

Nimrod Compiler User Guide 0.9.2

Andreas Rumpf

May 21, 2013

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	Configuration files	4
2.5	Generated C code directory	6
3	Compilation cache	6
4	Cross compilation	6
5	DLL generation	7
6	Additional compilation switches	7
7	Additional Features	7
7.1	NoDecl pragma	7
7.2	Header pragma	8
7.3	IncompleteStruct pragma	8
7.4	Compile pragma	8
7.5	Link pragma	8
7.6	Emit pragma	8
7.7	ImportCpp pragma	8
7.8	ImportObjC pragma	9
7.9	LineDir option	9
7.10	StackTrace option	9
7.11	LineTrace option	9
7.12	Debugger option	10
7.13	Breakpoint pragma	10
7.14	Volatile pragma	10
8	Nimrod interactive mode	10
9	Nimrod for embedded systems	10
10	Nimrod for realtime systems	10
11	Debugging with Nimrod	10
12	Optimizing for Nimrod	11
12.1	Optimizing string handling	11
13	The JavaScript target	12

"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 *Nimrod compiler* on the different supported platforms. It is not a definition of the Nimrod programming language (therefore is the manual).

Nimrod is free software; it is licensed under the GNU General Public License.

2 Compiler Usage

2.1 Command line switches

Basic command line switches are:

Usage:

```
nimrod 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 the whole project

i start Nimrod in interactive mode (limited)

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** define a conditional 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

-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

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

rst2html convert a reStructuredText file to HTML

rst2tex convert a reStructuredText file to TeX

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

idetools compiler support for IDEs: possible options:

-track:FILE,LINE,COL track a file/cursor position

-suggest suggest all possible symbols at position

-def list all possible definitions at position

-context list possible invocation context

-usages list all usages of the symbol at position

-eval evaluates an expression

Advanced options:

-m, **-mainmodule:FILE** set the project main module

-o, **-out:FILE** set the output filename

-stdout output to stdout

-listFullPaths list full paths in messages

-w, **-warnings:on|off** turn all warnings on|off

-warning[X]:on|off turn specific warning X on|off

-hints:on|off turn all hints on|off

-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

-debugger:on|off turn Embedded Nimrod Debugger on|off

-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 (Nimrod, mangled) identifier pairs
- project** document the whole project (doc2)
- 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
- symbolFiles:on|off** turn symbol files on|off (experimental)
- implicitStatic:on|off** turn implicit compile time evaluation on|off
- patterns:on|off** turn pattern matching on|off
- 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|none** select the GC to use; default is 'refc'
- index:on|off** turn index file generation on|off
- putenv:key=value** set an environment variable
- babelPath:PATH** add a path for Babel support
- 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 Nimrod's verbosity level (1 is default)
- 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 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 nimrod executable processes configuration files in the following directories (in this order; later files overwrite previous settings):

1. \$nimrod/config/nimrod.cfg, /etc/nimrod.cfg (UNIX) or %NIMROD%/config/nimrod.cfg (Windows). This file can be skipped with the **-skipCfg** command line option.
2. /home/\$user/.config/nimrod.cfg (UNIX) or %APPDATA%/nimrod.cfg (Windows). This file can be skipped with the **-skipUserCfg** command line option.

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.
AnalysisLoophole	The thread analysis was incomplete due to an indirect call.
DifferentHeaps	The code mixes different local heaps in a very dangerous way.
WriteToForeignHeap	The code contains a threading error.
EachIdentIsTuple	The code contains a confusing <code>var</code> 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.

3. `$parentDir/nimrod.cfg` where `$parentDir` stands for any parent directory of the project file's path. These files can be skipped with the `-skipParentCfg` command line option.
4. `$projectDir/nimrod.cfg` where `$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 `$project.nimrod.cfg` that resides in the same directory as `$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:

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

2.5 Generated C code directory

The generated files that Nimrod produces all go into a subdirectory called `nimcache` in your project directory. This makes it easy to delete all generated files.

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 Cross compilation

To cross compile, use for example:

```
nimrod c --cpu:i386 --os:linux --compile_only --gen_script 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 Nimrod invoke a cross compiler toolchain:

```
nimrod 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"
```

Define	Effect
release	Turns off runtime checks and turns on the optimizer.
useWinAnsi	Modules like <code>os</code> and <code>osproc</code> use the Ansi versions of the Windows API. The default build uses the Unicode version.
useFork	Makes <code>osproc</code> use <code>fork</code> instead of <code>posix_spawn</code> .
useNimRtl	Compile and link against <code>nimrtl.dll</code> .
useMalloc	Makes Nimrod use C's <code>malloc</code> instead of Nimrod's own memory manager. This only works with <code>gc:none</code> .
useRealtimeGC	Enables support of Nimrod's GC for <i>soft</i> realtime systems. See the documentation of the <code>gc</code> for further information.
nodejs	The JS target is actually <code>node.js</code> .
ssl	Enables OpenSSL support for the <code>sockets</code> module.
memProfiler	Enables memory profiling for the native GC.

5 DLL generation

Nimrod 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 Nimrod DLL depends on `nimrtl.dll`. To generate the "nimrtl.dll" file, use the command:

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

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

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

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

6 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.

7 Additional Features

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

7.1 NoDecl pragma

The `noDecl` pragma can be applied to almost any symbol (variable, proc, type, etc.) and is sometimes useful for interoperability with C: It tells Nimrod that it should not generate a declaration for the symbol in the C code. For example:

```
var
  EACCES {.importc, noDecl.}: cint # pretend EACCES was a variable, as
                                   # Nimrod does not know its value
```

However, the header pragma is often the better alternative.

Note: This will not work for the LLVM backend.

7.2 Header pragma

The header pragma is very similar to the `noDecl` pragma: It can be applied to almost any symbol and specifies that it should not be declared and instead the generated code should contain an `#include`:

```
type
  PFile {.importc: "FILE*", header: "<stdio.h>".} = distinct pointer
  # import C's FILE* type; Nimrod will treat it as a new pointer type
```

The header pragma always expects a string constant. The string constant contains the header file: As usual for C, a system header file is enclosed in angle brackets: `<>`. If no angle brackets are given, Nimrod encloses the header file in `" "` in the generated C code.

Note: This will not work for the LLVM backend.

7.3 IncompleteStruct pragma

The `incompleteStruct` pragma tells the compiler to not use the underlying C struct in a `sizeof` expression:

```
type
  TDIR* {.importc: "DIR", header: "<dirent.h>",
        final, pure, incompleteStruct.} = object
```

7.4 Compile pragma

The compile pragma can be used to compile and link a C/C++ source file with the project:

```
{.compile: "myfile.cpp".}
```

Note: Nimrod computes a CRC checksum and only recompiles the file if it has changed. You can use the `-f` command line option to force recompilation of the file.

7.5 Link pragma

The link pragma can be used to link an additional file with the project:

```
{.link: "myfile.o".}
```

7.6 Emit pragma

The emit pragma can be used to directly affect the output of the compiler's code generator. So it makes your code unportable to other code generators/backends. Its usage is highly discouraged! However, it can be extremely useful for interfacing with C++ or Objective C code.

Example:

```
{.emit: ""static int cvariable = 420;""}.}
```

```
proc embedsC() {.noStackFrame.} =
  var nimrodVar = 89
  # use backticks to access Nimrod symbols within an emit section:
  {.emit: ""fprintf(stdout, "%d\n", cvariable + (int)`nimrodVar`);""}.}
```

```
embedsC()
```

7.7 ImportC++ pragma

The `importcpp` pragma can be used to import C++ methods. The generated code then uses the C++ method calling syntax: `obj->method(arg)`. In addition with the header and emit pragmas this allows *sloppy* interfacing with libraries written in C++:


```
# Horrible example of how to interface with a C++ engine ... ;-)

{.link: "/usr/lib/libIrrlicht.so".}

{.emit: ""using namespace irr;using namespace core;using namespace scene;using namespace video;using namespace i

const
    irr = "<irrlicht/irrlicht.h>"

type
    TIrrlichtDevice {.final, header: irr, importc: "IrrlichtDevice".} = object
    PIrrlichtDevice = ptr TIrrlichtDevice

proc createDevice(): PIrrlichtDevice {.
    header: irr, importc: "createDevice".}
proc run(device: PIrrlichtDevice): bool {.
    header: irr, importcpp: "run".}
```

The compiler needs to be told to generate C++ (command `cpp`) for this to work. The conditional symbol `cpp` is defined when the compiler emits C++ code.

7.8 ImportObjC pragma

The `importobjc` pragma can be used to import Objective C methods. The generated code then uses the Objective C method calling syntax: `[obj method param1: arg]`. In addition with the header and emit pragmas this allows *sloppy* interfacing with libraries written in Objective C:

```
# horrible example of how to interface with GNUStep ...

{.passL: "-lobjc".}
{.emit: ""#include <objc/Object.h>@interface Greeter:Object()- (void)greet:(long)x y:(long)dummy;@end#include <s

type
    TId {.importc: "id", header: "<objc/Object.h>", final.} = distinct int

proc newGreeter: TId {.importobjc: "Greeter new", nodecl.}
proc greet(self: TId, x, y: int) {.importobjc: "greet", nodecl.}
proc free(self: TId) {.importobjc: "free", nodecl.}

var g = newGreeter()
g.greet(12, 34)
g.free()
```

The compiler needs to be told to generate Objective C (command `objc`) for this to work. The conditional symbol `objc` is defined when the compiler emits Objective C code.

7.9 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.

7.10 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.

7.11 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.

7.12 Debugger option

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

7.13 Breakpoint pragma

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

7.14 Volatile pragma

The volatile pragma is for variables only. It declares the variable as `volatile`, whatever that means in C/C++ (its semantics are not well defined in C/C++).

Note: This pragma will not exist for the LLVM backend.

8 Nimrod interactive mode

The Nimrod compiler supports an interactive mode. This is also known as a REPL (*read eval print loop*). If Nimrod has been built with the `-d:useGnuReadline` switch, it uses the GNU readline library for terminal input management. To start Nimrod in interactive mode use the command `nimrod i`. To quit use the `quit()` command. To determine whether an input line is an incomplete statement to be continued these rules are used:

1. The line ends with `[-+*/\<>!\?\\|%&$@~,;:=#^]\s*$` (operator symbol followed by optional whitespace).
2. The line starts with a space (indentation).
3. The line is within a triple quoted string literal. However, the detection does not work if the line contains more than one `"""`.

9 Nimrod 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.

So to generate code for an AVR processor use this command:

```
nimrod c --cpu:avr --os:standalone --gc:none -d:useMalloc --genScript x.nim
```

10 Nimrod for realtime systems

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

11 Debugging with Nimrod

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

12 Optimizing for Nimrod

Nimrod 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. Nimrod has been designed to encourage efficient code: The most readable code in Nimrod 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.

12.1 Optimizing string handling

String assignments are sometimes expensive in Nimrod: 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))
```

13 The JavaScript target

Nimrod can also generate JavaScript code. However, the JavaScript code generator is experimental!

Nimrod targets JavaScript 1.5 which is supported by any widely used browser. Since JavaScript does not have a portable means to include another module, Nimrod just generates a long `.js` file.

Features or modules that the JavaScript platform does not support are not available. This includes:

- manual memory management (`alloc`, etc.)
- casting and other unsafe operations (`cast` operator, `zeroMem`, etc.)
- file management
- most modules of the Standard library
- proper 64 bit integer arithmetic
- unsigned integer arithmetic

However, the modules `strutils`, `math`, and `times` are available! To access the DOM, use the `dom` module that is only available for the JavaScript platform.

To compile a Nimrod module into a `.js` file use the `js` command; the default is a `.js` file that is supposed to be referenced in an `.html` file. However, you can also run the code with `nodejs`:

```
nimrod js -d:nodejs -r examples/hallo.nim
```