If value1 and less than 0 or greater than 29, Stop execution.

If value2 less than zero, or greater than size of reading file, Stop execution.

If value3 less than zero, Stop execution.

If value2+value3 greater than size of reading file, Stop execution.

PARAMETERS:

- value1 = Value
- value2 = Value
- value3 = Value

RETURN VALUE:

- String

EXCEPTIONS:

- if value1, value2 and value3 are not integer data type return invalid.

EXAMPLE:

var a = Vps.FileRead(0,0,10); // a = ten bytes data from file

6.35 FileWrite

FUNCTION: FileWrite(value1,value2, string);

FUNCTION ID: 66

DESCRIPTION:

Value1 is file number

Value2 is offset of the file

String will be written to the indicated file.

Writes string from given offset value2 to the file.

Value1, value2 must be integer and string is must be string type. The legal integer and string syntax are specified by the NIJAScript (see [NIJAScript Specifications]).

If value1 less than 0 or greater than 29, Stop execution.

If value2 less than zero, or greater than size of reading file, Stop execution.

If length of the string is greater than 64K, Stop execution.

PARAMETERS:

- value1 = Value
- value2 = Value
- string = String

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Return invalid. If value1 and value2 are not integer data type or string is string data type.

EXAMPLE:

var a = Vps.FileWrite(0,0,"Protekila Ltd-Istanbul"); // a = "" Empty

6.36 FileAppend

FUNCTION: FileAppend(value, string);

FUNCTION ID: 67

DESCRIPTION:

Value is file number.

String will be appended to the indicated file.

Value must be integer and string is must be string data type. The legal integer and string syntax are specified by the NIJAScript (see [NIJAScript Specifications]).

If value1 and less than 0 or greater than 29, stop execution.

If length of the string is greater than 64K, stop execution.

PARAMETERS:

- value = Value
- string = String

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Return invalid. If value is not integer data type or string is string data type.

EXAMPLE:

var a = Vps.FileAppend(0,"Protekila Ltd-Istanbul"); // a = " " Empty

6.37 FileTest

FUNCTION: FileTest(void);

FUNCTION ID: 68

DESCRIPTION:

This function is used for file test to reinitialize FAT table. When an error is occurred on file system FAT system is fetched to memory by this function.

PARAMETERS:

RETURN VALUE:

- Empty string

EXCEPTIONS:

- Return invalid. If the function is unsuccessful.

EXAMPLE:

var a = Vps.FileAppend(0,"Protekila Ltd-Istanbul"); // a = " " Empty

6.38 ParamRead

FUNCTION: ParamRead(value1, value2,value3);

FUNCTION ID: 72

DESCRIPTION:

Read parameter file from flash memory. On successful completion ParamRead returns a parameter value which is read specified by the data pointer value is value2. In the event of error it returns third parameter as a default value.

PARAMETERS:

- value1 = value (file number)
- value2 = value (parameter pointer)
- value3 = any type

RETURN VALUE:

- return read value.

EXCEPTIONS:

- Return invalid. If value1 and value2 are not integer data type.

EXAMPLE:

```
var a = Vps.ParamRead(1,4,0);    // a= read value from 1.parameter file
                                                                  (on successful)
                                 // a= 0 (on error)
```

6.39 ParamWrite

FUNCTION: ParamWrite(value1, value2, value3);

FUNCTION ID: 73

DESCRIPTION:

Write to parameter file in the flash memory. On successful completion ParamWrite returns empty string and writes value3 to the parameter file. In the event of error it returns invalid.

PARAMETERS:

- value1 = value
- value2 = value
- value3 = any type

RETURN VALUE:

- Empty string.

EXCEPTIONS:

- Return invalid. If value1 and value2 are not integer data type.

EXAMPLE:

var a = Vps.ParamWrite(1,1,"data");   // a= Write value (on successful)

6.40 MarsUnit

FUNCTION: MarsUnit(value);

FUNCTION ID: 80

DESCRIPTION:

If value greater than 0, Open port and initialize communication with vending machine. If value less than 0 then close communication.

PARAMETERS:

- value = value

RETURN VALUE:

- Empty string.

EXCEPTIONS:

- Return invalid. If value is not integer data type.

EXAMPLE:

var a = Vps.MarsUnit(1);   // a= "" // initialize port

var a = Vps.MarsUnit(0);   // a= "" // close port

6.41 MarsValue

FUNCTION: MarsUnit(value1, value2, value3);

FUNCTION ID: 81

DESCRIPTION:

PARAMETERS:

-

RETURN VALUE:

-

EXCEPTIONS:

- **EXAMPLE:**

6.42 MarsVend

FUNCTION: MarsVend();

FUNCTION ID: 82

DESCRIPTION:

Serve the desired product.

PARAMETERS:

-

RETURN VALUE:

- Empty string.

EXCEPTIONS:

- Return invalid. If any mechanical problem

EXAMPLE:

var a = Vps.MarsVend(); // a= ""

6.43 MarsRequest

FUNCTION: MarsRequest();

FUNCTION ID: 83

DESCRIPTION:

When pushed any product button, get request of customer

PARAMETERS:

-

RETURN VALUE:

- Returns selected number of product.

EXCEPTIONS:

- Return invalid. If any mechanical problem

EXAMPLE:

var a = Vps.MarsRequest();   // a= product number

6.44 MarsState

FUNCTION: MarsState();

FUNCTION ID: 84

DESCRIPTION:

Get status of the vending machine

PARAMETERS:

-

RETURN VALUE:

- Returns status of the vending machine.

EXCEPTIONS:

- Return invalid. If any mechanical problem

EXAMPLE:

var a = Vps.MarsState();   // a= state of machine

6.45 CoinUnit

FUNCTION: CoinUnit(value);

FUNCTION ID: 85

DESCRIPTION:

If value greater than 0, Enabled coin mechanism and get ready to read coin money, when it is thrown to coin mechanism. If value is less than 0, then disabled coin mechanism.

PARAMETERS:

- value = value

RETURN VALUE:

- Empty string.

EXCEPTIONS:

- Return invalid. If value is not integer data type.

EXAMPLE:

var a = Vps.CoinUnit(1);   // a= “” // Enabled coin

var a = Vps.CoinUnit(0);   // a= “” // Disabled coin

6.46 CoinRead

FUNCTION: CoinRead();

FUNCTION ID: 86

DESCRIPTION:

When any coin money is thrown to coin mechanism, reads coin.

PARAMETERS:

-

RETURN VALUE:

- Returns an integer value to dedicate that coin money.

EXCEPTIONS:

- Return invalid. If any mechanical problem

EXAMPLE:

var a = Vps.CoinRead();   // a= coin number

Appendix A. Library Summary

The libraries and their library identifiers:

Library name	Library ID	Page
Lang	0	
Float	1	
String	2	
Protek	3	
Vps	133	

Lang library Function ID

Abs	0
Min	1
Max	2

toInt	3
toFloat	4
isInt	5
isFloat	6
maxInt	7
minInt	8
Random	12
seed	13

Float library Function ID

Int	0
Floor	1
Ceil	2
Pow	3
Round	4
Sqrt	5
maxFloat	6
minFloat	7

String library Function ID

Length	0
isEmpty	1
charAt	2
subString	3
find	4
replace	5
elements	6
elementAt	7
removeAt	8
replaceAt	9
insertAt	10
squeeze	11
Trim	12
compare	13
toString	14

Format	15

Protek library Function ID

toAscii(v)	0
toChar(v)	1
StrToHex(v)	2
HexToStr(v)	3
IntToHex(v,c)	4
HexToInt(v)	5
subShort(v,c)	6
subLong(v,c)	7
swapInt(v)	8
swapLong(v)	9

VPS library Function ID

Clear()	0
Print(x,y,s)	1
Cursor(x,y,b)	2
CurHide()	3
Backlight(v)	4
Contrast(v)	5
UniqeId()	6
Keyboard(e)	10
KeyRead()	11
Remote(e)	12
RemRead()	13
LedGreen(e)	15
LedRed(e)	16
Relay(t)	17
IsoResult()	31