nsL Assembler Language Reference

Introduction

nsL is a high-level language for NSIS (http://nsis.sourceforge.net). The nsL assembler takes nsL code and translates it into NSIS script which can then be compiled into an NSIS installation wizard. nsL has a uniform syntax that is similar to familiar programming languages such as C and Java. Complex expressions can be written freely while being assembled into basic NSIS instructions. Functions are defined and called much like they are in C and Java with the additional syntax for multiple return values. nsL introduces assemble time scope checking of variables as well as automatic declaration and support for global variables declaration and initialization. nsL also provides more powerful pre-processor directives such as macros which can have multiple inputs and outputs, just like run-time functions.

Source Files

Just like with NSIS, one writes their installation wizard code in a plain text file with a text editor such as Notepad. For nsL, the source code files must have an “nsl” file extension. Right clicking on an nsL source code file in Windows Explorer will show the “Compile nsL Script” option. This option will run the nsL Assembler on the chosen file, which assembles the corresponding NSIS (.nsi) script. The makensisw compiler executable is then automatically run on the assembled NSIS script to build the installation wizard executable.

Syntax

Naming

nsL is **case sensitive** and therefore it is important to use the correct case when naming and calling functions, when referring to variables and when using nsL keywords (which are all in lower case).

Comments

Comments in nsL are C-style; that is they can be single line comments starting with `//` or multi-line comments using `/* my comment */`.

Expressions

Expressions in nsL can consist of mathematical expressions, Boolean expressions, concatenation expressions and function call expressions, and can vary in complexity.

```
$0 = $1 + 5 - $2; // math

$0 = $1 > 9 || $2 == 3; // Boolean

$0 = ($1 = myFunc()) > 9 || $1 == 3; // Boolean + call to myFunc()
```

The last expression above contains an assignment to the variable `$1`. Because nsL evaluates from left-to-right, `$1` will have a value if and when it is compared to 3. For both Boolean expressions, `$0` will have the value of `true` or `false`.

Variables

Variables (registers) in nsL remain the same as in NSIS, with a dollar sign (`$`) denoting a variable name. nsL introduces assemble-time enforced scope checking. Before a variable can be used, it must be assigned to:

```
$myVar = 0;
```

When introducing new variable names in nsL, you do not need to declare them using the NSIS `Var` instruction. It is also perfectly fine to use the built in variables `$0`-`$9` and `$R0`-`$R9` and this is still recommended to reduce memory use at run time. To use variables globally, assign a value to them outside any functions and sections.

Strings

Strings can still use all 3 quote characters: `"`, `'` and `` ` ``. However the `$\` escape sequence is now a single `\` and so it is now required to use `\r`, `\n`, `\t`, `\"` etc. instead. Two `\\` characters will result in a single `\`. Note that currently an incorrect escape sequence will not throw an error. A lone `\` character will simply be removed from the string.

Variables used in strings

Unlike NSIS, it is not possible to place variables directly inside quoted strings. The value of the variable will not be substituted at run time. Instead, one can use the string concatenation operator (`.`) akin to PHP:

```
$myVar = "some string" . $myOtherVar . "some other string";
```

You can also use the nsL `format()` assemble time function to insert a variable into a format string:

```
$myVar = format("some string {0} some other string". $myOtherVar);
```

Boolean values

Boolean values are the string literals `true` and `false` (with no quotes). These are used in Boolean expressions, for passing flags to NSIS instructions and for the return value of Boolean instructions (such as `FileExists()`):

```
$1 = FileExists($INSTDIR."\\some_file.exe");
// $1 is now true or false
```

Operators

nsL includes all the operators of the NSIS `IntOp` instruction plus more.

Arithmatic

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus
|	Binary OR
&	Binary AND
^	Binary XOR
<<	Shift left

>>	Shift right
~	Bitwise negate

Boolean

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical negate
==	Equality (equal to)
!=	Equality (not equal to)
>	Relative (greater than)
<	Relative (less than)
>=	Relative (greater than or equal to)
<=	Relative (less than or equal to)

For unsigned integer comparisons, place a lower case "u" after the operator. For case sensitive string comparisons, place an upper case "S" after the operator. For case **in**sensitive string comparisons, place a lower case "s" after the comparison operator. Note that string equality comparisons can also be replaced with calls to StrCmp and StrCmpS. These are semantically identical:

```
$0 = $R0 ==s "str"; // Is the same as:
$0 = StrCmp($R0, "str");

$0 = $R0 ==S "str"; // Is the same as:
$0 = StrCmpS($R0, "str");
```

Note that the assembler will evaluate all expressions to the fullest extent possible at assemble time. The expression `9 + 9` will be replaced with the value of `18`. Similarly `true && false` will evaluate to `false`.

Assignment

Operator	Description
=	Assignment
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulus and assign
|=	Binary OR and assign
&=	Binary AND and assign
^=	Binary XOR and assign
<<=	Shift left and assign
>>=	Shift right and assign

Concatenation

Operator	Description
.	String concatenation

Ternary

Operator	Description
? :	Ternary operator

For those unfamiliar with the ternary operator, it is an `if` statement that can be used as or within an expression on the right hand side of an assignment. For example:

```
$R1 = $R0 >= 0 ? $R0 : 0;
```

This is equivalent to:

```
if ($R0 >= 0)
  $R1 = $R0;
else
  $R1 = 0;
```

Note that the assembler optimises ternary operators in the exact same way that if statements are optimised. If the Boolean expression on the left evaluates to `false` at assemble-time, then the first operand (after the `?`) will never be assembled. Similarly if the Boolean expression on the left evaluates to `true` at assemble-time, then the second operand (after the `:`) will never be assembled. Therefore the ternary operator can also be used as an in-line version of the `#if` directive.

Code blocks

Blocks of code are surrounded by curly braces. These code blocks start a new variable scope and can be used anywhere inside functions and sections:

```
{
  $0 = 0;
}
DetailPrint($0);
```

This will result in the assemble time error: *Variable $0 may not have been initialised* because `$0` is out of scope when used in the `DetailPrint` call.

Functions

Functions are called using the familiar curved brackets and commas for arguments:

```
myFunc(arg1, arg2, arg3, …);
```

Notice that we end with a semicolon. All statements must end with one, just like in Java and PHP. Therefore the NSIS `MessageBox` instruction when written in nsL will be written as:

```
MessageBox("MB_OK", "Hello world");
```

Functions are declared using the `function` keyword like so:

```
function myFunc($arg1, $arg2)
{
  // do something here
  return ($1, $2, $3);
}
```

A function can return multiple values in its `return` statement which are passed to the callers' variables using the following syntax:

```
($R0, $R1, $R2) = myFunc("blah", "blah");
```

Only functions that return a single value can be called within expressions as nsL operators can only have a single left or right operand. Functions that return multiple values must be called independently using the above syntax.

Functions can be overloaded in nsL, that is, you can have multiple functions of the same name that require a different number of arguments or have a different number of return values:

```
function myFunc($1, $2) { return 1; }

function myFunc($1, $2) { return (1, 2); }

function myFunc($1, $2, $3) { return (1, 2); }
```

Uninstall functions (for the uninstall executable) must have the `uninstall` keyword:

```
uninstall function myFunc() { … }
```

NSIS functions

All NSIS functions have been hard coded into the assembler. A high proportion of them have exactly the same arguments as you would have used before (albeit with the new syntax). Many however that return values or use “switches” (such as `/REBOOTOK`) to change their behaviour have been changed.

Functions that have one or more output variables, such as `GetDlgItem` or `GetDLLVersion`, now use assignment:

```
$R0 = GetDlgItem($HWNDPARENT, 2);

($R1, $R2) = GetDLLVersion($INSTDIR.“\\some_file.dll”);
```

Those functions that use switches, such as `RMDir /REBOOTOK “some_directory”` now use a Boolean argument after the main (non optional) arguments to facilitate the switch being specified (in this case, `/REBOOTOK`. For example, the `RMDir` line above will become:

```
RMDir(“some_directory”, true);
```

Sections

Sections are declared using the `section` keyword like so:

```
section MySection(
  [String : description],
  [Boolean : readOnly],
  [Boolean : optional],
  [Boolean : bold],
  [Integer: instType1],
  [Integer: instType2],
  ...
  [Integer: instTypeN])

{
  // my install code here
}
```

Sections can have multiple arguments, all of which are optional.

- The first, `description`, must be a string and is the section name that shows on the *Components* page (not to be confused with the *Modern UI* section description).
- The second argument, `readOnly`, if `true`, specifies that the section be unselectable (read only). This is the same as inserting `SectionIn(true);` within the section body.
- The third argument, `optional`, if `true`, specifies that the section be unchecked by default. The fourth argument, `bold`, if `true`, makes the section name bold on the *Components* page.
- Further arguments must be of an integer type and specify the `InstTypes` that the section belongs to. Note that the `readOnly`, `optional` or `bold` arguments can be omitted before inserting any `InstType` numbers.
- An example:

```
// read only, bold, insttype 1
section MyInstallSection("Installs my app", true, false, true, 1)
{
  // Install my app here
}

// optional, insttypes 1 2 and 3
section MyOptionalSection("Installs my app", false, true, 1, 2, 3)
{
  // Install optional files
}

// read only
uninstall section MyUninstallSection("Uninstalls my app", true)
{
  // Uninstall my app here
}
```

The `SectionIn` instruction is still available which takes multiple arguments of type integer and Boolean. A Boolean argument of true equates to read only (`RO`) and integer arguments equate to the `InstTypes`.

Section groups

Section groups are declared using the `sectiongroup` keyword and use curly brackets to encapsulate sections in the group:

```
[uninstall] sectiongroup MySectionGroup(
  [String : description],
  [Boolean : expanded],
  [Boolean : bold])
{
  // sections go in here
}
```

Section groups can have up to 3 arguments, all of which are optional. The first, `description`, must be a string and is the section group name that shows on the *Components* page (not to be confused with the *Modern UI* section description). The second argument, `expanded`, if `true`, specifies that the section group be expanded by default. The last argument, `bold`, if `true`, makes the section group name bold on the *Components* page.

Pages

Pages are declared using the `page` keyword like so:

```
[uninstall] page Name(
  [String : preFunction],
  [String : showFunction],
  [String : leaveFunction],
  [Boolean : enableCancel]);
```

Where `Name` can be one of `UninstConfirm`, `License`, `Components`, `Directory` or `InstFiles`, or:

```
[uninstall] page Custom(
  [String : createFunction],
  [String : leaveFunction],
  [String : caption]);
```

This will declare a custom page with the given creation and leave functions and window caption. In both cases, all arguments are optional and can be omitted.

To declare a `PageEx`, simple add a code block with curly braces instead of the semi colon. For example:

```
page Directory("MyPreFunction")
{
  DirVar($SOMEVAR);
}
```

The `uninstall` keyword can be added to declare a page as an uninstall page.

Plug-ins

Plug-in functions are called just like nsL functions. For example:

```
($1, $2, $3) = MyPlugin::MyFunc("arg1", "arg2", 3);
```

For additional return values, `Pop` is available:

```
$4 = Pop();
```

Pre-processor directives

nsL boasts some very powerful pre-processor directives that can be used to control which code is assembled and how. All pre-processor directives start with a `#`.

Defining constants

Evaluation at definition

```
#define MyConstName evaluatable_expression
```

Here the expression is evaluated and the result is assigned to the constant. For example:

```
#define MyOtherConst 60
#define MyConst 9 * 5 + MyOtherConst
```

At this point, `MyConst` will have a value of 105.

Evaluation at substitution

```
  #define MyConstName `late_eval_value`
```

Here the special ` quote character is used around the value.

- The value does not need to be a complete expression; it can be any incomplete portion of nsL code.
- The concatenation operator can be used to combine items to build the value.
- The value will only be evaluated into nsL code where the constant is used.
- The value can contain macro and function calls or constant names that have yet been defined.

For example:

```
  #define MyOtherConst 60
  #define MyConst `MyFunc(45, `.MyOtherConst.`, YetAnotherConst)`

  #define YetAnotherConst 99
  if (MyConst) // MyFunc(45, 60, 99)
    MessageBox("Does this make sense?");
```

Like NSIS, using `#define` to define a constant that has previously been defined will throw an assembler error. To bypass this behaviour, nsL has a `#redefine` directive which redefines a constant to a new value. nsL also has an `#undef` directive to undefined a previously defined constant.

Is a constant defined?

To check if a constant has been defined, use the `defined()` assembler function which takes multiple constant names and returns `true` of `false`.

Macros

Macros in nsL are blocks of code which can be inserted multiple times throughout a script. This allows code, which would otherwise be repeated and thus hard to maintain, to be coded only once in a macro and then inserted repeatedly when needed. Macros are in many ways just like functions, in that they can have multiple parameters, can return multiple values, can be overloaded and can call (insert) themselves. They are therefore very suitable for being "called", just like functions, from within complex expressions. Macros are defined like so:

```
  #macro MacroName(Arg1, Arg2, ... ArgN)
    // nsL Script goes here
    #return SomeValue
  #macroend
```

Unlike the `#define` directive, the `#return` directives' value will always be evaluated when the macro is inserted (at substitution). Therefore you only need to use the special ` quote character if the return value is not a complete expression (for example, `` `3 +` ``).

Using macros you can perform an assemble-time loop using recursion with the help of the `#if` directive:

```
  #macro MacroLoop(From, To, Call, Arg)
  #if (From <= To)
    eval(Call . "(From, Arg)");
    MacroLoop(From + 1, To, Call, Arg);
  #endif
#macroend
```

With the use of the `eval` assembler function, which takes a single string argument and evaluates it, inside `MacroLoop` as shown, we can insert any macro within our loop by passing its name as a parameter:

```
MacroLoop(1, 5, "MyMacro", "My argument");
```

The `MyMacro` macro would look like this:

```
#macro MyMacro(Counter, Arg)
  // My code is in here!
#macroend
```

The `eval` assembler function could be used in `MyMacro` as well, if, for example, the value of the `Arg` parameter required further evaluation.

In addition to the macro parameters, a `Returns` constant can be used which contains the number of return values that the current macro insertion requires. Within `MyMacro` above, `Returns` will have a value of 0 when used in `MacroLoop` because `MyMacro` is not assigning to any registers nor is it being inserted within an expression. `Returns` can be used to `#return` the correct number of values in any possible usage scenarios. It can also be used to generate a list of return values, as shown in the *Macros.nsl* example script (the *ZeroMemory* macro).

Inline NSIS code

NSIS code can be written within nsL code by using the `#nsis` directive:

```
#nsis
  MessageBox MB_OK "Hello world!"
#nsisend
```

When the `#nsis` directive is used within an nsL macro, the macro arguments and return variables will be defined as NSIS constants automatically.

- The parameters will be defined as:
  `${[macro_param_name]}`
- The return registers/variables will be defined as:
  `${ReturnVar[number]}`

For example, the NSIS `ReadINIStr` instruction could be written using the `#nsis` directive within a macro:

```
#macro MyReadINIStr(INIFile, Section, Value)
  #if (Returns != 1)
    #error "MyReadINIStr returns 1 value!"
  #endif
  #nsis
    ReadINIStr ${ReturnVar1} "${INIFile}" "${Section}" "${Value}"
  #nsisend
#macroend

$R0 = MyReadINIStr("file.ini", "Section", "Value");
```

Note that the `ReadINIStr` NSIS instruction already exists built in to the assembler.

If directive

The #if directive allows conditional compilation using a Boolean expression that can be evaluated at assemble time:

```
  #if some_boolean_expression
    // Assemble this
  #elseif some_boolean_expression
    // Assemble this
  #else
    // Otherwise assemble this
  #endif
```

Include directive

Including separate source files can be achieved using the `#include` directive which takes a file name or path (string) as an argument. Unlike NSIS, it is possible for an nsL source file to `#include` itself multiple times until a Java *StackOverflowException* occurs. This provides another form of assemble-time recursion just like that of nsL macros.

Special assembler functions

nsL has a couple of handy assemble-time functions to make life easier for the programmer without relying on any unnecessary NSIS run-time instructions. All nsL assembler functions have names in lower case.

returnvar(n)

When used on the right hand side of an assignment operator (=, += etc.), returns the *n*th variable/register being assigned to, starting from 1. Therefore, `$R0 = returnvar(1)` results in nothing being assembled (a register assigning to itself is ignored) and `$R0 = returnvar(2)` results in an error (but `($R0, $R1) = returnvar(2)` will not).

toint(literal)

This converts the given literal argument to an integer literal. The argument can be:

- A string literal of a decimal or hexadecimal representation of an integer, of which is parsed.
- A Boolean literal, which is translated to 1 for `true` or 0 for `false`.
- An integer literal; whereby no conversion is needed.
- A register/variable (such as `$R0`) or constant (such as `$SYSDIR`) which is converted to their internal index number, starting from 0 (`$0`).

eval(string)

This evaluates the given string literal as nsL code. `eval("1 + 1")` is the same as writing `1 + 1` however one can be aided by the use of string concatenation in order to generate nsL code.

defined(const, ...)

Checks if one or more constant names are defined. The return value is a Boolean `true` if all constants are defined or `false` otherwise.

type(literal)

Returns the type of the given literal argument, which can be:

- String
- Boolean
- Integer
- Register (such as `$R0`)
- Constant (such as `$WINDIR`)
- Nonliteral (the value is not a literal or cannot be evaluated to one at assemble-time)

format(format_string, arg1, arg2, ...)

Allows arguments (*arg1*, *arg2* etc.) to be injected into a string literal (*format_string*) using numeric placeholders in the format of `{0}`, `{1}` etc. (zero based index). For example,

```
$R0 = format(“I like {0} and {1}!”, “chicken”, “pizza”);
```

If a `{` is required as part of the text (and does not denote an argument substitution placeholder) then it can be escaped using another `{` in front of it (so two in total).

length(literal)

This returns the length in characters of the given literal. This is an assemble-time alternative to the `StrLen` NSIS instruction.

nsisconst(name)

Returns the NSIS constant of the given name, i.e. `${name}`.

NSIS instructions

Many of the NSIS instructions have had to be redesigned in the way that they are called. The use of switches, such as `/r` in NSIS, are not used in nsL. Instead these are replaced with Boolean arguments which always precede the main argument rather than the other way around.

A non exhaustive list is currently in functions.txt. The list will be merged with this document at a later date.