

Table of Contents

Chapter 1: Introduction to NSIS.....	1
Chapter 2: Tutorial: The Basics.....	3
2.1 Introduction.....	3
2.2 Script Files.....	3
2.3 Scripting structure.....	4
2.3.1 Installer Attributes.....	4
2.3.2 Sections.....	4
2.3.3 Functions.....	5
2.3.4 Working with Scripts.....	5
2.3.5 Compiler Commands.....	6
2.4 Compiler.....	7
2.5 Enhancing NSIS.....	7
Chapter 3: MakeNSIS Usage.....	9
Chapter 4: Scripting Reference.....	11
4.1 Script File Format.....	12
4.2 Variables.....	12
4.2.1 Modifiable Variables Used in Instructions.....	12
4.2.2 Constant Variables Used in Instructions and InstallDir.....	13
4.2.3 Variables Used in Strings.....	14
4.3 Labels.....	15
4.4 Relative Jumps.....	15
4.5 Pages.....	16
4.5.1 Ordering.....	16
4.5.2 Callbacks.....	16
4.5.3 Page.....	17
4.5.4 UninstallPage.....	17
4.6 Sections.....	17
4.6.1 Section Commands.....	18
4.6.2 Uninstall Section.....	19
4.7 Functions.....	19
4.7.1 Function Commands.....	19
4.7.2 Callback Functions.....	20
4.8 Installer Attributes.....	23
4.8.1 General Attributes.....	23
4.8.2 Compiler Flags.....	32
4.9 Instructions.....	33
4.9.1 Basic Instructions.....	33
4.9.2 Registry, INI, File Instructions.....	35
4.9.3 General Purpose Instructions.....	38
4.9.4 Flow Control Instructions.....	41
4.9.5 File Instructions.....	44
4.9.6 Uninstaller Instructions.....	46
4.9.7 Miscellaneous Instructions.....	46
4.9.8 String Manipulation Instructions.....	47
4.9.9 Stack Support.....	47

Table of Contents

Chapter 4: Scripting Reference

<u>4.9.10 Integer Support</u>	48
<u>4.9.11 Reboot Instructions</u>	48
<u>4.9.12 Install Logging Instructions</u>	49
<u>4.9.13 Section Management</u>	49
<u>4.9.14 User Interface Instructions</u>	50
<u>4.9.15 Multiple Languages Instructions</u>	53
<u>4.10 Multiple Languages</u>	54
<u>4.10.1 Language Selection</u>	54
<u>4.10.2 The /LANG Parameter</u>	54
<u>4.10.3 LangDLL Plug-in</u>	55
<u>4.11 Plugin DLLs</u>	55
<u>4.11.1 Using Plugin Commands</u>	55
<u>4.11.2 Disabling Plugin Unloading</u>	56

Chapter 1: Introduction to NSIS

NSIS is a free scriptable win32 installer/uninstaller system that doesn't suck and isn't huge.

NSIS exists largely because of the need to distribute Winamp.

In the beginning, Winamp was distributed as a simple .ZIP file (see Winamp 1.0). Eventually, Winzip was nice enough to give us a license to the Winzip self extractor (see Winamp 2.0)

The self extracting ZIP file became too restrictive, so we started using a custom developed installer. This installer was generated using a combination of "bin2h", batch files, and Visual C++. To add a new file into the distribution, or to have the installer do something new on install, it required writing quite a bit of code. (see Winamp 2.5)

Another thing we needed was a good way to distribute Winamp plug-ins. The solution: PIMP (Plug-In Mini Packager). PIMP was actually based on the custom installer (though was a lot simpler and removed a lot of code), but stored the data and strings that were unique to that installer in a block at the end of the file. It was small and simple, allowing the simple functions of detecting where Winamp was installed and extracting files. The limitations of PIMP were that it could only extract files, and it was designed for small installers (i.e. an 8mb installer would take 8mb of memory).

NSIS, which stands for "Nullsoft SuperPIMP Installation System" or "Nullsoft Scriptable Installation System" or whatever you want, is based on PIMP but is designed to be much more flexible. NSIS creates installers that are capable of installing, uninstalling, setting system settings, extracting files, etc. Pretty much anything. All with the minimum of overhead– NSIS installers don't bother throwing up a big blue gradient, they don't bother decompressing themselves three different times, telling the user to "please wait". They get to the point and get the job done.

In order to make NSIS even more powerful than it already is, we have released it under an open source license (it is actually the zlib/libpng license, which is approved by opensource.org). What does this mean? It means that if you want to add the functionality you need to NSIS, you can. It means if you want to make your own custom version of NSIS (or some product that includes NSIS), and sell it, you can. Or if you just want to distribute your software using NSIS, you sure as hell can.

The result of all of this is an installation system for win32 that lets you compile nice little scripts (which are text files, no wizards to slow you down) into tiny installers. Many features are supported, and the whole thing just works pretty damn well (at least, we think).

Here's a short list of some of NSIS' features.

- SuperPiMP technology (so advanced, so amazing, we won't even tell you what it is).
- Generates self contained, win32 executable installer.
- Uninstall support (installer can automatically generate an uninstaller)
- Optional installer self-verification using a CRC32.
- Compression choices of zlib or bzip2 based compression. The installer can compress everything together, or individually.
- Approximately 20–40k overhead over compressed data size (depending on features enabled, compression algorithm, and so on – the default options are ~34k).
- Ability to display a license agreement, text or RTF
- Ability to detect destination directory from the registry, and let the user override (or not let them)
- Customizable appearance (background, icons, text, checkmarks, images)
- Multiple install configurations (usually Minimal, Typical, Full), and custom configuration

Introduction to NSIS

- Easy to use plug-in system (includes plug-ins for generic dialog construction and user interaction, as well as HTTP downloading)
- Fully multilingual. 28 translations available including German, French, Spanish, Italian, Dutch, Chinese and more...
- Installers can be as large as 2GB (theoretically -- when building on Win9x the limit seems to be around 500MB, however building on NT then installing on Win9x works with larger sizes)
- Optional Silent mode for automated installations
- A primitive preprocessor
- User interface changeable from head to toe
- An optional modern user interface
- A lovely coding experience with elements of PHP and assembly (includes 20 general purpose variables, a stack, real flow control, etc)
- Installers have their own VMs that let you write code that can support:
 - ◆ File extraction (with configurable overwrite parameters)
 - ◆ File/directory copying, renaming, deletion
 - ◆ DLL loading (ActiveX control registration/deregistration, extension DLL calling, etc)
 - ◆ Executable execution (shell execute and wait options)
 - ◆ Shortcut creation
 - ◆ Registry key reading/setting/enumerating/deleting
 - ◆ INI file reading/writing
 - ◆ Generic text file reading/writing
 - ◆ Directory scanning
 - ◆ Powerful string and integer manipulation
 - ◆ Window finding based on class name or title (for is-application-running detection)
 - ◆ Window message sending.
 - ◆ User interaction with MessageBox.
 - ◆ Branching, comparisons, etc.
 - ◆ Error checking.
 - ◆ Reboot support, including delete or rename on reboot
 - ◆ Installer behaviour commands (such as show/hide/wait/etc)
 - ◆ User functions in script
 - ◆ Callback functions that let you customize the way the installer behaves from script.
 - ◆ Macros in script
- Completely free for any use. See [license](#).
- More

Chapter 2: Tutorial: The Basics

- **Chapter 2: Tutorial: The Basics**
 - ♦ Introduction
 - ♦ Script Files
 - ♦ Scripting structure
 - ◇ Installer Attributes
 - ◇ Sections
 - ◇ Functions
 - ◇ Working with Scripts
 - ◇ Compiler Commands
 - ♦ Compiler
 - ♦ Enhancing NSIS

2.1 Introduction

Most software packages you download or buy come with an installer. The installer copies and/or updates files, writes registry keys, writes configuration, creates shortcuts, etc. All of this is done automatically for the user. All the user needs to do is supply some information and the installer will do the rest. The user goes through a wizard, makes the appropriate choices and waits until the installer finishes. After the installer has finished the user is left only with the simple task of starting the program. The user doesn't have to worry about things he might have forgotten because all of the necessary steps were done by the installer.

NSIS is a tool for developers to create such installers. NSIS allows you to create everything from basic installers that just copy files to very complex installers that handle a lot of advanced tasks such as writing registry keys, settings environment variables, downloading the latest files from the internet, customizing the configuration file and more. NSIS is very flexible and its scripting language is easy to learn.

NSIS compiles all of the files and the installation script into one executable file, so your application will be easy to distribute. NSIS adds only about 34KB of code of its own (for the default configuration) to the data. NSIS has the smallest overhead available and still have a lot of options thanks to its powerful scripting language and support of external plug-ins.

2.2 Script Files

To create a NSIS installer, you first have to write a NSIS script. A NSIS script is just a regular text file with a special syntax. You can edit scripts with every text editor. It's recommended you use a text editor that shows line numbers because NSIS uses line numbers to indicate where errors lay, and to warn you about where errors might lay. An editor that supports syntax highlighting is also recommended. You can download editors made especially for NSIS and files for syntax highlighting at the [NSIS Archive](#).

In a NSIS script every line is treated as a command. If your command is too long for one line you can use a back-slash – '\' – at the end of the line. The compiler will treat the new line as an addition to the previous line and will not expect a new command. For example:

```
MessageBox MB_OK|MB_ICONINFORMATION \  
"This is a sample that shows how to use line breaks for larger commands in NSIS scripts"
```

If you want to use a double-quote in a string you can either use `\$"` to escape the quote or quote the string with a different type of quote such as ``` or `'`.

For more details about the script format, see [section 4.1](#).

The default extension for a script file is `.nsi`. Header files have the `.nsh` extension. Header files can help you arrange your script by dividing it to more than one block of code, you can also put functions or macros in header files and include the header files in multiple installers. This makes updating easier and it also makes your scripts easier to read. To include a header file in your script use `!include`. Header files that reside in the Include directory under your NSIS directory can be included just by their name. For example:

```
!include Sections.nsh
```

2.3 Scripting structure

A NSIS script can contain Installer Attributes and Sections/Functions. You can also use Compiler Commands for compile-time operations. The minimum is `OutFile`, which tells NSIS where to write the installer, and one section.

2.3.1 Installer Attributes

Installer Attributes determine the behavior and the look and feel of your installer. With these attributes you can define what pages are shown in which order, texts that will be shown during the installation, the number of installation types etc. Most of these commands can only be set and are not changeable during runtime.

The most basic attributes are `Name`, `InstallDir` and `DirText`.

For more information about installer attributes, have a look at [section 4.8](#).

2.3.2 Sections

In a common installer there are several things the user can install. For example in the NSIS distribution installer you can choose to install the source code, additional plug-ins, examples and more. Each of these components has its own piece of code. If the user selects to install this component, then the installer will execute that code. In the script, that code is in sections. Each visible section is a component for the user to choose from. We will not discuss invisible sections in this tutorial. It is possible to build your installer with only one section, but if you want to use the components page and let the user choose what to install you'll have to use more than one section.

The instructions that can be used in sections are very different from the installer attributes instructions, they are executed at runtime on the user's computer. Those instructions can extract files, read from and write to the registry, INI files or normal files, create directories, create shortcuts and a lot more. You can find out about those instructions in [section 4.9](#).

The most basic instructions are `SetOutPath` which tells the installer where to extract files and `File` which extracts files.

Example:

```
Section "My Program"  
  SetOutPath $INSTDIR
```

```
File "My Program.exe"
File "Readme.txt"
SectionEnd
```

For more information about sections see [section 4.6](#).

2.3.3 Functions

Functions, just like sections, can contain code. The difference between sections and functions is the way they are called. There are two types of functions, user functions and callback functions.

User functions are called by the user from within sections or other functions using the [Call](#) instruction. User functions will not execute unless you call them. After the code of the function will be executed the installer will continue executing the instructions that came after the Call instruction, unless you have aborted the installation inside the function. User functions are very useful if you have a set of instructions that need to be executed at several locations in the installers. If you put the code into a function you can save the copying time and you can maintain the code more easily.

Callback functions are called by the installer upon certain defined events such as when the installer starts. Callbacks are optional. If for example you want to welcome the user to your installer you will define a function called `.onInit`. The NSIS compiler will recognize this function as a callback function by the name and will call it when the installer starts.

```
Function .onInit
    MessageBox MB_YESNO "This will install My Program. Do you wish to continue?" IDYES gogogo
    Abort
    gogogo:
FunctionEnd
```

[Abort](#) has a special meaning in callback functions. Each callback function has its own meaning for it, have a look at [section 4.7.2](#) for more information. In the above example Abort tells the installer to stop initializing the installer and quit immediately.

For more information about sections see [section 4.7](#).

2.3.4 Working with Scripts

2.3.4.1 Variables

NSIS offers 20 Variables that can be used in any Section or Function. In addition there is a Stack, which can also be used for temporary storage. To access the stack use the commands [Push](#) and [Pop](#). Push adds a value to the stack, Pop removes one and sets the variable. For example, the value of `$0` is 123.

Now you are going to call the function bla:

```
Function bla

    Push $0 ;123 added to the stack

    ...the function can use $0...
```

Introduction to NSIS

```
Pop $0 ;Remove 123 from the stack and set $0 back to 123
```

```
FunctionEnd
```

After calling the function, the variables contain the same value as before. Note the order when using multiple variables (last-in first-out):

```
Function bla
```

```
    Push $0
```

```
    Push $1
```

```
    ...code...
```

```
    Pop $1
```

```
    Pop $0
```

```
FunctionEnd
```

2.3.4.2 Debugging Scripts

The more you work with NSIS the more complex the scripts will become. This will increase the potential of mistakes, especially when dealing with lots of variables. There are a few possibilities to help you debugging the code. To display the contents of variables you should use [MessageBoxes](#) or [DetailPrint](#). To get a brief overview about all variables you should use the plugin [Dumpstate](#). By default all actions of the Installer are printed out in the Log Window. You can access the log if you right-click in the Log Window and select "Copy Details To Clipboard". There is also a way to write it directly to a file. See [here](#). Very useful if you force to close the installer automatically using [SetAutoClose](#).

2.3.5 Compiler Commands

Compiler commands will be executed on compile time on your computer. They can be used for conditional compilation, to include header files, to execute applications, to change the working directory and more. The most common usage is defines. Defines are compile time constants. You can define your product's version number and use it in your script. For example:

```
!define VERSION "1.0.3"
Name "My Program ${VERSION}"
OutFile "My Program Installer - ${VERSION}.exe"
```

For more information about defines see [section 5.2](#).

Another common use is macros. Macros are used to insert code on compile time, depending on defines and using the values of the defines. An example of a macro is [UpgradeDLL](#), which you can use to upgrade a DLL file. The macro's commands are inserts at compile time. This allows you to write a general code only once and use it a lot of times but with a few changes. For example:

```
!macro MyFunc UN
Function ${UN}MyFunc
    Call ${UN}DoRegStuff
    ReadRegStr $0 HKLM Software\MyProgram key
    DetailPrint $0
FunctionEnd
```

```
!insertmacro MyFunc ""  
!insertmacro MyFunc "un."
```

This macro helps you avoid writing the same code for both the installer and the uninstaller. The two `!insertmacro`s insert two functions, one for the installer called `MyFunc` and one for the uninstaller called `un.MyFunc` and both do exactly the same thing.

For more information see [chapter 5](#).

2.4 Compiler

The second thing you need to do in order to create your installer after you have created your script is to compile your script. `MakeNSIS.exe` is the NSIS compiler. It reads your script, parses it and creates an installer for you.

To compile you have to right-click your `.nsi` file and select `Compile NSI` or `Compile NSIS (with bz2)`. This will cause `MakeNSISw` to launch and call `MakeNSIS` to compile your script. `MakeNSISw` will get the output of `MakeNSIS` and present it to you in a window where you can see it, copy it, test the installer, browse for it and more. If you don't have the `Compile NSI` entry in the context (right-click) menu you have probably unselected `Shell Extensions` in the NSIS installer. You can either use `MakeNSIS.exe` from a command prompt window (DOS) or reinstall NSIS with `Shell Extensions` selected.

The compiler will check your script and give you warnings or an error. If an error occurs (i.e. 2 parameters required but only 1 given) the compiler will abort and a short error message including the line number will be displayed. For non-critical error the compiler will give a warning (i.e. two `DirText` commands in one script). If your script has no errors the compiler will output an installer for you to distribute.

NSIS supports different compression methods `zlib` and `bzip2`. `zlib` is fast and is very efficient in resources consumption. `bzip2` usually gives better results for large installers, but requires a bit more memory and is a little slower. To set the compressor use [SetCompressor](#).

2.5 Enhancing NSIS

NSIS scripts support plug-in calls. Plug-ins are DLL files written in C, C++, Delphi or another programming language. Plug-ins provide a more powerful code base to NSIS as they can contain anything from a code that calculates `1 + 1` to a code that communicates with another computer through `FireWire`.

To call a plug-in function in your script you need to add a line as follows:

```
DLLName::FunctionName "parameter number 1" "parameter number 2" "parameter number 3"
```

Every plug-in's function has its own requirements when it comes to parameters, some will require none, some will accept as many parameters as you want to send. For example:

```
nsExec::ExecToLog "${NSISDIR}\makensis.exe" /CMDHELP'  
InstallOptions::dialog "$PLUGINS\test.ini"  
NSISdl::download http://download.nullsoft.com/winamp/client/winamp281_lite.exe $2
```

Introduction to NSIS

The plug-ins that NSIS knows of are listed at the top of the output of the compiler. NSIS searches for plug-ins in the Plugins directory under your NSIS directory and lists all of their available functions. You can use !addPluginDir to tell NSIS to search in other directories too.

There are several plug-ins that come with the NSIS distribution. InstallOptions is a popular plug-in that allows you to add custom pages to your installer, in combination with the NSIS Page commands (See section 4.5). The Startmenu plug-in provides a page that allows the user to choose a Start Menu folder. There are a lot of plug-ins for different purposes, have a look at the Contrib directory for help files and examples. You can find additional plug-ins at the on-line NSIS Archive.

You can also create a plug-in of your own if you know programming. ExDLL is the basic plug-in example. As all of the plug-ins packed with NSIS and most of the plug-ins in the archive come with source code you can have a look at the Contrib directory and the on-line NSIS Archive for more examples.

You can also customize the dialog resources without modifying or recompiling the source code. Use a resource editor to customize one of the UI files and use the ChangeUI command to use the customized resources.

A popular user interface for NSIS is the Modern User Interface, with an interface like the wizards of recent Windows versions. The Modern UI is not only a customized resource file, it has a lots of new interface elements. It features a white header to describe the current step, a description area on the component page, a Finish page that allows you to run the application or reboot the system and more. The Modern UI language files make it easy to create a multilingual installer, because they contain translations for every label in the installer.

The Modern UI has a macro system that inserts the code to handle all the new UI elements, so you only have to insert a few lines of code to use it. For more information, check the Modern UI Readme and the Modern UI Examples.

Chapter 3: MakeNSIS Usage

NSIS installers are generated by using the 'MakeNSIS' program to compile a NSIS script (.NSI) into an installer executable. The NSIS development kit installer sets up your computer so that you can compile a .nsi file by simply right-clicking on it in explorer, and selecting 'compile'.

If you want to use MakeNSIS on the command line, the syntax of the makensis command is:

```
Makensis [/Vx] [/Olog] [/LICENSE] [/PAUSE] [/NOCONFIG] [/CMDHELP [command]] [/HDRINFO] [/NOCD]
         [/Ddefine[=value] ...] ["/XCommand parameter" ...] [Script.nsi | - [...]]
```

- /LICENSE displays a keen license page.
- The /V switch followed by a number between 0 and 4 will set the verbosity of output accordingly. 0=no output, 1=errors only, 2=warnings and errors, 3=info, warnings, and errors, 4=all output.
- The /O switch followed by a filename tells the compiler to print its log to that file (instead of the screen)
- /PAUSE makes Makensis pause before quitting, which is useful when executing directly from Windows (the auto-installed shell extensions use it).
- /NOCONFIG disables inclusion of [path to makensis.exe]\nsisconf.nsh . Without this parameter, installer defaults are set from nsisconf.nsi. See NSIS Configuration File.
- /CMDHELP prints basic usage information for command (if specified), or all commands (if command is not specified).
- /HDRINFO prints out information on what options Makensis was compiled with.
- /NOCD disabled the current directory change to that of the .nsi file
- Using the /D switch one or more times will add to symbols to the globally defined list (See !define).
- Using the /X switch one or more times will execute the code you specify following it. Example:
"/XAutoCloseWindow false"
- Specifying a dash (-) for the script name will tell Makensis to use the standard input as a source.
- If multiple scripts are specified, they are treated as one concatenated script.

Chapter 4: Scripting Reference

- **Chapter 4: Scripting Reference**
 - ◆ Script File Format
 - ◆ Variables
 - ◇ Modifiable Variables Used in Instructions
 - ◇ Constant Variables Used in Instructions and InstallDir
 - ◇ Variables Used in Strings
 - ◆ Labels
 - ◆ Relative Jumps
 - ◆ Pages
 - ◇ Ordering
 - ◇ Callbacks
 - ◇ Page
 - ◇ UninstPage
 - ◆ Sections
 - ◇ Section Commands
 - ◇ Uninstall Section
 - ◆ Functions
 - ◇ Function Commands
 - ◇ Callback Functions
 - ◆ Installer Attributes
 - ◇ General Attributes
 - ◇ Compiler Flags
 - ◆ Instructions
 - ◇ Basic Instructions
 - ◇ Registry, INI, File Instructions
 - ◇ General Purpose Instructions
 - ◇ Flow Control Instructions
 - ◇ File Instructions
 - ◇ Uninstaller Instructions
 - ◇ Miscellaneous Instructions
 - ◇ String Manipulation Instructions
 - ◇ Stack Support
 - ◇ Integer Support
 - ◇ Reboot Instructions
 - ◇ Install Logging Instructions
 - ◇ Section Management
 - ◇ User Interface Instructions
 - ◇ Multiple Languages Instructions
 - ◆ Multiple Languages
 - ◇ Language Selection
 - ◇ The /LANG Parameter
 - ◇ LangDLL Plug-in
 - ◆ Plugin DLLs
 - ◇ Using Plugin Commands
 - ◇ Disabling Plugin Unloading

4.1 Script File Format

A NSIS Script File (.nsi) is just a text file with a series of commands.

- Lines beginning with ; or # are comments.
- Non-comment lines are in the form of 'command [parameters]'
- To call a plugin, use 'plugin::command [parameters]'. For more info see [section 4.11](#).
- Anything after a ; or # that is not in a parameter (i.e. in quotes or part of another string) is treated as a comment. (i.e. "File myfile ; this is the file" would work)
- For parameters that are treated as numbers, use decimal (the number) or hexadecimal (with 0x prepended to it, i.e. 0x12345AB), or octal (numbers beginning with a 0 and no x).
- To represent strings that have spaces, use quotes.
- Quotes only have the property of containing a parameter if they begin the parameter.
- Quotes can be either single quotes, double quotes, or the backward single quote.
- You can escape quotes using \$\.
- Examples:

```
MessageBox MB_OK "I'll be happy" ; this one puts a ' inside a string
MessageBox MB_OK 'And he said to me "Hi there!"' ; this one puts a " inside a string
MessageBox MB_OK `And he said to me "I'll be fucked!"` ; this one puts both ' and "s inside
MessageBox MB_OK "$\"A quote from a wise man$\" said the wise man" ; this one shows escaping
```

- To extend a command over multiple lines, use a backslash (\) at the end of the line, and the next line will effectively be concatenated the end of it. For example:

```
CreateShortCut "$SMPROGRAMS\NSIS\ZIP2EXE project workspace.lnk" \
"$INSTDIR\source\zip2exe\zip2exe.dsw"

MessageBox MB_YESNO|MB_ICONQUESTION \
    "Remove all files in your NSIS directory? (If you have anything \
    you created that you want to keep, click No)" \
    IDNO NoRemoveLabel
```

- If a file named "nsisconf.nsh" in the same directory as makensis.exe exists, it will be included by default before any scripts (unless the /NOCONFIG command line parameter is used).

4.2 Variables

4.2.1 Modifiable Variables Used in Instructions

Note: All Variables provided by NSIS are case sensitive.

\$INSTDIR

The installation directory (\$INSTDIR is modifiable using [StrCpy](#), [ReadRegStr](#), [ReadINIStr](#), etc. – This could be used, for example, in the .onInit function to do a more advanced detection of install location).

\$OUTDIR

Introduction to NSIS

The current output directory (set implicitly via SetOutPath or explicitly via StrCpy, ReadRegStr, ReadINIStr, etc)

\$0, \$1, \$2, \$3, \$4, \$5, \$6, \$7, \$8, \$9, \$R0, \$R1, \$R2, \$R3, \$R4, \$R5, \$R6, \$R7, \$R8, \$R9

User variables (set via StrCpy, ReadRegStr, ReadINIStr, etc, and use like any other variable). It is recommended (but not required) that you use \$R1–\$R9 as local registers, and \$0–\$9 as global values. Note that any function that lets you specify one of these variables as an output, can use \$INSTDIR or \$OUTDIR as well (but has different implications).

\$CMDLINE

The command line of the installer. The format of the command line can be one of the following:

- "full\path to\installer.exe" PARAMETER PARAMETER PARAMETER
- installer.exe PARAMETER PARAMETER PARAMETER
- For parsing out the PARAMETER portion, see GetParameters on the useful functions appendix. If /D= is specified on the command line (to override the install directory) it won't show up in \$CMDLINE.

\$LANGUAGE

The identifier of the language that is currently used. For example, English is 1033. You can change this variable in .onInit.

4.2.2 Constant Variables Used in Instructions and InstallDir

\$PROGRAMFILES

The program files directory (usually C:\Program Files but detected at runtime).

\$DESKTOP

The windows desktop directory (usually C:\windows\desktop but detected at runtime). The context of this variable (All Users or Current user) depends on the SetShellVarContext setting. The default is the current user.

\$EXEDIR

The location of the installer executable. (technically you can modify this variable, but it is probably not a good idea)

\${NSISDIR}

A symbol that contains the path where NSIS is installed. Detected at compile time. Useful if you want to call resources that are in NSIS directory e.g. Icons, UI's...

\$WINDIR

The windows directory (usually C:\windows or C:\winnt but detected at runtime)

\$SYSDIR

The windows system directory (usually C:\windows\system or C:\winnt\system32 but detected at runtime)

\$TEMP

The system temporary directory (usually C:\windows\temp but detected at runtime)

\$STARTMENU

The start menu folder (useful in adding start menu items using CreateShortCut). The context of this variable (All Users or Current user) depends on the SetShellVarContext setting. The default is the current user.

\$SMPROGRAMS

The start menu programs folder (use this whenever you want \$STARTMENU\Programs). The context of this variable (All Users or Current user) depends on the SetShellVarContext setting. The default is the current user.

\$SMSTARTUP

The start menu programs / startup folder. The context of this variable (All Users or Current user) depends on the SetShellVarContext setting. The default is the current user.

\$QUICKLAUNCH

The quick launch folder for IE4 active desktop and above. If quick launch is not available, simply returns the same as \$TEMP. The context of this variable (All Users or Current user) depends on the SetShellVarContext setting. The default is the current user.

\$HWNDPARENT

The decimal HWND of the parent window.

\$PLUGINS_DIR

The path to a temporary folder created upon the first usage of a plugin or a call to InitPluginsDir. This folder is automatically deleted when the installer exits. This makes this folder the ideal folder to hold INI files for InstallOptions, bitmaps for the splash plugin, or any other file that a plugin needs to work.

\$\$

Use to represent \$.

4.2.3 Variables Used in Strings

\$\r

Use to represent a carriage return (\r).

\$\n

Use to represent a newline (\n).

\${SYMBOL}

Introduction to NSIS

Where SYMBOL is the name of something globally defined, this will be replaced with the value of that symbol. The order of calling Symbols in the script is important.

```
Name "Test Program ${VERSION}"
!define VERSION "V.1.0"
```

This code will set the name of the installer to "Test Program \${VERSION}". The Same happens if the Symbol has not been defined.

```
!define VERSION "V.1.0"
Name "Test Program ${VERSION}"
```

This code sets the name of the installer to "Test Program V.1.0"

4.3 Labels

Labels are the targets of Goto instructions, or of the various branching instructions (such as IfErrors, MessageBox, IfFileExists, and StrCmp). Labels must be within a Section or a Function. Labels are local in scope, meaning they are only accessible from within the Section or Function that they reside in. To declare a label, simply do:

MyLabel:

Labels cannot begin with a -, +, !, \$, or 0-9. When specifying labels for the various instructions that require them, remember that both an empty string ("") and 0 both represent the next instruction (meaning no Goto will occur). Labels beginning with a period (.) are global, meaning you can jump to them from any function or section (though you cannot jump to an uninstall global label from the installer, and vice versa).

4.4 Relative Jumps

Unlike labels, relative jumps are, as the name suggests, relative to the place they are called from. You can use relative jumps wherever you can use labels. Relative jumps are marked by numbers. +1 jumps to the next instruction (the default advancement), +2 will skip one instruction and go to the second instruction from the current instruction, -2 will jump two instructions backward, and +10 will skip 9 instructions, jumping to the tenth instruction from the current instruction.

An instruction is every command that is executed at run-time, when the installer is running. MessageBox, Goto, GetDLLVersion, FileRead, SetShellVarContext are all instructions. AddSize, Section, SubSection, SectionEnd, SetOverwrite (and everything under Compiler Flags), Name, SetFont, LangString, are not instructions because they are executed at compile time.

Examples:

```
Goto +2
  MessageBox MB_OK "You will never ever see this message box"
MessageBox MB_OK "The last message was skipped, this one should be shown"
```

```
Goto +4
MessageBox MB_OK "The following message will be skipped"
Goto +3
MessageBox MB_OK "You will never ever see this message box"
```

```
Goto -3  
MessageBox MB_OK "Done"
```

Note: relative jumps don't work with Exch, File, plug-ins (Plugin::Function), InitPluginsDir, GetFileTimeLocal or GetDLLVersionLocal. Do *not* try to jump over them using relative jumps, you will not get the result you were expecting.

4.5 Pages

Each (non-silent) NSIS installer has a set of pages. Each page can be a NSIS built-in page or a custom page created by a user's function (with InstallOptions for example).

4.5.1 Ordering

The page order is set simply by the order they are in the script. For example:

```
Page license  
Page components  
Page directory  
Page instfiles
```

This code will show the license page, then the components selection page, then the directory selection page and then the install log, just like in old installers.

You can specify the same page type more than once, just make sure you know what you are doing.

Please note that you must still use LicenseText and LicenseData for the license page to show, ComponentText for the components selection page to show and DirText for the directory page to show.

If you don't use any Page command the installer pages order will be just as in older version: license (if LicenseText and LicenseData were specified), components (if ComponentText was specified and there is more than one visible section), directory (if DirText was specified), instfiles.

4.5.2 Callbacks

Each built-in page has three callback functions. The pre-function, the show-creation function and the leave-function. The pre-function is called right before the page is created, the show-function is called right after it is created and before it is shown so you can tweak its user interface with CreateFont, SetBkColor and SendMessage; the leave-function is called right after the user has pressed the next button and before the page is left.

A custom page has only two callback functions, one that creates it which is mandatory, and one leave-function that acts just like the leave-function for built-in pages.

Use Abort from a built-in page's pre-function to skip the page, and from a built-in page's leave-function to stay in the page.

Examples:

Introduction to NSIS

```
Page license skipLicense "" stayInLicense
Page custom customPage "" ": custom page"
Page instfiles

Function skipLicense
    MessageBox MB_YES "Do you want to skip the license page?" IDNO no
    Abort
no:
FunctionEnd

Function stayInLicense
    MessageBox MB_YES "Do you want to stay in the license page?" IDNO no
    Abort
no:
FunctionEnd

Function customPage
    GetTempFileName $R0
    File /oname=$R0 customPage.ini
    InstallOptions::dialog $R0
    Pop $R1
    StrCmp $R1 "cancel" done
    StrCmp $R1 "back" done
    StrCmp $R1 "success" done
    error: MessageBox MB_OK|MB_ICONSTOP "InstallOptions error:$\r$\n$R1"
done:
FunctionEnd
```

4.5.3 Page

((custom [creator_function] [leave_function] [caption])) | ((license|components|directory|instfiles) [

Adds an installer page. See the above sections for more information about built-in versus custom pages and about callback functions. If `define_if_last` is specified, the parameter will be !defined if the page turns out to be the last page before a `instfiles` page. This helps you decide on dynamic scripts (such as the [Modern UI script](#)) what to put in this page's text, "click the install button to start installation" or "click next to continue" for example.

4.5.4 UninstPage

((custom [creator_function] [leave_function] [caption])) | ((uninstConfirm|instfiles) [pre_function] [

Adds an uninstaller page. See the above sections for more information about built-in versus custom pages and about callback functions. See [Page](#) for an explanation about `define_if_last`.

4.6 Sections

Each NSIS installer contains one or more Sections. Each These sections are created, modified, and ended with the following commands.

- Each section contains zero or more instructions.
- Sections are executed in order by the resulting installer, and if `ComponentText` is set, the user will have the option of disabling/enabling each section.

- If a section's name is 'Uninstall', then it is a special Uninstall Section.

4.6.1 Section Commands

4.6.1.1 AddSize

size_kb

Tells the installer that the current section needs an additional "size_kb" kilobytes of disk space. Only valid within a section (will have no effect outside of a section or in a function).

4.6.1.2 Section

[/e] [(**!**)|[-)]section_name [section index output]

Begins and opens a new section. If section_name is empty, omitted, or begins with a -, then it is a required section and the user will not see it, nor have the option of disabling it. If the section name is 'Uninstall', then it is a special Uninstall Section. If section index output is specified, the parameter will be !defined with the section index (that can be used for SectionSetText etc). If the section name begins with a !, the section will be displayed as bold. If /e is present, the sub sections of the section will be expanded by default.

4.6.1.3 SectionEnd

This command closes the current open section.

4.6.1.4 SectionIn

insttype_index [insttype_index] [RO]

This command specifies which install types (see InstType) the current section defaults to the enabled state in. Multiple SectionIn commands can be specified (they are combined). If you specify RO as a parameter, then the section will be read-only, meaning the user won't be able to change its state.

4.6.1.5 SubSection

[/e] Caption [section index output]

This command inserts a subsection. The subsection must be close with SubSectionEnd, and should contain 1 or more Sections. If the subsection name begins with a !, the subsection will be displayed as bold. If /e is present, the sub sections of the sub section will be expanded by default. If section index output is specified, the parameter will be !defined with the section index (that can be used for SectionSetText etc).

4.6.1.6 SubSectionEnd

Closes a subsection opened with SubSection.

4.6.2 Uninstall Section

A special Section named 'Uninstall' must be created in order to generate an uninstaller. This section should remove all files, registry keys, etc that were installed by the installer, from the system. Here is an example of a simple uninstall section:

```
Section "Uninstall"  
    Delete $INSTDIR\Uninst.exe ; delete self (see explanation below why this works)  
    Delete $INSTDIR\myApp.exe  
    RMDir $INSTDIR  
    DeleteRegKey HKLM SOFTWARE\myApp  
SectionEnd
```

The first Delete instruction works (deleting the uninstaller), because the uninstaller is transparently copied to the system temporary directory for the uninstall.

4.7 Functions

Functions are similar to Sections in that they contain zero or more instructions. User functions are not called by the installer directly, instead they are called from Sections using the Call instruction. Callback functions will be called by the installer when a certain event occurs.

Functions must be declared outside of Sections or other Functions.

4.7.1 Function Commands

4.7.1.1 Function

[function_name]

Begins and opens a new function. Function names beginning with "." (e.g. ".Whatever") are generally reserved for callback functions. Function names beginning with "un." are functions that will be generated in the Uninstaller. Hence, normal install Sections and functions cannot call uninstall functions, and the Uninstall Section and uninstall functions cannot call normal functions.

4.7.1.2 FunctionEnd

This command closes the current open function.

4.7.2 Callback Functions

You can create callback functions which have special names, that will be called by the installer at certain points in the install. Below is a list of currently available callbacks:

4.7.2.1 Install Callbacks

4.7.2.1.1 .onGUIInit

This callback will be called just before the first page is loaded and the installer dialog is shown, allowing you to tweak the user interface.

Example:

```
!include "${NSISDIR}\Include\WinMessages.nsh"

Function .onGUIInit
  # 1028 is the id of the branding text control
  GetDlgItem $R0 $HWNDPARENT 1028
  CreateFont $R1 "Tahoma" 10 700
  SendMessage $R0 ${WM_SETFONT} $R1 0
  SetBkColor $R0 0x00FFFFFF
FunctionEnd
```

4.7.2.1.2 .onInit

This callback will be called when the installer is nearly finished initializing. If the '.onInit' function calls Abort, the installer will quit instantly.

Here are two examples of how this might be used:

```
Function .onInit
  MessageBox MB_YESNO "This will install. Continue?" IDYES NoAbort
  Abort ; causes installer to quit.
NoAbort:
FunctionEnd
```

or:

```
Function .onInit
  ReadINIStr $INSTDIR $WINDIR\wincmd.ini Configuration InstallDir
  StrCmp $INSTDIR "" 0 NoAbort
  MessageBox MB_OK "Windows Commander not found. Unable to get install path."
  Abort ; causes installer to quit.
NoAbort:
FunctionEnd
```

4.7.2.1.3 .onInstFailed

This callback is called when the user hits the 'cancel' button after the install has failed (if it could not extract a file, or the install script used the Abort command).

Example:

```
Function .onInstFailed
    MessageBox MB_OK "Better luck next time."
FunctionEnd
```

4.7.2.1.4 .onInstSuccess

This callback is called when the install was successful, right before the install window closes (which may be after the user clicks 'Close' if AutoCloseWindow is set to false).

Example:

```
Function .onInstSuccess
    MessageBox MB_YESNO "Congrats, it worked. View readme?" IDNO NoReadme
    Exec notepad.exe ; view readme or whatever, if you want.
    NoReadme:
FunctionEnd
```

4.7.2.1.5 .onMouseOverSection

This callback is called whenever the mouse position over the sections tree has changed. This allows you to set a description for each section for example. The section id on which the mouse is over currently is stored, temporarily, in \$0.

Example:

```
Function .onMouseOverSection
    FindWindow $R0 "#32770" "" $HWNDPARENT
    GetDlgItem $R0 $R0 1043 ; description item

    StrCmp $0 0 "" +2
        SendMessage $R0 ${WM_SETTEXT} 0 "first section description"

    StrCmp $0 1 "" +2
        SendMessage $R0 ${WM_SETTEXT} 0 "second section description"
FunctionEnd
```

4.7.2.1.6 .onSelChange

Called when the selection changes on the component page. Useful for using with SectionSetFlags and SectionGetFlags.

4.7.2.1.7 .onUserAbort

This callback is called when the user hits the 'cancel' button, and the install hasn't already failed. If this function calls Abort, the install will not be aborted.

Example:

```
Function .onUserAbort
    MessageBox MB_YESNO "Abort install?" IDYES NoCancelAbort
    Abort ; causes installer to not quit.
    NoCancelAbort:
FunctionEnd
```

4.7.2.1.8 .onVerifyInstDir

This callback enables control over whether or not an installation path is valid for your installer. This code will be called every time the user changes the install directory, so it shouldn't do anything crazy with MessageBox or the likes. If this function calls Abort, the installation path in \$INSTDIR is deemed invalid.

Example:

```
Function .onVerifyInstDir
    IfFileExists $INSTDIR\Winamp.exe PathGood
    Abort ; if $INSTDIR is not a winamp directory, don't let us install there
    PathGood:
FunctionEnd
```

4.7.2.2 Uninstall Callbacks

4.7.2.2.1 un.onGUIInit

This callback will be called just before the first page is loaded and the installer dialog is shown, allowing you to tweak the user interface.

Have a look at .onGUIInit for an example.

4.7.2.2.2 un.onInit

This callback will be called when the uninstaller is nearly finished initializing. If the 'un.onInit' function calls Abort, the uninstaller will quit instantly. Note that this function can verify and/or modify \$INSTDIR if necessary.

Here are two examples of how this might be used:

```
Function un.onInit
    MessageBox MB_YESNO "This will uninstall. Continue?" IDYES NoAbort
    Abort ; causes uninstaller to quit.
    NoAbort:
FunctionEnd
```

or:

```
Function un.onInit
  IfFileExists $INSTDIR\myfile.exe found
  MessageBox MB_OK "Uninstall path incorrect"
  Abort
found:
FunctionEnd
```

4.7.2.2.3 un.onUninstFailed

This callback is called when the user hits the 'cancel' button after the uninstall has failed (if it used the Abort command or otherwise failed).

Example:

```
Function un.onUninstFailed
  MessageBox MB_OK "Better luck next time."
FunctionEnd
```

4.7.2.2.4 un.onUninstSuccess

This callback is called when the uninstall was successful, right before the install window closes (which may be after the user clicks 'Close' if AutoCloseWindow is set to false).

Example:

```
Function un.onUninstSuccess
  MessageBox MB_OK "Congrats, it's gone."
FunctionEnd
```

4.7.2.2.5 un.onUserAbort

This callback is called when the user hits the 'cancel' button and the uninstall hasn't already failed. If this function calls Abort, the install will not be aborted.

Example:

```
Function un.onUserAbort
  MessageBox MB_YESNO "Abort uninstall?" IDYES NoCancelAbort
  Abort ; causes uninstaller to not quit.
NoCancelAbort:
FunctionEnd
```

4.8 Installer Attributes

4.8.1 General Attributes

The commands below all adjust attributes of the installer. These attributes control how the installer looks and functions, including which pages are present in the installer, as what text is displayed in each part of each page, how

the installer is named, what icon the installer uses, the default installation directory, what file it writes out, and more. Note that these attributes can be set anywhere in the file except in a Section or Function. With the exception of InstallDir, none of these attributes allow use of Variables other than \$!r and \$!n in their strings.

Defaults are bold

4.8.1.1 AddBrandingImage

(left|right|top|bottom) (width|height) [padding]

Adds a branding image on the top, bottom, left, or right of the installer. Its size will be set according to the width/height specified, the installer width/height and the installer font. The final size will not always be the same as the what you have requested. Have a look at the output of the command for the final size. Because this depends on the installer font, you should use SetFont before AddBrandingImage. The default padding value is 2.

4.8.1.2 AllowRootDirInstall

true|**false**

Controls whether or not installs are enabled to the root directory of a drive, or directly into a network share. Set to 'true' to change the safe behavior, which prevents users from selecting C:\ or \\Server\Share as an install (and later on, uninstall) directory. For additional directory selection page customizability, see [.onVerifyInstDir](#).

4.8.1.3 AutoCloseWindow

true|**false**

Sets whether or not the install window automatically closes when completed. This is overrideable from a section using SetAutoClose.

4.8.1.4 BGGradient

[**off**|(topc botc [textcolor|notext])]

Specifies whether or not to use a gradient background window. If 'off', the installer will not show a background window, if no parameters are specified, the default black to blue gradient is used, and otherwise the top_color or bottom_color are used to make a gradient. Top_color and bottom_color are specified using the form RRGGBB (in hexadecimal, as in HTML, only minus the leading '#', since # can be used for comments). 'textcolor' can be specified as well, or 'notext' can be specified to turn the big background text off.

4.8.1.5 BrandingText

[/LANG=lang_id] /TRIM(LEFT|RIGHT|CENTER) text

Sets the text that is shown (by default it is 'Nullsoft Install System vX.XX') in the bottom of the install window. Setting this to an empty string ("") uses the default; to set the string to blank, use " " (a space). If it doesn't matter to you, leave it the default so that everybody can know why the installer didn't suck. heh. Use /TRIMLEFT,

/TRIMRIGHT or /TRIMCENTER to trim down the size of the control to the size of the string.

4.8.1.6 Caption

[/LANG=lang_id] caption

Sets what the titlebars of the installer will display. By default, it is 'Name Setup', where Name is specified with the Name command. You can, however, override it with 'MyApp Installer' or whatever. If you specify an empty string (""), the default will be used (you can however specify " " to achieve a blank string)

4.8.1.7 ChangeUI

dialog ui_file.exe

Replaces dialog (*IDD_LICENSE*, *IDD_DIR*, *IDD_SELCOM*, *IDD_INST*, *IDD_INSTFILES*, *IDD_UNINST* or *IDD_VERIFY*) by a dialog with the same resource ID in ui_file.exe. You can also specify 'all' as the dialog if you wish to load replace all 7 of the dialogs at once from the same UI file. For some example UIs look at Contrib\UIs under your NSIS directory.

- *IDD_LICENSE* must contain *IDC_EDIT1* (RICHEDIT control).
- *IDD_DIR* must contain *IDC_DIR* (edit box), *IDC_BROWSE* (button) and *IDC_CHECK1* (checkbox).
- *IDD_SELCOM* must contain *IDC_TREE1* (SysTreeView32 control), and *IDC_COMBO1* (combo box).
- *IDD_INST* must contain *IDC_BACK* (button), *IDC_CHILDDIRECT* (static control the size of all other dialogs), *IDC_VERSTR* (static), *IDOK* (button), and *IDCANCEL* (button). If an image control (static with *SS_BITMAP* style) will be found in this dialog it will be used as the default for SetBrandingImage.
- *IDD_INSTFILES* must contain *IDC_LIST1* (SysListView32 control), *IDC_PROGRESS* (msctls_progress32 control), and *IDC_SHOWDETAILS* (button).
- *IDD_UNINST* must contain *IDC_EDIT1* (edit box).
- *IDD_VERIFY* must contain *IDC_STR* (static).

4.8.1.8 CheckBitmap

bitmap.bmp

Specifies the bitmap with the images used for the checks of the component-selection page treeview.

This bitmap should have a size of 96x16 pixels and contain six 16x16 images for the different states (in order: selection mask, not checked, checked, greyed out, unchecked & read-only, checked & read-only). Use magenta as mask color (this area will be transparent).

4.8.1.9 CompletedText

[/LANG=lang_id] Completed text

Replaces the default text ("Completed") that is printed at the end of the install if parameter is specified. Otherwise, the default is used.

4.8.1.10 ComponentText

```
[[/LANG=lang_id] text [subtext] [subtext2]]
```

Specifies a string that is above the component list. This can be something that notifies the user what it is they are actually installing. Note that if no parameter is specified, or if the ComponentText command is omitted, then the component page will not be visible, and all of the sections will be installed. Note: if text is specified and non-empty and you leave subtext or subtext2 empty, the defaults will be used (to set one to blank, use a string like " "). empty strings mean default on subtext and subtext2. Likewise, if you wish to enable the component page, but don't want any text on the top line, set text to " ".

4.8.1.11 CRCCheck

on|off|force

Specifies whether or not the installer will perform a CRC on itself before allowing an install. Note that if the user uses /NCRC on the command line when executing the installer, and you didn't specify 'force', the CRC will not occur, and the user will be allowed to install a (potentially) corrupted installer.

4.8.1.12 DetailsButtonText

```
[/LANG=lang_id] show details text
```

Replaces the default details button text of "Show details", if parameter is specified (otherwise the default is used).

4.8.1.13 DirShow

show|hide

Specifies whether or not the user will see the directory selection page. Note that if 'hide' is specified, the installer will still check the validity of the installation path (using internal logic and .onVerifyInstDir if it is declared). If the path is deemed invalid, the directory page will be shown. To completely disable the Directory page (and install without prompting, even if a valid installation path is not available), specify DirText with no parameter (this might be useful if the installer installs everything into \$SYSDIR or something like that).

4.8.1.14 DirText

```
[[/LANG=lang_id] text [subtext] [browse text]]
```

Specifies a string that is above the directory selection area. If this command is not specified, or no parameter is specified, then the directory page is never visible to the user (even if DirShow show is specified). If subtext is specified and not empty, it overrides the default text above the path entry box ("Select the directory to install MyApp in:"). If browse button text is specified but not empty, it overrides the default browse button text ("Browse...").

4.8.1.15 FileErrorText

`[/LANG=lang_id] file error text`

Replaces the default text that comes up when a file cannot be written to. This string can contain a reference to \$0, which is the filename (\$0 is temporarily changed to this value). Example: "Can not write to file \$r\$\n\$0\$r\$\ngood luck, bitch."

4.8.1.16 Icon

`[path\]icon.ico`

Sets the icon of the installer. Every icon in the icon file will be included in the installer. Note that if you use different icons for installer and uninstaller the file size and structure of the icons has to match otherwise the build of your installer will fail.

4.8.1.17 InstallButtonText

`[/LANG=lang_id] install button text`

If parameter is specified, overrides the default install button text (of "Install") with the specified text.

4.8.1.18 InstallColors

`/windows | foreground background`

Sets the colors to use for the install info screen (the default is 00FF00 000000. Use the form RRGGBB (in hexadecimal, as in HTML, only minus the leading '#', since # can be used for comments). Note that if "/windows" is specified as the only parameter, the default windows colors will be used.

4.8.1.19 InstallDir

`definstdir`

Sets the default installation directory is. See the variables section for variables that can be used to make this string (especially \$PROGRAMFILES). Note that the part of this string following the last \ will be used if the user selects 'browse', and may be appended back on to the string at install time (to disable this, end the directory with a \ (which will require the entire parameter to be enclosed with quotes)). If this doesn't make any sense, play around with the browse button a bit.

4.8.1.20 InstallDirRegKey

`root_key subkey key_name`

This attribute tells the installer to check a string in the registry, and use it for the install dir if that string is valid. If this attribute is present, it will override the InstallDir attribute if the registry key is valid, otherwise it will fall back to the InstallDir default. When querying the registry, this command will automatically remove any quotes. If the string ends

in ".exe", it will automatically remove the filename component of the string (i.e. if the string is "C:\program files\poop\poop.exe", it will know to use "C:\program files\poop"). For more advanced install directory configuration, set \$INSTDIR in .onInit.

4.8.1.21 InstProgressFlags

[flag [...]]

Valid values for flag are "smooth" (smooth the progress bar) or "colored" (color the progress bar with the colors set by InstallColors. Examples: "InstProgressFlags" (default old-school windows look), "InstProgressFlags smooth" (new smooth look), "InstProgressFlags smooth colored" (colored smooth look whee). Note: neither "smooth" or "colored" work with XPStyle on when the installer runs on Windows XP with a modern theme.

4.8.1.22 InstType

install_type_name | /NOCUSTOM | ([/LANG=lang_id] /CUSTOMSTRING=str) | /COMPONENTSONLYONCUSTOM

Adds an install type to the install type list, or disables the custom install type. There can be as many as 8 types, each one specifying the name of the install. The first type is the default (generally 'Typical'). Each type is numbered, starting at 1. See SectionIn for information on how those numbers are used. If the /NOCUSTOM switch is specified, then the "custom" install type is disabled, and the user has to choose one of the pre-defined install types. Alternatively, if the /CUSTOMSTRING switch is specified, the parameter will override the "Custom" install type text. Alternatively, if the /COMPONENTSONLYONCUSTOM flag is specified, the component list will only be shown if the "Custom" install type is selected.

4.8.1.23 LicenseBkColor

color

Sets the background color of the license data. Color is specified using the form RRGGBB (in hexadecimal, as in HTML, only minus the leading '#', since # can be used for comments).

4.8.1.24 LicenseData

[/LANG=lang_id] licdata.(txt|rtf)

Specifies a text file or a RTF file to use for the license that the user can read. Omit this to not have a license displayed. Note that the file must be in the evil DOS text format (\r\n, yeah!)

If you make your license file a RTF file it is recommended you edit it with WordPad and not MS Word. Using WordPad will result in a much smaller file.

4.8.1.25 LicenseForceSelection

[/LANG=lang_id] (checkbox [accept_text] | radiobuttons [accept_text] [decline_text] | off)

Specifies if the displayed license must be accept explicit or not. This can be done either by a checkbox or by radiobuttons. By default the "next button" is disabled and will only be enabled if the checkbox is enabled or the right radio button is selected. If off is specified the "next button" is enabled by default.

4.8.1.26 LicenseText

```
[[/LANG=lang_id] text [button_text]]
```

Specifies a string that is above the license text. Omit this to not have a license displayed. If button_text is specified, it will override the default button text of "I Agree".

4.8.1.27 MiscButtonText

```
[[/LANG=lang_id] back button text [next button text] [cancel button text] [close button text]]
```

Replaces the default text strings for the four buttons (< Back, Next >, Cancel and Close). If parameters are omitted, the defaults are used.

4.8.1.28 Name

```
[/LANG=lang_id] name
```

Sets the name of the installer. The name is usually simply the product name such as 'MyApp' or 'CrapSoft MyApp'.

4.8.1.29 OutFile

```
[path\]install.exe
```

Specifies the output file that the MakeNSIS should write the installer to. This is just the file that MakeNSIS writes, it doesn't affect the contents of the installer.

4.8.1.30 SetFont

```
font_face_name font_size
```

Sets the installer font. Please remember that the font you choose must be present on the user's machine as well. Don't use rare fonts that only you have.

4.8.1.31 ShowInstDetails

```
hide|show|nevershow
```

Sets whether or not the details of the install are shown. Can be 'hide' to hide the details by default, allowing the user to view them, or 'show' to show them by default, or 'nevershow', to prevent the user from ever seeing them. Note that sections can override this using SetDetailsView.

4.8.1.32 ShowUninstallDetails

hide|show|nevershow

Sets whether or not the details of the uninstall are shown. Can be 'hide' to hide the details by default, allowing the user to view them, or 'show' to show them by default, or 'nevershow', to prevent the user from ever seeing them. Note that sections can override this using SetDetailsView.

4.8.1.33 SilentInstall

normal|silent|silentlog

Specifies whether or not the installer should be silent. If it is 'silent' or 'silentlog', all sections that have the SF_SELECTED flag are installed quietly (you can set this flag using see [SectionSetFlags](#)), with no screen output from the installer itself (MessageBoxes are still displayed on error, and the script can still display whatever it wants). Note that if this is set to 'normal' and the user runs the installer with /S on the command line, it will behave as if SilentInstall 'silent' was used. Note: see also [LogSet](#).

4.8.1.34 SilentUninstall

normal|silent

Specifies whether or not the uninstaller should be silent. If it is 'silent' or 'silentlog', the uninstall section will run quietly, with no screen output from the uninstaller itself (MessageBoxes are still displayed on error, and the script can still display whatever it wants). Note that if this is set to 'normal' and the user runs the uninstaller with /S on the command line, it will behave as if SilentUninstall 'silent' was used. Note: see also [LogSet](#).

4.8.1.35 SpaceTexts

[[/LANG=lang_id] req text [avail text]]

If parameters are specified, overrides the space required and space available text ("Space required: " and "Space available: " by default). If 'none' is specified as the required text no space texts will be shown.

4.8.1.36 SubCaption

[[/LANG=lang_id] page_number subcaption]

Overrides the subcaptions for each of the installer pages (0=": License Agreement",1=": Installation Options",2=": Installation Directory", 3=": Installing Files", 4=": Completed"). If you specify an empty string (""), the default will be used (you can however specify " " to achieve a blank string)

4.8.1.37 UninstallButtonText

[/LANG=lang_id] button text

Changes the text of the button that by default says "Uninstall" in the uninstaller. If no parameter is specified, the default text is used. See also [WriteUninstaller](#) (replaces UninstallEXENAME).

4.8.1.38 UninstallCaption

`[/LANG=lang_id] caption`

Sets what the titlebars of the uninstaller will display. By default, it is 'Name Uninstall', where Name is specified with the Name command. You can, however, override it with 'MyApp uninstaller' or whatever. If you specify an empty string (""), the default will be used (you can however specify " " to achieve a blank string)

4.8.1.39 UninstallIcon

`[path\]icon.ico`

Sets the icon of the uninstaller. This icon file *must* have the exact same structure as the installer icon file.

4.8.1.40 UninstallSubCaption

`[/LANG=lang_id] page_number subcaption`

Overrides the subcaptions for each of the uninstaller pages (0=": Confirmation",1=": Uninstalling Files",2=": Completed"). If you specify an empty string (""), the default will be used (you can however specify " " to achieve a blank string)

4.8.1.41 UninstallText

`[/LANG=lang_id] text [subtext]`

Specifies the text on the first page of the uninstaller. If subtext is specified and not empty, it will replace the default secondary text on that page, "Uninstall from:".

4.8.1.42 WindowIcon

`on|off`

Sets whether or not the installer's icon is being displayed.

4.8.1.43 XPStyle

`on|off`

Sets whether or not an XP manifest will be added to the installer. This affects the uninstaller too.

4.8.2 Compiler Flags

The following commands change how the compiler generates code and compresses data. These commands are valid anywhere in the script, and effect every line below where each one is placed (until overridden by another command).

4.8.2.1 SetCompress

auto|force|off

This command sets the compress flag which is used by the installer to determine whether or not data should be compressed. Typically the SetCompress flag will effect the commands after it, and the last SetCompress command in the file also determines whether or not the install info section and uninstall data of the installer is compressed. If compressflag is 'auto', then files are compressed if the compressed size is smaller than the uncompressed size. If compressflag is set to 'force', then the compressed version is always used. If compressflag is 'off' then compression is not used (which can be faster). Note that this option has no effect on bzip2 installers (compression is always used on bzip2 installers).

4.8.2.2 SetCompressor

zlib|bzip2

This command sets the compression algorithm used to compress files/data in the installer. Options are ZLib or BZip2. ZLib (the default) uses deflate compression. This mode uses less memory at runtime and is faster. BZip2 compression is usually better for large installers, but it is slower and uses a lot more memory at runtime.

4.8.2.3 SetDatablockOptimize

on|off

This command tells the compiler whether or not to do datablock optimizations. Datablock optimizations have the compiler check to see if any data being added to the data block is already in the data block, and if so, it is simply referenced as opposed to added (can save a little bit of size). It is highly recommended to leave this option on.

4.8.2.4 SetDateSave

on|off

This command sets the file date/time saving flag which is used by the File command to determine whether or not to save the last write date and time of the file, so that it can be restored on installation. Valid flags are 'on' and 'off'. 'on' is the default.

4.8.2.5 SetOverwrite

on|off|try|ifnewer

This command sets the overwrite flag which is used by the File command to determine whether or not the file should

overwrite any existing files that are present. If `overwriteflag` is 'on', files are overwritten (this is the default). If `overwriteflag` is 'off', files that are already present are not overwritten. If `overwriteflag` is 'try', files are overwritten if possible (meaning that if the file is not able to be written to, it is skipped without any user interaction). If `overwriteflag` is 'ifnewer', then files are only overwritten if the existing file is older than the new file (note that when in 'ifnewer' mode, the destination file's date is set, regardless of what `SetDateSave` is set to).

4.8.2.6 SetPluginUnload

`manual|alwaysoff`

This command sets the unload plugin flag which is by `CallInstDLL` and plugin calls (`dll::func`). Setting this to always off will behave as if you have added the `/NOUNLOAD` to every `CallInstDLL` and plugin call. Setting this to manual will only not unload if you specifically use `/NOUNLOAD`.

4.9 Instructions

4.9.1 Basic Instructions

The instructions that NSIS uses for scripting are sort of a cross between PHP and assembly. There are no real high level language constructs, but the instructions themselves are (for the most part) high level, and you have handy string capability (i.e. you don't have to worry about concatenating strings, etc). You essentially have 25 registers (20 general purpose, 5 special purpose), and a stack.

4.9.1.1 Delete

`[/REBOOTOK] file`

Delete file (which can be a file or wildcard, but should be specified with a full path) from the target system. If `/REBOOTOK` is specified and the file cannot be deleted then the file is deleted when the system reboots -- if the file will be deleted on a reboot, the reboot flag will be set. The error flag is set if files are found and cannot be deleted. The error flag is not set from trying to delete a file that does not exist.

4.9.1.2 Exec

`command`

Execute the specified program and continue immediately. Note that the file specified must exist on the target system, not the compiling system. `$OUTDIR` is used for the working directory. The error flag is set if the process could not be launched. Note, if the command could have spaces, you should put it in quotes to delimit it from parameters. e.g.: `Exec "$INSTDIR\command.exe" parameters'`. If you don't put it in quotes it will *not* work on Windows 9x with or without parameters.

4.9.1.3 ExecShell

action command [parameters] [SW_SHOWNORMAL | SW_SHOWMAXIMIZED | SW_SHOWMINIMIZED]

Execute the specified program using ShellExecute. Note that action is usually "open", "print", etc, but can be an empty string to use the default action. Parameters and the show type are optional. \$OUTDIR is used for the working directory. The error flag is set if the process could not be launched.

4.9.1.4 ExecWait

command [user_var(exit code)]

Execute the specified program and wait for the executed process to quit. See Exec for more information. If no output variable is specified ExecWait sets the error flag if the program executed returns a nonzero error code, or if there is an error. If an output variable is specified, ExecWait sets the variable with the exit code (and only sets the error flag if an error occurs; if an error occurs the contents of the user variable are undefined). Note, if the command could have spaces, you should put it in quotes to delimit it from parameters. e.g.: ExecWait "\$INSTDIR\command.exe" parameters'. If you don't put it in quotes it will *not* work on Windows 9x with or without parameters.

4.9.1.5 File

[/nonfatal] [/a] ([/r] (file|wildcard) [...] | /oname=file.dat infile.dat)

Adds file(s) to be extracted to the current output path (\$OUTDIR).

- Note that the output file name is \$OUTDIR\filename_portion_of_file.
- If the /oname=X switch is used, the output name becomes \$OUTDIR\X. When using the /oname= switch, only one file can be specified, and the file name can contain variables (or a fully qualified path, e.g. \$SYSDIR\whatever.dll).
- Wildcards are supported.
- If the /r switch is used, files and directories are added recursively. If is no trailing wildcard (e.g. File /r C:\whatever\mydir), then the whole tree of mydir will go in \$OUTDIR\mydir. To put it in \$OUTDIR, use File /r C:\whatever\mydir*.*
- If the /a switch is used, the attributes of the file(s) added will be preserved.
- The File command sets the error flag if overwrite mode is set to 'try' and the file could not be overwritten, or if the overwrite mode is set to 'on' and the file could not be overwritten and the user selects ignore.
- If the /nonfatal switch is used, a warning will be issued if no files found instead of an error.

4.9.1.6 Rename

[/REBOOTOK] source_file dest_file

Rename source_file to dest_file. Functions just like the Win32 API MoveFile, which means you can move a file from anywhere on the system to anywhere else, and you can move a directory to somewhere else on the same drive. If /REBOOTOK is specified, and the file cannot be overwritten, then the file is moved when the system reboots -- if the file will be moved on a reboot, the reboot flag will be set. The error flag is set if the file cannot be renamed (and /REBOOTOK is not used) or if the source file does not exist.

4.9.1.7 ReserveFile

```
[/nonfatal] [/r] file [file...]
```

Reserves a file in the data block for later use. As files are added in the order they are used in the script files that are used in the .onInit function, for example, might be added last and slow the loading of the installer. This is where this command comes useful, allowing you to speed up the loading process by including the file at the top of the data block instead of letting NSIS seek all the way down to the bottom of the *compressed* data block.

See [File](#) for more information about the parameters.

4.9.1.8 RMDir

```
[/r|/REBOOTOK] directory_name
```

Remove the specified directory (which should be a full path). Without /r, the directory will only be removed if it is completely empty. If /r is specified, the directory will be removed recursively, so all directories and files in the specified directory will be removed. If /REBOOTOK is specified, and the directory cannot be overwritten, then the directory will be deleted when the system reboots -- if the directory will be removed on a reboot, the reboot flag will be set. The error flag is set if the directory cannot be removed.

4.9.1.9 SetOutPath

outpath

Sets the output path (\$OUTDIR) and creates it (recursively if necessary), if it does not exist. Must be a full pathname, usually is just \$INSTDIR (you can specify \$INSTDIR if you are lazy with a single "-").

4.9.2 Registry, INI, File Instructions

4.9.2.1 DeleteINISec

```
ini_filename section_name
```

Deletes the entire section [section_name] from ini_filename. If the section could not be removed from the ini file, the error flag is set.

4.9.2.2 DeleteINIStr

```
ini_filename section_name str_name
```

Deletes the string str_name from section [section_name] from ini_filename. If the string could not be removed from the ini file, the error flag is set.

4.9.2.3 DeleteRegKey

`[/ifempty] root_key subkey`

Deletes a registry key. If `/ifempty` is specified, the registry key will only be deleted if it has no subkeys (otherwise, the whole registry tree will be removed). Valid values for `root_key` are listed under `WriteRegStr`. The error flag is set if the key could not be removed from the registry (or if it didn't exist to begin with).

4.9.2.4 DeleteRegValue

`root_key subkey key_name`

Deletes a registry value. Valid values for `root_key` are listed under `WriteRegStr`. The error flag is set if the value could not be removed from the registry (or if it didn't exist to begin with).

4.9.2.5 EnumRegKey

`user_var(output) root_key subkey index`

Set user variable `$x` with the name of the 'index'th registry key in `root_key\Subkey`. Valid values for `root_key` are listed under `WriteRegStr`. Returns an empty string if there are no more keys, and returns an empty string and sets the error flag if there is an error.

4.9.2.6 EnumRegValue

`user_var(output) root_key subkey index`

Set user variable `$x` with the name of the 'index'th registry value in `root_key\Subkey`. Valid values for `root_key` are listed under `WriteRegStr`. Returns an empty string if there are no more values, and returns an empty string and sets the error flag if there is an error.

4.9.2.7 ExpandEnvStrings

`user_var(output) string`

Expands environment variables in "string" into the user variable `$x`. If error, the variable is set to empty, and the error flag is set.

4.9.2.8 FlushINI

`ini_filename`

Flushes the INI file's buffers. Windows 9x keeps all changes to the INI file in memory. This command causes the changes to be written to the disk immediately. Use it if you edit the INI manually, delete it, move it or copy it right after you change it with [WriteINIStr](#), [DeleteINISec](#) or [DeleteINStr](#).

4.9.2.9 ReadEnvStr

`user_var(output) name`

Reads from the environment string "name" and sets the value into the user variable \$x. If there is an error reading the string, the user variable is set to empty, and the error flag is set.

4.9.2.10 ReadINIStr

`user_var(output) ini_filename section_name entry_name`

Reads from entry_name in [section_name] of ini_filename and stores the value into user variable \$x. The error flag will be set and \$x will be assigned to an empty string if the entry is not found.

4.9.2.11 ReadRegDWORD

`user_var(output) root_key sub_key name`

Reads a 32 bit DWORD from the registry into the user variable \$x. Valid values for root_key are listed under WriteRegStr. The error flag will be set and \$x will be set to an empty string ("" which is 0) if the DWORD is not present. If the value is present, but is not a DWORD, it will be read as a string and the error flag will be set.

4.9.2.12 ReadRegStr

`user_var(output) root_key sub_key name`

Reads from the registry into the user variable \$x. Valid values for root_key are listed under WriteRegStr. The error flag will be set and \$x will be set to an empty string ("" if the string is not present. If the value is present, but is of type REG_DWORD, it will be read and converted to a string and the error flag will be set.

4.9.2.13 WriteINIStr

`ini_filename section_name entry_name value`

Writes entry_name=value into [section_name] of ini_filename. The error flag is set if the string could not be written to the ini file.

4.9.2.14 WriteRegBin

`root_key subkey key_name valuedata`

This command writes a block of binary data to the registry. Valid values for root_key are listed under WriteRegStr. Valuedata is in hexadecimal (e.g. DEADBEEF01223211151). The error flag is set if the binary data could not be written to the registry. If the registry key doesn't exist it will be created.

4.9.2.15 WriteRegDWORD

root_key subkey key_name value

This command writes a dword (32 bit integer) to the registry (a user variable can be specified). Valid values for root_key are listed under WriteRegStr. The error flag is set if the dword could not be written to the registry. If the registry key doesn't exist it will be created.

4.9.2.16 WriteRegStr

root_key subkey key_name value

WriteRegExpandStr root_key subkey key_name value

Write a string to the registry. root_key must be one of:

- *HKCR* or *HKEY_CLASSES_ROOT*
- *HKLM* or *HKEY_LOCAL_MACHINE*
- *HKCU* or *HKEY_CURRENT_USER*
- *HKU* or *HKEY_USERS*
- *HKCC* or *HKEY_CURRENT_CONFIG*
- *HKDD* or *HKEY_DYN_DATA*
- *HKPD* or *HKEY_PERFORMANCE_DATA*

The error flag is set if the string could not be written to the registry. The type of the string will be REG_SZ for WriteRegStr, or REG_EXPAND_STR for WriteRegExpandStr. If the registry key doesn't exist it will be created.

4.9.3 General Purpose Instructions

4.9.3.1 CallInstDLL

dllfile [/NOUNLOAD] function_name

Calls a function_name inside a NSIS extension DLL. See Contrib\ExDLL for an example of how to make one. Extension DLLs can access the stack and variables. Use /NOUNLOAD to force the installer to leave the DLL loaded.

4.9.3.2 CopyFiles

[/SILENT] [/FILESONLY] filespec_on_destsys destination_path [size_of_files_in_kb]

Copies files from the source to the destination on the installing system. Useful with \$EXEDIR if you want to copy from installation media, or to copy from one place to another on the system. Uses SHFileOperation, so the user might see a status window of the copy operation if it is large (to disable this, use /SILENT). The last parameter specifies how big the copy is (in kilobytes), so that the installer can approximate the disk space requirements. On error, or if the user cancels the copy (only possible when /SILENT was omitted), the error flag is set. If /FILESONLY is specified, only files are copied.

4.9.3.3 CreateDirectory

path_to_create

Creates (recursively if necessary) the specified directory. The error flag is set if the directory couldn't be created.

4.9.3.4 CreateShortcut

link.lnk target.file [parameters [icon.file [icon_index_number [start_options [keyboard_shortcut [description]

Creates a shortcut 'link.lnk' that links to 'target.file', with optional parameters 'parameters'. The icon used for the shortcut is 'icon.file,icon_index_number'; for default icon settings use empty strings for both icon.file and icon_index_number. start_options should be one of: *SW_SHOWNORMAL*, *SW_SHOWMAXIMIZED*, *SW_SHOWMINIMIZED*, or an empty string. keyboard_shortcut should be in the form of 'flag|c' where flag can be a combination (using |) of: *ALT*, *CONTROL*, *EXT*, or *SHIFT*. c is the character to use (a-z, A-Z, 0-9, F1-F24, etc). Note that no spaces are allowed in this string. A good example is "ALT|CONTROL|F8". \$OUTDIR is used for the working directory. You can change it by using SetOutPath before creating the Shortcut. description should be the description of the shortcut, or comment as it is called under XP. The error flag is set if the shortcut cannot be created (i.e. the path does not exist, or some other error).

4.9.3.5 GetDLLVersion

filename user_var(high dword output) user_var(low dword output)

Gets the version information from the DLL (or any other executable containing version information) in "filename". Sets the user output variables with the high and low dwords of version information on success; on failure the outputs are empty and the error flag is set. The following example reads the DLL version and copies a human readable version of it into \$0:

```
GetDllVersion "$INSTDIR\MyDLL.dll" $R0 $R1
IntOp $R2 $R0 / 0x00010000
IntOp $R3 $R0 & 0x0000FFFF
IntOp $R4 $R1 / 0x00010000
IntOp $R5 $R1 & 0x0000FFFF
StrCpy $0 "$R2.$R3.$R4.$R5"
```

4.9.3.6 GetDLLVersionLocal

localfilename user_var(high dword output) user_var(low dword output)

This is similar to GetDLLVersion, only it acts on the system building the installer (it actually compiles into two StrCpy commands). Sets the two output variables with the DLL version information of the DLL on the build system.

4.9.3.7 GetFileTime

filename user_var(high dword output) user_var(low dword output)

Gets the last write time of "filename". Sets the user output variables with the high and low dwords of the timestamp on success; on failure the outputs are empty and the error flag is set.

4.9.3.8 GetFileTimeLocal

```
localfilename user_var(high dword output) user_var(low dword output)
```

This is similar to `GetFileTime`, only it acts on the system building the installer (it actually compiles into two `StrCpy` commands). Sets the two output variables with the file timestamp of the file on the build system.

4.9.3.9 GetFullPathName

```
[/SHORT] user_var(output) path_or_file
```

Assign to the user variable `$x`, the full path of the file specified. If the path portion of the parameter is not found, the error flag will be set and `$x` will be empty. If `/SHORT` is specified, the path is converted to the short filename form.

4.9.3.10 GetTempFileName

```
user_var(output)
```

Assign to the user variable `$x`, the name of a temporary file. The file will have been created, so you can then overwrite it with what you please. The name of the temporary file is guaranteed to be unique. Delete the file when done with it.

4.9.3.11 SearchPath

```
user_var(output) filename
```

Assign to the user variable `$x`, the full path of the file named by the second parameter. The error flag will be set and `$x` will be empty if the file cannot be found. Uses `SearchPath()` to search the system paths for the file.

4.9.3.12 SetFileAttributes

```
filename attribute1|attribute2|...
```

Sets the file attributes of 'filename'. Valid attributes can be combined with `|` and are:

- *NORMAL* or *FILE_ATTRIBUTE_NORMAL* (you can use 0 to abbreviate this)
- *ARCHIVE* or *FILE_ATTRIBUTE_ARCHIVE*
- *HIDDEN* or *FILE_ATTRIBUTE_HIDDEN*
- *OFFLINE* or *FILE_ATTRIBUTE_OFFLINE*
- *READONLY* or *FILE_ATTRIBUTE_READONLY*
- *SYSTEM* or *FILE_ATTRIBUTE_SYSTEM*
- *TEMPORARY* or *FILE_ATTRIBUTE_TEMPORARY*

The error flag will be set if the file's attributes cannot be set (i.e. the file doesn't exist, or you don't have the right permissions). You can only set attributes. It's not possible to unset them. If you want to remove an attribute use *NORMAL*. This way all attributes are erased. This command doesn't support wildcards.

4.9.3.13 RegDLL

`dllfile [entrypoint_name]`

Loads the specified DLL and calls `DllRegisterServer` (or `entrypoint_name` if specified). The error flag is set if an error occurs (i.e. it can't load the DLL, initialize OLE, find the entry point, or the function returned anything other than `ERROR_SUCCESS (=0)`).

4.9.3.14 UnRegDLL

`dllfile`

Loads the specified DLL and calls `DllUnregisterServer`. The error flag is set if an error occurs (i.e. it can't load the DLL, initialize OLE, find the entry point, or the function returned anything other than `ERROR_SUCCESS (=0)`).

4.9.4 Flow Control Instructions

4.9.4.1 Abort

`user_message`

Cancels the install, stops execution of script, and displays `user_message` in the status display. Note: you can use this from Callback functions to do special things. Page callbacks also uses `Abort` for special purposes.

4.9.4.2 Call

`function_name | :label_name`

Calls the function named `function_name`. If in the `Uninstall` section, `Call` can only be used with function names beginning with "un.". If the parameter starts with a ':' it will be treated as a label (so you can call to a label in your function – this is probably not going to be used most of the time).

4.9.4.3 ClearErrors

Clears the error flag.

4.9.4.4 GetCurrentAddress

`user_var(output)`

Gets the address of the current instruction (the `GetCurrentAddress`) and stores it in the output user variable. This user variable then can be passed to `Call` or `Goto`.

4.9.4.5 GetFunctionAddress

```
user_var(output) function_name
```

Gets the address of the function and stores it in the output user variable. This user variable then can be passed to Call or Goto. Note that if you Goto an address which is the output of GetFunctionAddress, your function will never be returned to (when the function you Goto'd to returns, you return instantly).

4.9.4.6 GetLabelAddress

```
user_var(output) label
```

Gets the address of the label and stores it in the output user variable. This user variable then can be passed to Call or Goto. Note that you may only call this with labels accessible from your function, but you can call it from anywhere (which is potentially dangerous). Note that if you Call the output of GetLabelAddress, code will be executed until it Return's (explicitly or implicitly at the end of a function), and then you will be returned to the statement after the Call.

4.9.4.7 Goto

```
label_to_jump_to | +offset| -offset| user_var(target)
```

If label is specified, goto the label 'label_to_jump_to:'.

If +offset or -offset is specified, jump is relative by offset instructions. Goto +1 goes to the next instruction, Goto -1 goes to the previous instruction, etc.

If a user variable is specified, jumps to absolute address (generally you will want to get this value from a function like GetLabelAddress). Compiler flag commands and SectionIn aren't instructions so jumping over them has no effect.

4.9.4.8 IfErrors

```
jumpto_iferror [jumpto_ifnoerror]
```

Checks and clears the error flag, and if it is set, it will goto jumpto_iferror, otherwise it will goto jumpto_ifnoerror. The error flag is set by other instructions when a recoverable error (such as trying to delete a file that is in use) occurs.

4.9.4.9 IfFileExists

```
file_to_check_for jump_if_present [jump_otherwise]
```

Checks for existence of file(s) file_to_check_for (which can be a wildcard, or a directory), and Gotos jump_if_present if the file exists, otherwise Gotos jump_otherwise. If you want to check to see if a file is a directory, use IfFileExists DIRECTORY*.*

4.9.4.10 IfRebootFlag

`[jump_if_set] [jump_if_not_set]`

Checks the reboot flag, and jumps to `jump_if_set` if the reboot flag is set, otherwise jumps to `jump_if_not_set`. The reboot flag can be set by Delete and Rename, or manually with `SetRebootFlag`.

4.9.4.11 IntCmp

`val1 val2 jump_if_equal [jump_if_val1_less] [jump_if_val1_more]`

Compares two integers `val1` and `val2`. If `val1` and `val2` are equal, Gotos `jump_if_equal`, otherwise if `val1 < val2`, Gotos `jump_if_val1_less`, otherwise if `val1 > val2`, Gotos `jump_if_val1_more`.

4.9.4.12 IntCmpU

`val1 val2 jump_if_equal [jump_if_val1_less] [jump_if_val1_more]`

Compares two unsigned integers `val1` and `val2`. If `val1` and `val2` are equal, Gotos `jump_if_equal`, otherwise if `val1 < val2`, Gotos `jump_if_val1_less`, otherwise if `val1 > val2`, Gotos `jump_if_val1_more`. Performs the comparison as unsigned integers.

4.9.4.13 MessageBox

`mb_option_list messagebox_text [return_check jumpto] [return_check_2 jumpto_2]`

Displays a MessageBox containing the text "`messagebox_text`". `mb_option_list` must be one or more of the following, delimited by `|`s (e.g. `MB_YESNO|MB_ICONSTOP`).

- `MB_OK` – Display with an OK button
- `MB_OKCANCEL` – Display with an OK and a cancel button
- `MB_ABORTRETRYIGNORE` – Display with abort, retry, ignore buttons
- `MB_RETRYCANCEL` – Display with retry and cancel buttons
- `MB_YESNO` – Display with yes and no buttons
- `MB_YESNOCANCEL` – Display with yes, no, cancel buttons
- `MB_ICONEXCLAMATION` – Display with exclamation icon
- `MB_ICONINFORMATION` – Display with information icon
- `MB_ICONQUESTION` – Display with question mark icon
- `MB_ICONSTOP` – Display with stop icon
- `MB_TOPMOST` – Make messagebox topmost
- `MB_SETFOREGROUND` – Set foreground
- `MB_RIGHT` – Right align text
- `MB_DEFBUTTON1` – Button 1 is default
- `MB_DEFBUTTON2` – Button 2 is default
- `MB_DEFBUTTON3` – Button 3 is default
- `MB_DEFBUTTON4` – Button 4 is default

`Return_check` can be 0 (or empty, or left off), or one of the following:

- *IDABORT* – Abort button
- *IDCANCEL* – Cancel button
- *IDIGNORE* – Ignore button
- *IDNO* – No button
- *IDOK* – OK button
- *IDRETRY* – Retry button
- *IDYES* – Yes button

If the return value of the `MessageBox` is `return_check`, the installer will `Goto` `jump_to`.

4.9.4.14 Return

Returns from a function or section.

4.9.4.15 Quit

Causes the installer to exit as soon as possible. After `Quit` is called, the installer will exit (no callback functions will get a chance to run).

4.9.4.16 SetErrors

Sets the error flag.

4.9.4.17 StrCmp

```
str1 str2 jump_if_equal [jump_if_not_equal]
```

Compares (case insensitively) `str1` to `str2`. If `str1` and `str2` are equal, Gotos `jump_if_equal`, otherwise Gotos `jump_if_not_equal`.

4.9.5 File Instructions

4.9.5.1 FileClose

```
handle
```

Closes a file handle opened with `FileOpen`.

4.9.5.2 FileOpen

```
user_var(handle output) filename openmode
```

Opens a file named "`filename`", and sets the `handle` output variable with the handle. The `openmode` should be one of "`r`" (read) "`w`" (write, all contents of file are destroyed) or "`a`" (append, meaning opened for both read and write,

contents preserved). In all open modes, the file pointer is placed at the beginning of the file. If the file cannot be opened, the handle output is set to empty, and the error flag is set.

4.9.5.3 FileRead

```
handle user_var(output) [maxlen]
```

Reads a string from a file opened with FileOpen. The string is read until either a newline (or carriage return newline pair) occurs, or until a null byte is read, or until maxlen is met (if specified). Strings are limited to 1024 characters. If the end of file is read and no more data is available, the output string will be empty, and the error flag will be set.

4.9.5.4 FileReadByte

```
handle user_var(output)
```

Reads a byte from a file opened with FileOpen. The byte is stored in the output as an integer (0–255). If the end of file is read and no more data is available, the output will be empty, and the error flag will be set.

4.9.5.5 FileSeek

```
handle offset [mode] [user_var(new position)]
```

Seeks a file opened with FileOpen. If mode is omitted or specified as SET, the file is positioned to "offset". If mode is specified as CUR, then the file pointer is moved by offset. If mode is specified as END, the file pointer is set to a position relative to EOF. If the final parameter "new position" is specified, the new file position will be stored to that variable.

4.9.5.6 FileWrite

```
handle string
```

Writes a string to a file opened with FileOpen. If an error occurs writing, the error flag will be set.

4.9.5.7 FileWriteByte

```
handle string
```

Writes the integer interpretation of 'string' to a file opened with FileOpen. Of course you can enter the integer value directly. The following code writes a "Carriage Return / Line Feed" – Enter to the file.

```
FileWriteByte file_handle "13"  
FileWriteByte file_handle "10"
```

You can If an error occurs writing, the error flag will be set. Note that the low byte of the integer is used, i.e. writing 256 is the same as writing 0, etc.

4.9.5.8 FindClose

handle

Closes a search opened with FindFirst.

4.9.5.9 FindFirst

user_var(handle output) user_var(filename output) filespec

Performs a search for 'filespec', placing the first file found in filename_output (a user variable). It also puts the handle of the search into handle_output (also a user variable). If no files are found, both outputs are set to empty, and the error flag is set. Best used with FindNext and FindClose. Note that the filename output is without path.

4.9.5.10 FindNext

handle user_var(filename_output)

Continues a search began with FindFirst. handle should be the handle_output_variable returned by FindFirst. If the search is completed (there are no more files), filename_output is set to empty, and the error flag is set. Note that the filename output is without path.

4.9.6 Uninstaller Instructions

4.9.6.1 WriteUninstaller

[Path\]exename.exe

Writes the uninstaller to the filename (and optionally path) specified. Only valid from within an install section or function, and requires that you have an uninstall section in your script. See also Uninstall configuration. You can call this one or more times to write out one or more copies of the uninstaller.

4.9.7 Miscellaneous Instructions

4.9.7.1 InitPluginsDir

Initializes the plugins dir (\$PLUGINS_DIR) if not already initialized.

4.9.7.2 SetShellVarContext

current|all

Sets the context of \$SMPROGRAMS and other shell folders. If set to 'current' (the default), the current user's shell folders are used. If set to 'all', the 'all users' shell folder is used. The all users folder may not be supported on all OSes.

If the all users folder is not found, the current user folder will be used. Please take into consideration that a "normal user" has no rights to write in the all users area. Only admins have full access rights to the all users area. You can check this by using the UserInfo Plugin. See Contrib\UserInfo\UserInfo.nsi for an example.

4.9.7.3 Sleep

`sleeptime_in_ms`

Pauses execution in the installer for `sleeptime_in_ms` milliseconds. `sleeptime_in_ms` can be a variable, e.g. "\$0" or a number, i.e. "666".

4.9.8 String Manipulation Instructions

4.9.8.1 StrCpy

`user_var(destination) str [maxlen] [start_offset]`

Sets the user variable \$x with str. Note that str can contain other variables, or the user variable being set (concatenating strings this way is possible, etc). If maxlen is specified, the string will be a maximum of maxlen characters (if maxlen is negative, the string will be truncated abs(maxlen) characters from the end). If start_offset is specified, the source is offset by it (if start_offset is negative, it will start abs(start_offset) from the end of the string).

4.9.8.2 StrLen

`user_var(length output) str`

Sets user variable \$x with the length of str.

4.9.9 Stack Support

4.9.9.1 Exch

`[user_var | stack_index]`

When no parameter is specified, exchanges the top two elements of the stack. When a parameter is specified and is a user variable, exchanges the top element of the stack with the parameter. When a parameter is specified and is a positive integer, Exch will swap the item on the top of the stack with the item that is specified by the offset from the top of the stack in the parameter. If there are not enough items on the stack to accomplish the exchange, a fatal error will occur (to help you debug your code :).

4.9.9.2 Pop

`user_var(out)`

Pops a string off of the stack into user variable \$x. If the stack is empty, the error flag will be set.

4.9.9.3 Push

string

Pushes a string onto the stack. The string can then be Popped off of the stack.

4.9.10 Integer Support

4.9.10.1 IntFmt

user_var(output) format numberstring

Formats the number in "numberstring" using the format "format", and sets the output to user variable \$x. Example format strings include "%08X" "%u"

4.9.10.2 IntOp

user_var(output) value1 OP [value2]

Combines value1 and (depending on OP) value2 into the user variable \$x. OP is defined as one of the following:

- + ADDs value1 and value2
- – SUBTRACTs value2 from value1
- * MULTIPLIEs value1 and value2
- / DIVIDEs value1 by value2
- % MODULUSs value1 by value2
- | BINARY ORs value1 and value2
- & BINARY ANDs value1 and value2
- ^ BINARY XORs value1 and value2
- ~ BITWISE NEGATEs value1 (i.e. 7 becomes 4294917288)
- ! LOGICALLY NEGATEs value1 (i.e. 7 becomes 0)
- || LOGICALLY ORs value1 and value2
- && LOGICALLY ANDs value1 and value2

4.9.11 Reboot Instructions

4.9.11.1 Reboot

Reboots the computer. Be careful with this one. If there is an error rebooting, this function sets the error flag and continues. If the reboot is successful, this instruction does not return.

4.9.11.2 SetRebootFlag

true|false

Sets the reboot flag to either true or false.

4.9.12 Install Logging Instructions

4.9.12.1 LogSet

on|off

Sets whether install logging to \$INSTDIR\install.log will happen. \$INSTDIR must have a value before you call this function or it will not work. Note that *NSIS_CONFIG_LOG* must be set in the installer configuration (it is not by default) to support this.

4.9.12.2 LogText

text

If installer logging is enabled, inserts text "text" into the log file.

4.9.13 Section Management

4.9.13.1 SectionSetFlags

section_index section_flags

Sets the section's flags. The flag is a 32 bit integer. The first bit (lowest) represents whether the section is currently enabled, the second bit represents whether the section is a subsection (don't modify this unless you really know what you are doing), the third bit represents whether the section is a subsection end (again, don't modify), the fourth bit represents whether the section is shown in bold or not, the fifth bit represents whether the section is read-only and the sixth bit represents whether the sub-section is to be automatically expanded. The error flag will be set if an out of range section is specified.

For an example of usage please see the [one-section.nsi](#) example.

4.9.13.2 SectionGetFlags

section_index user_var(output)

Retrieves the section's flags. See above for a description of the flag. The error flag will be set if an out of range section is specified.

4.9.13.3 SectionSetText

```
section_index section_text
```

Sets the description for the section `section_index`. To set a subsection, you must use `-` at the beginning of the text. The error flag will be set if an out of range section is specified.

4.9.13.4 SectionGetText

```
section_index user_var(output)
```

Stores the text description of the section `section_index` into the output. If the section is hidden, stores an empty string. The error flag will be set if an out of range section is specified.

4.9.13.5 SectionSetInstTypes

```
section_index inst_types
```

Sets the install types the section `section_index` defaults to the enabled state in. Note that the section index starts with zero. Every bit of `inst_types` is a flag that tells if the section is in that install type or not. For example, if you have 3 install types and you want the first section to be included in install types 1 and 3, then the command should look like this:

```
SectionSetInstTypes 0 5
```

because the binary value for 5 is "00000101". The error flag will be set if the section index specified is out of range.

4.9.13.6 SectionGetInstTypes

```
section_index user_var(output)
```

Retrieves the install types flags array of a section. See above explanation about `SectionSetInstTypes` for a description of how to deal with the output. The error flag will be set if the section index specified is out of range.

4.9.14 User Interface Instructions

4.9.14.1 BringToFront

Makes the installer window visible and brings it to the top of the window list (i.e. if a command was executed that shows itself in front of the installer, a `BringToFront` would bring the installer back in focus).

4.9.14.2 CreateFont

```
user_var(handle output) face_name [height] [weight] [/ITALIC] [/UNDERLINE] [/STRIKE]
```

Creates a font and puts its handle into `user_var`. For more information about the different parameters have a look at [MSDN's page about the Win32 API function CreateFont\(\)](#).

4.9.14.3 DetailPrint

`user_message`

Adds the string "user_message" to the details view of the installer.

4.9.14.4 FindWindow

`user_var(hwnd output) windowclass [windowtitle] [windowparent] [childafter]`

Searches for a window. Behaves like the win32 `FindWindowEx()`. Searches by windowclass (and/or windowtitle if specified). If windowparent or childafter are specified, the search will be restricted as such. If windowclass or windowtitle is specified as "", they will not be used for the search. If the window is not found, the user variable returned is 0. To accomplish old-style FindWindow behavior, use FindWindow with `SendMessage`.

4.9.14.5 GetDlgItem

`user_var(output) dialog item_id`

Retrieves the handle of a control identified by `item_id` in the specified dialog box `dialog`. If you want to get the handle of a control on the inner dialog, first use `FindWindow user_var(output) "#32770" "" $HWNDPARENT` to get the handle of the inner dialog.

4.9.14.6 HideWindow

Hides the installer.

4.9.14.7 IsWindow

`HWND jump_if_window [jump_if_not_window]`

If `HWND` is a window, Gotos `jump_if_window`, otherwise, Gotos `jump_if_not_window` (if specified).

4.9.14.8 SendMessage

`HWND msg wparam lparam [user_var(return value)] [/TIMEOUT=time_in_ms]`

Sends a message to `HWND`. If a user variable `$x` is specified as the last parameter (or one before the last if you use `/TIMEOUT`), the return value of `SendMessage` will be stored to it. Note that when specifying 'msg' you must just use the integer value of the message. If you wish to send strings use "STR:a string" as `wParam` or `lParam` where needed.

- `WM_CLOSE` 16
- `WM_COMMAND` 273

- `WM_USER 1024`

Include `WinMessages.nsh` to have all of Windows messages defined in your script.

To send a string param, put `STR:` before the parameter, for example: `"STR:Some string"`.

Use `/TIMEOUT=time_in_ms` to specify the duration, in milliseconds, of the time-out period.

4.9.14.9 SetAutoClose

`true|false`

Overrides the default auto window-closing flag (specified for the installer using `AutoCloseWindow`, and false for the uninstaller). Specify 'true' to have the install window immediately disappear after the install has completed, or 'false' to make it require a manual close.

4.9.14.10 SetBrandingImage

`[/IMGID=item_id_in_dialog] [/RESIZETO FIT] path_to_bitmap_file.bmp`

Sets the current bitmap file displayed as the branding image. If no `IMGID` is specified, the first image control found will be used, or the image control created by `AddBrandingImage`. Note that this bitmap must be present on the user's machine. Use `File` first to put it there. If `/RESIZETO FIT` is specified the image will be automatically resized (very poorly) to the image control size. If you used `AddBrandingImage` you can get this size, by compiling your script and watching for `AddBrandingImage` output, it will tell you the size. `SetBrandingImage` will not work when called from `.onInit!`

4.9.14.11 SetDetailsView

`show|hide`

Shows or hides the details, depending on which parameter you pass. Overrides the default details view, which is set via `ShowInstDetails`.

4.9.14.12 SetDetailsPrint

`none|listonly|textonly|both|lastused`

Sets mode at which commands print their status. `None` has commands be quiet, `listonly` has status text only added to the listbox, `textonly` has status text only printed to the status bar, and `both` enables both (the default). For extracting many small files, `textonly` is recommended (especially on win9x with smooth scrolling enabled).

4.9.14.13 SetBkColor

`hwnd color`

Sets a background color for a static control, edit control, button or a dialog. Use `GetDlgItem` to get the handle (HWND) of the control. To make the control transparent specify "transparent" as the color value. *Do not* use this on branding image controls!

4.9.14.14 ShowWindow

`hwnd show_state`

Sets the visibility of a window. Possible `show_states` are the same as [Windows ShowWindow](#) function. `SW_*` constants are defined in [Include\WinMessages.nsh](#).

4.9.15 Multiple Languages Instructions

4.9.15.1 LoadLanguageFile

`language_file.nlf`

Loads a language file for the construction of a language table. All of the language files that come with NSIS are in [Contrib\Language Files](#)

For ease of use `LoadLanguageFile` defines `${LANG_language_file}` as the language id. Use it with `/LANG`, `LangString`, `LangStringUP`, and `LangDLL`.

4.9.15.2 LangString and LangStringUP

`[un.]name language_id string`

Defines a multilingual string and spares the comparing of `$LANGUAGE` to every language you have in your installer for every string you use. This also allows you to make section names and install types multilingual. To use in the uninstaller make sure you name the string with the `un.` prefix before.

`LangStrings` can you only be used on their own. You can't include them in other strings. "look at my `$(string)`! isn't it beautiful?" will be seen exactly as written, `$(string)` will not be expanded. If you want to use `LangStrings` in other strings you can first copy the `LangString` to a variable and then use the variable wherever you want. This is a temporary situation, it will be changed before NSIS 2 final.

Note #1: Unlike defines that use curly braces – {}, multilingual strings use parenthesis – ().

Note #2: If you see weird characters between letters in the string when you use `LangString`, use `LangStringUP` (`LangString` for unprocessed string such as `InstType`)

For example, instead of:

```
StrCmp $LANGUAGE ${LANG_ENGLISH} 0 +2
  MessageBox MB_OK "English message"
StrCmp $LANGUAGE ${LANG_FRENCH} 0 +2
  MessageBox MB_OK "French message"
StrCmp $LANGUAGE ${LANG_KOREAN} 0 +2
```

```
MessageBox MB_OK "Korean message"
```

Use:

```
LangString message ${LANG_ENGLISH} "English message"  
LangString message ${LANG_FRENCH} "French message"  
LangString message ${LANG_KOREAN} "Korean message"  
  
MessageBox MB_OK $(message)
```

4.10 Multiple Languages

As of version 2 NSIS fully supports multiple languages. An installer can have more than one language. Each string in the installer can be easily translated, and so can script strings such as messages in a message box.

Each installer has one or more language table which holds references to strings in the strings table. To create a language table all you need to do is use LoadLanguageFile, define strings used in your installer for that language such as Name and Caption, message box, install type, and other strings using LangString or LangStringUP and you have built your installer a language table.

For an example of usage see languages.nsi.

4.10.1 Language Selection

When the installer starts up it goes through these stages to select the interface language:

1. Find a perfect match between the user default language (GetUserDefaultLangID())
2. If there is no perfect match, find a primary language match
3. If no match, take the first language defined in the script
4. If \$LANGUAGE has changed during .onInit, go through steps 1 to 3 with the language inside \$LANGUAGE instead of the default user language.

4.10.2 The /LANG Parameter

All of the installer (and uninstaller) attributes setting commands have an optional parameter /LANG. This parameter tells the script compiler in which language table to put the specified string.

For example:

```
Caption /LANG=${LANG_ENGLISH} "English caption"  
Caption /LANG=${LANG_FRENCH} "French caption"  
Caption /LANG=${LANG_DUTCH} "Dutch caption"
```

When the installer will select the English language the caption will be "English caption", when it selects French the caption will be "French caption" and when it selects Dutch the caption will be "Dutch caption".

If no /LANG parameter is specified, the compiler will assume the last used language, or the last loaded language.

```
LoadLanguageFile "${NSISDIR}\Language files\English.nlf"
```

```
Name "English name"
LoadLanguageFile "${NSISDIR}\Language files\German.nlf"
Name "German name"
Caption "German caption"
Caption /LANG=${LANG_ENGLISH} "English caption"
ComponentText "English components text"
ComponentText /LANG=${LANG_GERMAN} "German components text"
```

4.10.3 LangDLL Plug-in

The LangDLL plug-in lets you give the user the option to choose the language of the installer. Just push the language id (\${LANG_*}) and its name for every language in your installer, then the number of languages pushed, the caption, and the text that tells the user to select the language, call the plug-in function named LangDialog, pop the returned value into \$LANGUAGE and you're good to go. If the user click on the cancel button the return value will be "cancel".

For an example of usage see [languages.nsi](#).

4.11 Plugin DLLs

The abilities of the NSIS scripting language can be extended by utilising functionality provided in a DLL file. Probably the best known example of this is the InstallOptions.dll bundled with every NSIS release.

When the NSIS compiler starts it scans the plugins directory for DLLs and makes a list of the plugins found and their exported functions. During compilation if a sequence such as fred::flintstone is encountered where the compiler expected to find a language keyword the compiler will look through this list. If a list entry specifies that fred.dll exports function flintstone NSIS will pack the fred.dll file into the created installer binary.

During execution of the created installer if a plugin command is executed NSIS will unpack the necessary DLL to the \$TEMP directory, push all of the arguments specified (right-to-left order), and then execute the DLL function. If the /NOUNLOAD option is specified the DLL will not be unloaded until the installer exits or the next time you use the DLL without /NOUNLOAD. Please note that the last call to the plugin must not have the /NOUNLOAD flag or the plugin will not be deleted from \$PLUGINS_DIR, thus garbage will be left on the user's machine.

4.11.1 Using Plugin Commands

The following two examples both invoke the same plugin command. The first example shows the (still okay) syntax that scripts written for versions of NSIS earlier than 2.0a4 had to use, and the second is how it can be scripted more succinctly now. The newer syntax automatically handles packing & extraction of the DLL file, and stacks up arguments for you too.

```
; Pre 2.0a4 syntax
SetOutPath $TEMP
GetTempFileName $8
File /oname=$8 InstallOptions.dll
Push "ini_file_location.ini"
CallInstDLL dialog
```

```
; Newer syntax
InstallOptions::dialog "ini_file_location.ini"
```

InstallOptions needs the name of its ini file as a parameter to the dialog function so it has to be pushed onto the stack before the dialog call is made. Some plugin commands may not need any parameters on the stack, others might require two or three. To use a plugin command you will need to read the documentation for the plugin so that you know what parameters its functions require, if any.

4.11.2 Disabling Plugin Unloading

CallInstDLL has an option not to unload the DLL after usage. To use it with the newer plugin command syntax just specify the first parameter as /NOUNLOAD. For example:

```
InstallOptions::dialog /NOUNLOAD "ini_file_location.ini"
```

You can also use SetPluginUnload always off to avoid writing /NOUNLOAD each and every time you use the same plugin.