NSIS System Plug-in

Table of Contents

Introduction

The System plug-in gives developers the ability to call any exported function from any DLL. For example, you can use it to call GetLogicalDriveStrings to get a list of available drives on the user's computer.

The System plug-in also allows the developer to allocate, free and copy memory; interact with COM objects and perform mathematical operations on 64-bit integers.

Programming knowledge is highly recommended for good understanding of the System plug-in.

Usage Examples From The Wiki

- Get local time
- Register conduits with Palm HotSync
- Get free memory
- Read REG_MULTI_SZ
- Get disk serial number
- *Search all...*

Available Functions

Memory Related Functions

- **Alloc** *SIZE*

  Allocates *SIZE* bytes and returns a memory address on the stack.

  Usage Example

  ```
  System::Alloc 64
  Pop $0
  DetailPrint "64 bytes allocated at $0"
  System::Free $0
  ```

- **StrAlloc** *SIZE*

  Allocates a string buffer for *SIZE* **TCHARs** and returns a memory address on the stack. This is extremely useful if you want to write an NSI script that will work for both ANSI and Unicode NSIS.

  Usage Example

  ```
  System::StrAlloc 64 ; String buffer for 63 characters and \0 termination.
  Pop $0
  DetailPrint "A string buffer for 64 characters allocated at $0"
  System::Free $0
  ```

- **Copy** [/*SIZE*] *DESTINATION SOURCE*

  Copies *SIZE* bytes from *SOURCE* to *DESTINATION*. If *SIZE* is not specified, *SOURCE*'s size will queried using GlobalSize. This means that if you don't allocate *SOURCE* using System::Alloc, System::Call or GlobalAlloc, you must specify *SIZE*. If *DESTINATION* is zero it will be allocated and its address will be pushed on the stack.

  Usage example

  ```
  # allocate a buffer and put 'test string' and an int in it
  System::Call "*(&t1024 'test string', i 5) p .s"
  Pop $0
  # copy to an automatically created buffer
  System::Copy 0 $0
  Pop $1
  # get string and int in $1 buffer
  System::Call "*$1(&t1024 .r2, i .r3)"
  # free buffer
  System::Free $1
  # print result
  DetailPrint $2
  DetailPrint $3
  # copy to our own buffer
  System::Alloc 1028
  Pop $1
  System::Copy $1 $0
  ```

```
# get string and int in $1 buffer
System::Call "*$1(&t1024 .r2, i .r3)"
# free
System::Free $0
System::Free $1
# print result
DetailPrint $2
DetailPrint $3
```

- **Free** *ADDRESS*

  Frees *ADDRESS*.

  Usage Example

  ```
  System::Alloc 64
  Pop $0
  DetailPrint "64 bytes allocated at $0"
  System::Free $0
  ```

- **Store** "*OPERATION* [*OPERATION* [*OPERATION* ...]]"

  Performs stack operations. An operation can be pushing or popping a single register from the NSIS stack or pushing or popping all of the registers ($0-$9 and $R0-$R9) from System's private stack. Operations can be separated by any character.

  Available Operations

  - To push $#, use p#, where # is a digit from 0 to 9.
  - To pop $#, use r#, where # is a digit from 0 to 9.
  - To push $R#, use P#, where # is a digit from 0 to 9.
  - To pop $R#, use R#, where # is a digit from 0 to 9.
  - To push $0-$9 and $R0-$R9 to System's private stack, use s or S.
  - To pop $0-$9 and $R0-$R9 from System's private stack, use l or L.

  Usage Examples

  ```
  StrCpy $0 "test"
  System::Store "p0"
  Pop $1
  DetailPrint "$0 = $1"

  StrCpy $2 "test"
  System::Store "p2 R2"
  DetailPrint "$2 = $R2"

  StrCpy $3 "test"
  System::Store "s"
  StrCpy $3 "another test"
  System::Store "l"
  DetailPrint $3

  System::Store "r4" "test"
  DetailPrint $4
  ```

Calling Functions

- **Call** *PROC* [(*PARAMS*) [*RETURN* [? *OPTIONS*]]]
- **Get** *PROC* [(*PARAMS*) [*RETURN* [? *OPTIONS*]]]

  Call and get both share a common syntax. As the names suggest, Call calls and Get gets. What does it call or get? It depends on *PROC*'s value.

  PARAMS is a list of parameters and what do to with them. You can pass data in the parameters and you can also get data from them. The parameters list is separated by commas. Each parameter is combined of three values: *type*, *source* and *destination*. *Type* can be an integer, a string, etc. *Source*, which is the source of the parameter value, can be a NSIS register ($0, $1, $INSTDIR), the NSIS stack, a concrete value (5, "test", etc.) or nothing (null). *Destination*, which is the destination of the parameter value after the call returns, can be a NSIS register, the NSIS stack or nothing which means no output is required. Either one of *source* or *destination* can also be a dot (`.') if it is not needed.

  RETURN is like a single parameter definition, but *source* is only used when creating callback functions. Normally *source* is a dot.

  OPTIONS is a list of options which control the way System plug-in behaves. Each option can be turned off by prefixing with an exclamation mark. For example: **?!e**.

  PARAMS, *RETURN* and *OPTIONS* can be repeated many times in one Get/Call line. When repeating, a lot can be omitted, and only what you wish to change can be used. *Type*, *source* and/or *destination* can be omitted for each parameter, even the return value. Options can be added or removed. This allows you to define function prototypes and save on some typing. The last two examples show this.

  PROC can also be repeated but it must be prefixed with a hash sign (`#') except if the hash sign is preceded by a double colon (`shell32::#18`) in which case it is interpreted as a function ordinal.

  Possible *PROC* Values and Meanings

Value	Meaning	Example
DLL::FUNC	*FUNC* exported from *DLL*	user32::MessageBox
::*ADDR*	Function located at *ADDR*	see below
`*ADDR`	Structure located at *ADDR*	see below
`*`	New structure	see below

IPTR->IDX	Member indexed *IDX* from interface pointed by *IPTR*	see below
<nothing>	New callback function	see below
PROC	*PROC* returned by Get	see below

Available Parameter Types

Type	Meaning
v	void (generally for return)
p	pointer (and other pointer sized types like handles and HWNDs)
b	int8, byte
h	int16, short
i	int32 (includes char, byte, short and so on when used as a pointer)
l	int64, large integer
m	ANSI text, string. (FYI: 'm' for multibyte string or 'w' flipped over.)
t	text, string (pointer to first character). *Like TCHAR*, it is a Unicode string in Unicode NSIS.*
w	WCHAR text, Unicode string
g	GUID
k	callback
@	Direct register memory access (Buffer is limited to `(NSIS_MAX_STRLEN - 24) * NSIS_CHAR_SIZE` bytes)
&v*N*	*N* bytes padding (structures only)
&i*N*	integer of *N* bytes (structures only)
&l	structure size (structures only)
&t*N*	array of *N* TCHAR text characters (structures only)
&m*N*	array of *N* CHAR ANSI characters (structures only)
&w*N*	array of *N* WCHAR Unicode characters (structures only)
&g*16*	*16* bytes of GUID (structures only)

Additionally, each type (except *b*, *h*, *k* and *@*) can be prefixed with an asterisk to denote a pointer. When using an asterisk, the System plug-in still expects the value of the parameter, rather than the pointer's address. To pass a direct address, use `p' with no asterisk. A usage example is available. Alloc returns addresses and its return value should therefore be used with `p', without an asterisk.

Available Sources and Destinations

Type	Meaning
.	ignored
number	concrete hex, decimal or octal integer value. several integers can be or'ed using the pipe symbol (`|')
'*string*' **"*string*"** **`*string*`**	concrete string value
r0* through *r9	$0 through $9 respectively
r10* through *r19 ***R0* through *R9***	$R0 through $R9 respectively
c	$CMDLINE
d	$INSTDIR
o	$OUTDIR
e	$EXEDIR
a	$LANGUAGE
s	NSIS stack
n	null for source, no output required for destination

Source is required when using the @ type and must be a register. When the call returns the source register already contains the memory address in string form so using destination is usually not necessary.

Callbacks

Callback functions are simply functions which are passed to a function and called back by it. They are frequently used to pass a possibly large set of data item by item. For example, EnumChildWindows uses a callback function. As NSIS functions are not quite regular functions, the System plug-in provides its own mechanism to support callback functions. It allows you to create callback functions and notifies you each time a callback function was called.

Creation of callback functions is done using Get and the callback creation syntax. As you will not call the callbacks yourself, the source of the parameters should be omitted using a dot. When the callback is called, the destination of the parameters will be filled with the values passed on to the callback. The value the callback will return is set by the source of the return "parameter". The destination of the return "parameter" should always be set as that's where System will notify you the callback was called.

```
System::Get "(i .r0, i .r1) iss"
```

To pass a callback to a function, use the k type.

```
System::Get "(i .r0, i .r1) isR0"
Pop $0
System::Call "dll::UseCallback(k r0)"
```

Each time the callback is called, the string callback#, where # is the number of the callback, will be placed in the destination of the return "parameter". The number of the first callback created is 1, the second's is 2, the third's is 3 and so on. As System is single threaded, a callback can only be called while calling another function. For example, EnumChildWindows's callback can only be called when EnumChildWindows is being called. You should therefore check for callback# after each function call that might call your callback.

```
System::Get "(i .r0, i .r1) isR0"
Pop $0
System::Call "dll::UseCallback(k r0)"
StrCmp $R0 "callback1" 0 +2
DetailPrint "UseCallback passed ($0, $1) to the callback"
```

After you've processed the callback call, you should use Call, passing it the value returned by Get - the callback. This tells System to return from the callback. Destination of the return "parameter" must be cleared prior to calling a function, to avoid false detection of a callback call. If you've specified a source for the return "parameter" when the callback was created, you should fill that source with the appropriate return value. Callbacks are not automatically freed, don't forget to free it after you've finished using it.

```
System::Get "(i .r0, i .r1) isR0"
Pop $0
System::Call "dll::UseCallback(k r0)"
loop:
        StrCmp $R0 "callback1" 0 done
        DetailPrint "UseCallback passed ($0, $1) to the callback"
        Push 1 # return value of the callback
        StrCpy $R0 "" # clear $R0 in case there are no more callback calls
        System::Call $0 # tell system to return from the callback
        Goto loop
done:
System::Free $0
```

A complete working example is available in the usage examples section.

Notes

- To find out the index of a member in a COM interface, you need to search for the definition of this COM interface in the header files that come with Visual C/C++ or the Platform SDK. The index is zero based.
- If a function can't be found or the `t` parameter type was used, an `A' or `W' will be appended to its name and it will be looked up again. This is done because a lot of Windows API functions have two versions, one for ANSI strings and one for Unicode strings. The ANSI version of the function is marked with `A' and the Unicode version is marked with `W'. For example: lstrcpyA and lstrcpyW.
- Libraries in the system32 directory can be loaded without a path. All other libraries should be loaded with a quoted full path.

Available Options

Option	Meaning
c	cdecl calling convention (the stack restored by caller). By default stdcall calling convention is used on x86 (the stack restored by callee).
r	Always return (for GET means you should pop result and proc, for CALL means you should pop result (at least)). By default result is returned for errors only (for GET you will pop either error result or right proc, and for CALL you will get either your return or result at defined return place).
n	No redefine. Whenever this proc will be used it will never be redefined either by GET or CALL. This options is never inherited to children.
s	Use general Stack. Whenever the first callback defined the system starts using the temporary stacks for function calls.
e	Call GetLastError() after procedure end and push result on stack.
u	Unload DLL after call (using FreeLibrary, so you'll be able to delete it for example).
2	Experimental v2 syntax

Experimental v2 syntax

- Struct types in uppercase are aligned to their natural alignment. Lowercased types are packed without alignment.
- Callback id based on the allocated callback

Usage Examples

```
System::Call 'user32::MessageBox(p $HWNDPARENT, t "NSIS System Plug-in", t "Test", i 0)'
System::Call '"$SysDir\MyLibrary.dll"::MyFunction(i 42)'

System::Call "kernel32::GetModuleHandle(t 'user32.dll') p .s"
System::Call "kernel32::GetProcAddress(p s, m 'MessageBoxA') p .r0"
System::Call "::$0(p $HWNDPARENT, m 'GetProcAddress test', m 'NSIS System Plug-in', i 0)"

System::Get "user32::MessageBox(p $HWNDPARENT, t 'This is a default text', t 'Default', i 0)"
Pop $0
System::Call "$0"

System::Get "user32::MessageBox(p $HWNDPARENT, t 'This is a default text', \
        t 'Default', i 0x1|0x10)"
Pop $0
System::Call "$0(, 'This is a System::Get test', 'NSIS System Plug-in',)"

System::Call "advapi32::GetUserName(t .r0, *i ${NSIS_MAX_STRLEN} r1) i.r2"
DetailPrint "User name - $0"
DetailPrint "String length - $1"
DetailPrint "Return value - $2"

System::Alloc 4
Pop $0
System::Call "*$0(i 5)" ; Write
System::Call "*$0(i .r1)" ; Read
```

```
System::Free $0
DetailPrint $1

System::Call "*(i 5) p .r0"
System::Call "*$0(i .r1)"
System::Free $0
DetailPrint $1

System::Call '*0(p, &l.r2, &t2)' ; &l. is not part of the struct
DetailPrint "Struct size=$2"

System::Call '*(&l4,i,i,i,i,&t128)p.r1' ; Fills dwOSVersionInfoSize with the struct size as a int32
${If} $1 Z<> 0
        System::Call 'kernel32::GetVersionEx(pr1)i.r0'
        System::Call '*$1(i,i.R1,i.R2,i.R3)'
        System::Free $1
        ${IfThen} $0 <> 0 ${|} DetailPrint "v$R1.$R2.$R3" ${|}
${EndIf}

System::Call "user32::GetClientRect(p $hwndparent, @ r0)"
System::Call "*$0(i,i,i.r1,i.r2)"
DetailPrint ClientRect=$1x$2

# defines
!define CLSCTX_INPROC_SERVER 1
!define CLSID_ActiveDesktop {75048700-EF1F-11D0-9888-006097DEACF9}
!define IID_IActiveDesktop {F490EB00-1240-11D1-9888-006097DEACF9}
# create IActiveDesktop interface
System::Call "ole32::CoCreateInstance( \
        g '${CLSID_ActiveDesktop}', p 0, \
        i ${CLSCTX_INPROC_SERVER}, \
        g '${IID_IActiveDesktop}', *p .r0) i.r1"
StrCmp $1 0 0 end
# call IActiveDesktop->GetWallpaper
System::Call "$0->4(w .r2, i ${NSIS_MAX_STRLEN}, i 0)"
# call IActiveDesktop->Release
System::Call "$0->2()"
# print result
DetailPrint $2
end:

InitPluginsDir
File "/oname=$PLUGINSDIR\MyDLL.dll" MyDLL.dll
System::Call 'KERNEL32::AddDllDirectory(w "$PLUGINSDIR")'
System::Call 'KERNEL32::LoadLibrary(t "$PLUGINSDIR\MyDLL.dll")p.r1'
System::Call 'MyDLL::MyFunc(i 5) ? u'
System::Call 'KERNEL32::FreeLibrary(pr1)'
Delete $PLUGINSDIR\MyDLL.dll

System::Get "(p.r1, p) iss"
Pop $R0
System::Call "user32::EnumChildWindows(p $HWNDPARENT, k R0, p) i.s"
loop:
        Pop $0
        StrCmp $0 "callback1" 0 done
        System::Call "user32::GetWindowText(pr1,t.r2,i${NSIS_MAX_STRLEN})"
        System::Call "user32::GetClassName(pr1,t.r3,i${NSIS_MAX_STRLEN})"
        IntFmt $1 "0x%X" $1
        DetailPrint "$1 - [$3] $2"
        Push 1 # callback's return value
        System::Call "$R0"
        Goto loop
done:
System::Free $R0
DetailPrint "EnumChildWindows returned $0"

System::Get '(m.r1)ir2r0 ?2' ; v2 syntax
Pop $9
System::Call 'kernel32::EnumSystemLocalesA(k r9, i 0)'
loop:
        StrCmp $0 "callback$9" 0 done
        DetailPrint "Locale: $1"
        StrCpy $2 1 ; EnumLocalesProc return value
        System::Call $9 ; return from EnumLocalesProc
        Goto loop
done:
System::Free $9

System::Call '*(&t50 "!")p.r2' ; DecimalSep
System::Call '*(&t50 "`")p.r3' ; ThousandSep
System::Call '*(i 2, i 0, i 3, P r2, P r3, i 1)p.r1 ?2'
System::Call 'kernel32::GetNumberFormat(i 0, i 0, t "1337.666" r4, p r1, t.r5, i ${NSIS_MAX_STRLEN})'
DetailPrint "Custom formated $4: $5"
System::Free $3
System::Free $2
System::Free $1

!define MB "user32::MessageBox(p$HWNDPARENT,t,t'NSIS System Plug-in',i0)"
System::Call "${MB}(,'my message',,)"
System::Call "${MB}(,'another message',,) i.r0"
MessageBox MB_OK "last call returned $0"

System::Call "user32::SendMessage(p $HWNDPARENT, t 'test', t 'test', p 0) p.s ? \
        e (,t'test replacement',,) i.r0 ? !e #user32::MessageBox"
DetailPrint $0
ClearErrors
Pop $0
IfErrors good
MessageBox MB_OK "this message box will never be reached"
good:
```

64-bit Functions

- **Int64Op** *ARG1 OP* [*ARG2*]

  Performs *OP* on *ARG1* and optionally *ARG2* and returns the result on the stack. Both *ARG1* and *ARG2* are 64-bit integers. This means they can range between -2^63 and 2^63 - 1.

  Available Operations

  - Addition -- **+**
  - Subtraction -- **-**
  - Multiplication -- *****
  - Division -- **/**
  - Modulo -- **%**
  - Shift left -- **<<**
  - Arithmetic shift right -- **>>**
  - Logical shift right -- **>>>**
  - Bitwise or -- **|**
  - Bitwise and -- **&**
  - Bitwise xor -- **^**
  - Bitwise not (one argument) -- **~**
  - Logical not (one argument) -- **!**
  - Logical or -- **||**
  - Logical and -- **&&**
  - Less than -- **<**
  - Equals -- **=**
  - Greater than -- **>**

  Usage Examples

```
System::Int64Op 5 + 5
Pop $0
DetailPrint "5 + 5 = $0" # 10

System::Int64Op 526355 * 1565487
Pop $0
DetailPrint "526355 * 1565487 = $0" # 824001909885

System::Int64Op 5498449498849818 / 3
Pop $0
DetailPrint "5498449498849818 / 3 = $0" # 1832816499616606

System::Int64Op 0x89498A198E4566C % 157
Pop $0
DetailPrint "0x89498A198E4566C % 157 = $0" # 118

System::Int64Op 1 << 62
Pop $0
DetailPrint "1 << 62 = $0" # 4611686018427387904

System::Int64Op 0x4000000000000000 >> 62
Pop $0
DetailPrint "0x4000000000000000 >> 62 = $0" # 1

System::Int64Op 0x8000000000000000 >> 1
Pop $0
DetailPrint "0x8000000000000000 >> 1 = $0" # -4611686018427387904 (0xC000000000000000)

System::Int64Op 0x8000000000000000 >>> 1
Pop $0
DetailPrint "0x8000000000000000 >>> 1 = $0" # 4611686018427387904 (0x4000000000000000)

System::Int64Op 0x12345678 & 0xF0F0F0F0
Pop $0
# IntFmt is 32-bit, this is just for the example
IntFmt $0 "0x%X" $0
DetailPrint "0x12345678 & 0xF0F0F0F0 = $0" # 0x10305070

System::Int64Op 1 ^ 0
Pop $0
DetailPrint "1 ^ 0 = $0" # 1

System::Int64Op 1 || 0
Pop $0
DetailPrint "1 || 0 = $0" # 1

System::Int64Op 1 && 0
Pop $0
DetailPrint "1 && 0 = $0" # 0

System::Int64Op 9302157012375 < 570197509190760
Pop $0
DetailPrint "9302157012375 < 570197509190760 = $0" # 1

System::Int64Op 5168 > 89873
Pop $0
DetailPrint "5168 > 89873 = $0" # 0

System::Int64Op 189189 = 189189
Pop $0
DetailPrint "189189 = 189189 = $0" # 1

System::Int64Op 156545668489 ~
Pop $0
DetailPrint "156545668489 ~ = $0" # -156545668490

System::Int64Op 1 !
Pop $0
```

```
DetailPrint "1 ! = $0" # 0
```

FAQ

- **Q:** How can I pass structs to functions?

  A: First of all, you must allocate the struct. This can be done in two ways. You can either use Alloc or Call with the special struct allocation syntax. Next, if you need to pass data in the struct, you must fill it with data. Then you call the function with a pointer to the struct. Finally, if you want to read data from the struct which might have been written by the called function, you must use Call with the struct handling syntax. After all is done, it's important to remember to free the struct.

  #### Allocation

  To allocate the struct using Alloc, you must know the size of the struct in bytes. Therefore, it would normally be easier to use Call. In this case it's easy to see the required size is 16 bytes, but other cases might not be that trivial. In both cases, the struct address will be located on the top of the stack and should be retrieved using Pop.

  ```
  System::Alloc 16
  ```

  ```
  System::Call "*(i, i, i, t)p.s"
  ```

  #### Setting Data

  Setting data can be done using Call. It can be done in the allocation stage, or in another stage using the struct handling syntax.

  ```
  System::Call "*(i 5, i 2, i 513, t 'test')p.s"
  ```

  ```
  # assuming the struct's memory address is kept in $0
  System::Call "*$0(i 5, i 2, i 513, t 'test')"
  ```

  #### Passing to the Function

  As all allocation methods return an address, the type of the passed data should be an integer, an address in memory.

  ```
  # assuming the struct's memory address is kept in $0
  System::Call "dll::func(p r0)"
  ```

  #### Reading Data

  Reading data from the struct can be done using the same syntax as setting it. The only difference is that the destination part of the parameter will be set and the source part will be omitted using a dot.

  ```
  # assuming the struct's memory address is kept in $0
  System::Call "*$0(i .r0, i .r1, i .r2, t .r3)"
  DetailPrint "first int = $0"
  DetailPrint "second int = $1"
  DetailPrint "third int = $2"
  DetailPrint "string = $3"
  ```

  #### Freeing Memory

  Memory is freed using Free.

  ```
  # assuming the struct's memory address is kept in $0
  System::Free $0
  ```

  #### A Complete Example

  ```
  # allocate
  System::Call "*(i,i,p,p,p,p,p,p)p.r1"
  # call
  System::Call "Kernel32::GlobalMemoryStatus(p r1)"
  # get
  System::Call "*$1(i.r2, i.r3, p.r4, p.r5, p.r6, p.r7, p.r8, p.r9)"
  # free
  System::Free $1
  # print
  DetailPrint "Structure size: $2 bytes"
  DetailPrint "Memory load: $3%"
  DetailPrint "Total physical memory: $4 bytes"
  DetailPrint "Free physical memory: $5 bytes"
  DetailPrint "Total page file: $6 bytes"
  DetailPrint "Free page file: $7 bytes"
  DetailPrint "Total virtual: $8 bytes"
  DetailPrint "Free virtual: $9 bytes"
  ```