Nim Compiler User Guide 0.17.2

Andreas Rumpf

January 3, 2018

Contents

1 Introduction **2**

2 Compiler Usage **2**
2.1 Command line switches . . . 2
2.2 List of warnings . . . 4
2.3 Verbosity levels . . . 4
2.4 Compile time symbols . . . 4
2.5 Configuration files . . . 5
2.6 Search path handling . . . 5
2.7 Generated C code directory . . . 6

3 Compilation cache **6**

4 Compiler Selection **6**

5 Cross compilation **6**

6 DLL generation **7**

7 Additional compilation switches **7**

8 Additional Features **7**
8.1 LineDir option . . . 7
8.2 StackTrace option . . . 7
8.3 LineTrace option . . . 7
8.4 Debugger option . . . 9
8.5 Breakpoint pragma . . . 9

9 DynlibOverride **9**

10 Backend language options **9**

11 Nim documentation tools **9**

12 Nim idetools integration **9**

13 Nim for embedded systems **9**

14 Nim for realtime systems **9**

15 Debugging with Nim **10**

16 Optimizing for Nim **10**
16.1 Optimizing string handling . . . 10

"Look at you, hacker. A pathetic creature of meat and bone, panting and sweating as you run through my corridors. How can you challenge a perfect, immortal machine?"

1 Introduction

This document describes the usage of the *Nim compiler* on the different supported platforms. It is not a definition of the Nim programming language (therefore is the manual).

Nim is free software; it is licensed under the MIT License.

2 Compiler Usage

2.1 Command line switches

Basic command line switches are:

Usage:

```
nim command [options] [projectfile] [arguments]
```

Command: compile, c compile project with default code generator (C)

doc generate the documentation for inputfile

doc2 generate the documentation for inputfile

Arguments: arguments are passed to the program being run (if –run option is selected)

Options: -p, –path:PATH add path to search paths

-d, –define:SYMBOL(:VAL) define a conditional symbol (Optionally: Define the value for that symbol)

-u, –undef:SYMBOL undefine a conditional symbol

-f, –forceBuild force rebuilding of all modules

–stackTrace:on|off turn stack tracing on|off

–lineTrace:on|off turn line tracing on|off

–threads:on|off turn support for multi-threading on|off

-x, –checks:on|off turn all runtime checks on|off

–objChecks:on|off turn obj conversion checks on|off

–fieldChecks:on|off turn case variant field checks on|off

–rangeChecks:on|off turn range checks on|off

–boundChecks:on|off turn bound checks on|off

–overflowChecks:on|off turn int over-/underflow checks on|off

-a, –assertions:on|off turn assertions on|off

–floatChecks:on|off turn all floating point (NaN/Inf) checks on|off

–nanChecks:on|off turn NaN checks on|off

–infChecks:on|off turn Inf checks on|off

–deadCodeElim:on|off whole program dead code elimination on|off

–opt:none|speed|size optimize not at all or for speed|size Note: use -d:release for a release build!

–debugger:native|endb use native debugger (gdb) | ENDB (experimental)

–app:console|gui|lib|staticlib generate a console app|GUI app|DLL|static library

-r, –run run the compiled program with given arguments

–advanced show advanced command line switches

-h, –help show this help

Note, single letter options that take an argument require a colon. E.g. -p:PATH.

Advanced command line switches are:

Advanced commands: compileToC, cc compile project with C code generator

compileToCpp, cpp compile project to C++ code

compileToOC, objc compile project to Objective C code

js compile project to Javascript

e run a Nimscript file

rst2html convert a reStructuredText file to HTML

rst2tex convert a reStructuredText file to TeX

jsondoc extract the documentation to a json file

jsondoc2 extract documentation to a json file (uses doc2)

buildIndex build an index for the whole documentation

run run the project (with Tiny C backend; buggy!)

genDepend generate a DOT file containing the module dependency graph

dump dump all defined conditionals and search paths

check checks the project for syntax and semantic

Advanced options: -o:FILE, –out:FILE set the output filename

–**stdout** output to stdout

–**colors:on|off** turn compiler messages coloring on|off

–**listFullPaths** list full paths in messages

-w:on|off|list, –warnings:on|off|list turn all warnings on|off or list all available

–**warning[X]:on|off** turn specific warning X on|off

–**hints:on|off|list** turn all hints on|off or list all available

–**hint[X]:on|off** turn specific hint X on|off

–**lib:PATH** set the system library path

–**import:PATH** add an automatically imported module

–**include:PATH** add an automatically included module

–**nimcache:PATH** set the path used for generated files

–**header:FILE** the compiler should produce a .h file (FILE is optional)

-c, –compileOnly compile only; do not assemble or link

–**noLinking** compile but do not link

–**noMain** do not generate a main procedure

–**genScript** generate a compile script (in the 'nimcache' subdirectory named 'compile_$project$scriptext')

–**os:SYMBOL** set the target operating system (cross-compilation)

–**cpu:SYMBOL** set the target processor (cross-compilation)

–**debuginfo** enables debug information

-t, –passC:OPTION pass an option to the C compiler

-l, –passL:OPTION pass an option to the linker

–**cincludes:DIR** modify the C compiler header search path

–**clibdir:DIR** modify the linker library search path

–**clib:LIBNAME** link an additional C library (you should omit platform-specific extensions)

–**genMapping** generate a mapping file containing (Nim, mangled) identifier pairs

–**project** document the whole project (doc2)

–**docSeeSrcUrl:url** activate 'see source' for doc and doc2 commands (see doc.item.seesrc in config/nimdoc.cfg)

–**lineDir:on|off** generation of #line directive on|off

–**embedsrc** embeds the original source code as comments in the generated output

–**threadanalysis:on|off** turn thread analysis on|off

–**tlsEmulation:on|off** turn thread local storage emulation on|off

–**taintMode:on|off** turn taint mode on|off

–**implicitStatic:on|off** turn implicit compile time evaluation on|off

–**patterns:on|off** turn pattern matching on|off

–**memTracker:on|off** turn memory tracker on|off

–**excessiveStackTrace:on|off** stack traces use full file paths

–**skipCfg** do not read the general configuration file

–**skipUserCfg** do not read the user's configuration file

–**skipParentCfg** do not read the parent dirs' configuration files

–**skipProjCfg** do not read the project's configuration file

–**gc:refc|v2|markAndSweep|boehm|go|none|regions** select the GC to use; default is 'refc'

–**index:on|off** turn index file generation on|off

–**putenv:key=value** set an environment variable

–**NimblePath:PATH** add a path for Nimble support

–**noNimblePath** deactivate the Nimble path

–**noCppExceptions** use default exception handling with C++ backend

–**excludePath:PATH** exclude a path from the list of search paths

–**dynlibOverride:SYMBOL** marks SYMBOL so that dynlib:SYMBOL has no effect and can be statically linked instead; symbol matching is fuzzy so that –dynlibOverride:lua matches dynlib: "liblua.so.3"

–**listCmd** list the commands used to execute external programs

–**parallelBuild:0|1|...** perform a parallel build value = number of processors (0 for auto-detect)

–**verbosity:0|1|2|3** set Nim's verbosity level (1 is default)

–**experimental** enable experimental language features

-v, –version show detailed version information

2.2 List of warnings

Each warning can be activated individually with `-warning[NAME]:on|off` or in a `push` pragma.

2.3 Verbosity levels

2.4 Compile time symbols

Through the `-d:x` or `-define:x` switch you can define compile time symbols for conditional compilation. The defined switches can be checked in source code with the when statement and defined proc. The typical use of this switch is to enable builds in release mode (`-d:release`) where certain safety checks are omitted for better performance. Another common use is the `-d:ssl` switch to activate SSL sockets.

Additionally, you may pass a value along with the symbol: `-d:x=y` which may be used in conjunction with the compile time define pragmas to override symbols during build time.

Name	Description
CannotOpenFile	Some file not essential for the compiler's working could not be opened.
OctalEscape	The code contains an unsupported octal sequence.
Deprecated	The code uses a deprecated symbol.
ConfigDeprecated	The project makes use of a deprecated config file.
SmallLshouldNotBeUsed	The letter 'l' should not be used as an identifier.
EachIdentIsTuple	The code contains a confusing `var` declaration.
ShadowIdent	A local variable shadows another local variable of an outer scope.
User	Some user defined warning.

Level	Description
0	Minimal output level for the compiler.
1	Displays compilation of all the compiled files, including those imported by other modules or through the compile pragma. This is the default level.
2	Displays compilation statistics, enumerates the dynamic libraries that will be loaded by the final binary and dumps to standard output the result of applying a filter to the source code if any filter was used during compilation.
3	In addition to the previous levels dumps a debug stack trace for compiler developers.

2.5 Configuration files

Note: The *project file name* is the name of the `.nim` file that is passed as a command line argument to the compiler.

The `nim` executable processes configuration files in the following directories (in this order; later files overwrite previous settings):

1. `$nim/config/nim.cfg`, `/etc/nim.cfg` (UNIX) or `%NIMROD%/config/nim.cfg` (Windows). This file can be skipped with the `-skipCfg` command line option.
2. `/home/$user/.config/nim.cfg` (UNIX) or `%APPDATA%/nim.cfg` (Windows). This file can be skipped with the `-skipUserCfg` command line option.
3. `$parentDir/nim.cfg` where `$parentDir` stands for any parent directory of the project file's path. These files can be skipped with the `-skipParentCfg` command line option.
4. `$projectDir/nim.cfg` where `$projectDir` stands for the project file's path. This file can be skipped with the `-skipProjCfg` command line option.
5. A project can also have a project specific configuration file named `$project.nim.cfg` that resides in the same directory as `$project.nim`. This file can be skipped with the `-skipProjCfg` command line option.

Command line settings have priority over configuration file settings.

The default build of a project is a debug build. To compile a release build define the `release` symbol:

```
nim c -d:release myproject.nim
```

2.6 Search path handling

Nim has the concept of a global search path (PATH) that is queried to determine where to find imported modules or include files. If multiple files are found an ambiguity error is produced.

`nim dump` shows the contents of the PATH.

However before the PATH is used the current directory is checked for the file's existence. So if PATH contains `$lib` and `$lib/bar` and the directory structure looks like this:

```
$lib/x.nim
$lib/bar/x.nim
foo/x.nim
foo/main.nim
other.nim
```

And `main` imports `x`, `foo/x` is imported. If `other` imports `x` then both `$lib/x.nim` and `$lib/bar/x.nim` match and so the compiler should reject it. Currently however this check is not implemented and instead the first matching file is used.

2.7 Generated C code directory

The generated files that Nim produces all go into a subdirectory called `nimcache` in your project directory. This makes it easy to delete all generated files. Files generated in this directory follow a naming logic which you can read about in the Nim Backend Integration document.

However, the generated C code is not platform independent. C code generated for Linux does not compile on Windows, for instance. The comment on top of the C file lists the OS, CPU and CC the file has been compiled for.

3 Compilation cache

Warning: The compilation cache is still highly experimental!

The `nimcache` directory may also contain so called rod or symbol files. These files are pre-compiled modules that are used by the compiler to perform incremental compilation. This means that only modules that have changed since the last compilation (or the modules depending on them etc.) are re-compiled. However, per default no symbol files are generated; use the `-symbolFiles:on` command line switch to activate them.

Unfortunately due to technical reasons the `-symbolFiles:on` needs to *aggregate* some generated C code. This means that the resulting executable might contain some cruft even when dead code elimination is turned on. So the final release build should be done with `-symbolFiles:off`.

Due to the aggregation of C code it is also recommended that each project resides in its own directory so that the generated `nimcache` directory is not shared between different projects.

4 Compiler Selection

To change the compiler from the default compiler (at the command line):

```
nim c --cc:llvm_gcc --compile_only myfile.nim
```

This uses the configuration defined in `config\nim.cfg` for `lvm_gcc`.

If nimcache already contains compiled code from a different compiler for the same project, add the `-f` flag to force all files to be recompiled.

The default compiler is defined at the top of `config\nim.cfg`. Changing this setting affects the compiler used by `koch` to (re)build Nim.

5 Cross compilation

To cross compile, use for example:

```
nim c --cpu:i386 --os:linux --compileOnly --genScript myproject.nim
```

Then move the C code and the compile script `compile_myproject.sh` to your Linux i386 machine and run the script.

Another way is to make Nim invoke a cross compiler toolchain:

```
nim c --cpu:arm --os:linux myproject.nim
```

For cross compilation, the compiler invokes a C compiler named like `$cpu.$os.$cc` (for example arm.linux.gcc) and the configuration system is used to provide meaningful defaults. For example for `ARM` your configuration file should contain something like:

```
arm.linux.gcc.path = "/usr/bin"
arm.linux.gcc.exe = "arm-linux-gcc"
arm.linux.gcc.linkerexe = "arm-linux-gcc"
```

6 DLL generation

Nim supports the generation of DLLs. However, there must be only one instance of the GC per process/address space. This instance is contained in `nimrtl.dll`. This means that every generated Nim DLL depends on `nimrtl.dll`. To generate the "nimrtl.dll" file, use the command:

```
nim c -d:release lib/nimrtl.nim
```

To link against `nimrtl.dll` use the command:

```
nim c -d:useNimRtl myprog.nim
```

Note: Currently the creation of `nimrtl.dll` with thread support has never been tested and is unlikely to work!

7 Additional compilation switches

The standard library supports a growing number of `useX` conditional defines affecting how some features are implemented. This section tries to give a complete list.

8 Additional Features

This section describes Nim's additional features that are not listed in the Nim manual. Some of the features here only make sense for the C code generator and are subject to change.

8.1 LineDir option

The `lineDir` option can be turned on or off. If turned on the generated C code contains `#line` directives. This may be helpful for debugging with GDB.

8.2 StackTrace option

If the `stackTrace` option is turned on, the generated C contains code to ensure that proper stack traces are given if the program crashes or an uncaught exception is raised.

8.3 LineTrace option

The `lineTrace` option implies the `stackTrace` option. If turned on, the generated C contains code to ensure that proper stack traces with line number information are given if the program crashes or an uncaught exception is raised.

Define	Effect
`release`	Turns off runtime checks and turns on the optimizer.
`useWinAnsi`	Modules like `os` and `osproc` use the Ansi versions of the Windows API. The default build uses the Unicode version.
`useFork`	Makes `osproc` use `fork` instead of `posix_spawn`.
`useNimRtl`	Compile and link against `nimrtl.dll`.
`useMalloc`	Makes Nim use C's malloc instead of Nim's own memory manager, ableit prefixing each allocation with its size to support clearing memory on reallocation. This only works with `gc:none`.
`useRealtimeGC`	Enables support of Nim's GC for *soft* realtime systems. See the documentation of the gc for further information.
`nodejs`	The JS target is actually `node.js`.
`ssl`	Enables OpenSSL support for the sockets module.
`memProfiler`	Enables memory profiling for the native GC.
`uClibc`	Use uClibc instead of libc. (Relevant for Unix-like OSes)
`checkAbi`	When using types from C headers, add checks that compare what's in the Nim file with what's in the C header (requires a C compiler with _Static_assert support, like any C11 compiler)
`tempDir`	This symbol takes a string as its value, like `-define:tempDir:/some/temp/path` to override the temporary directory returned by `os.getTempDir()`. The value **should** end with a directory separator character. (Relevant for the Android platform)
`useShPath`	This symbol takes a string as its value, like `-define:useShPath:/opt/sh/bin/sh` to override the path for the `sh` binary, in cases where it is not located in the default location `/bin/sh`

8.4 Debugger option

The `debugger` option enables or disables the *Embedded Nim Debugger*. See the documentation of endb for further information.

8.5 Breakpoint pragma

The *breakpoint* pragma was specially added for the sake of debugging with ENDB. See the documentation of endb for further information.

9 DynlibOverride

By default Nim's `dynlib` pragma causes the compiler to generate `GetProcAddress` (or their Unix counterparts) calls to bind to a DLL. With the `dynlibOverride` command line switch this can be prevented and then via `-passL` the static library can be linked against. For instance, to link statically against Lua this command might work on Linux:

```
nim c --dynlibOverride:lua --passL:liblua.lib program.nim
```

10 Backend language options

The typical compiler usage involves using the `compile` or `c` command to transform a `.nim` file into one or more `.c` files which are then compiled with the platform's C compiler into a static binary. However there are other commands to compile to C++, Objective-C or Javascript. More details can be read in the Nim Backend Integration document.

11 Nim documentation tools

Nim provides the doc and doc2 commands to generate HTML documentation from `.nim` source files. Only exported symbols will appear in the output. For more details see the docgen documentation.

12 Nim idetools integration

Nim provides language integration with external IDEs through the idetools command. See the documentation of idetools for further information.

13 Nim for embedded systems

The standard library can be avoided to a point where C code generation for 16bit micro controllers is feasible. Use the standalone target (`-os:standalone`) for a bare bones standard library that lacks any OS features.

To make the compiler output code for a 16bit target use the `-cpu:avr` target.

For example, to generate code for an AVR processor use this command:

```
nim c --cpu:avr --os:standalone --deadCodeElim:on --genScript x.nim
```

For the `standalone` target one needs to provide a file `panicoverride.nim`. See `tests/manyloc/standalone/panicoverride.nim` for an example implementation. Additionally, users should specify the amount of heap space to use with the `-d:StandaloneHeapSize=<size>` command line switch. Note that the total heap size will be `<size> * sizeof(float64)`.

14 Nim for realtime systems

See the documentation of Nim's soft realtime GC for further information.

15 Debugging with Nim

Nim comes with its own *Embedded Nim Debugger*. See the documentation of endb for further information.

16 Optimizing for Nim

Nim has no separate optimizer, but the C code that is produced is very efficient. Most C compilers have excellent optimizers, so usually it is not needed to optimize one's code. Nim has been designed to encourage efficient code: The most readable code in Nim is often the most efficient too.

However, sometimes one has to optimize. Do it in the following order:

1. switch off the embedded debugger (it is **slow**!)
2. turn on the optimizer and turn off runtime checks
3. profile your code to find where the bottlenecks are
4. try to find a better algorithm
5. do low-level optimizations

This section can only help you with the last item.

16.1 Optimizing string handling

String assignments are sometimes expensive in Nim: They are required to copy the whole string. However, the compiler is often smart enough to not copy strings. Due to the argument passing semantics, strings are never copied when passed to subroutines. The compiler does not copy strings that are a result from a procedure call, because the callee returns a new string anyway. Thus it is efficient to do:

```
var s = procA() # assignment will not copy the string; procA allocates a new
                # string already
```

However it is not efficient to do:

```
var s = varA    # assignment has to copy the whole string into a new buffer!
```

For `let` symbols a copy is not always necessary:

```
let s = varA    # may only copy a pointer if it safe to do so
```

If you know what you're doing, you can also mark single string (or sequence) objects as shallow:

```
var s = "abc"
shallow(s) # mark 's' as shallow string
var x = s  # now might not copy the string!
```

Usage of `shallow` is always safe once you know the string won't be modified anymore, similar to Ruby's freeze.

The compiler optimizes string case statements: A hashing scheme is used for them if several different string constants are used. So code like this is reasonably efficient:

```
case normalize(k.key)
of "name": c.name = v
of "displayname": c.displayName = v
of "version": c.version = v
of "os": c.oses = split(v, {';'})
of "cpu": c.cpus = split(v, {';'})
of "authors": c.authors = split(v, {';'})
of "description": c.description = v
of "app":
  case normalize(v)
  of "console": c.app = appConsole
  of "gui": c.app = appGUI
  else: quit(errorStr(p, "expected: console or gui"))
of "license": c.license = UnixToNativePath(k.value)
else: quit(errorStr(p, "unknown variable: " & k.key))
```