

Compiler

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.
Compiler 6.0.3.1
December 1, 2017

Copyright © 1997-2017 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 1, 2017

1 Reference Manual

The `Compiler` application compiles Erlang code to byte-code. The highly compact byte-code is executed by the Erlang emulator.

compile

Erlang module

This module provides an interface to the standard Erlang compiler. It can generate either a new file, which contains the object code, or return a binary, which can be loaded directly.

Exports

file(File)

Is the same as `file(File, [verbose,report_errors,report_warnings])`.

file(File, Options) -> CompRet

Types:

```
CompRet = ModRet | BinRet | ErrRet
ModRet = {ok,ModuleName} | {ok,ModuleName,Warnings}
BinRet = {ok,ModuleName,Binary} | {ok,ModuleName,Binary,Warnings}
ErrRet = error | {error,Errors,Warnings}
```

Compiles the code in the file `File`, which is an Erlang source code file without the `.erl` extension. `Options` determine the behavior of the compiler.

Returns `{ok,ModuleName}` if successful, or `error` if there are errors. An object code file is created if the compilation succeeds without errors. It is considered to be an error if the module name in the source code is not the same as the basename of the output file.

Available options:

`basic_validation`

> This option is a fast way to test whether a module will compile successfully. This is useful for code generators that want to verify the code that they emit. No code is generated. If warnings are enabled, warnings generated by the `erl_lint` module (such as warnings for unused variables and functions) are also returned.
>
> Use option `strong_validation` to generate all warnings that the compiler would generate.

`strong_validation`

> Similar to option `basic_validation`. No code is generated, but more compiler passes are run to ensure that warnings generated by the optimization passes are generated (such as clauses that will not match, or expressions that are guaranteed to fail with an exception at runtime).

`binary`

> The compiler returns the object code in a binary instead of creating an object file. If successful, the compiler returns `{ok,ModuleName,Binary}`.

`bin_opt_info`

> The compiler will emit informational warnings about binary matching optimizations (both successful and unsuccessful). For more information, see the section about *bin_opt_info* in the Efficiency Guide.

`compressed`

> The compiler will compress the generated object code, which can be useful for embedded systems.

`debug_info`

Includes debug information in the form of abstract code (see *The Abstract Format* in ERTS User's Guide) in the compiled beam module. Tools such as `Debugger`, `Xref`, and `Cover` require the debug information to be included.

Warning: Source code can be reconstructed from the debug information. Use encrypted debug information (`encrypt_debug_info`) to prevent this.

For details, see *beam_lib(3)*.

`{debug_info_key,KeyString}`
`{debug_info_key,{Mode,KeyString}}`

Includes debug information, but encrypts it so that it cannot be accessed without supplying the key. (To give option `debug_info` as well is allowed, but not necessary.) Using this option is a good way to always have the debug information available during testing, yet protecting the source code.

`Mode` is the type of crypto algorithm to be used for encrypting the debug information. The default (and currently the only) type is `des3_cbc`.

For details, see *beam_lib(3)*.

`encrypt_debug_info`

Similar to the `debug_info_key` option, but the key is read from an `.erlang.crypt` file.

For details, see *beam_lib(3)*.

`makedep`

Produces a Makefile rule to track headers dependencies. No object file is produced.

By default, this rule is written to `<File>.Pbeam`. However, if option `binary` is set, nothing is written and the rule is returned in `Binary`.

For example, if you have the following module:

```
-module(module).

-include_lib("eunit/include/eunit.hrl").
-include("header.hrl").
```

The Makefile rule generated by this option looks as follows:

```
module.beam: module.erl \
  /usr/local/lib/erlang/lib/eunit/include/eunit.hrl \
  header.hrl
```

`{makedep_output, Output}`

Writes generated rules to `Output` instead of the default `<File>.Pbeam`. `Output` can be a filename or an `io_device()`. To write to stdout, use `standard_io`. However, if `binary` is set, nothing is written to `Output` and the result is returned to the caller with `{ok, ModuleName, Binary}`.

`{makedep_target, Target}`

Changes the name of the rule emitted to `Target`.

`makedep_quote_target`

Characters in `Target` special to make(1) are quoted.

`makedep_add_missing`

Considers missing headers as generated files and adds them to the dependencies.

`makedep_phony`

Adds a phony target for each dependency.

`'P'`

Produces a listing of the parsed code, after preprocessing and parse transforms, in the file `<File>.P`. No object file is produced.

`'E'`

Produces a listing of the code, after all source code transformations have been performed, in the file `<File>.E`. No object file is produced.

`'S'`

Produces a listing of the assembler code in the file `<File>.S`. No object file is produced.

`report_errors/report_warnings`

Causes errors/warnings to be printed as they occur.

`report`

A short form for both `report_errors` and `report_warnings`.

`return_errors`

If this flag is set, `{error,ErrorList,WarningList}` is returned when there are errors.

`return_warnings`

If this flag is set, an extra field, containing `WarningList`, is added to the tuples returned on success.

`warnings_as_errors`

Causes warnings to be treated as errors. This option is supported since R13B04.

`return`

A short form for both `return_errors` and `return_warnings`.

`verbose`

Causes more verbose information from the compiler, describing what it is doing.

`{source,FileName}`

Sets the value of the source, as returned by `module_info(compile)`.

`{outdir,Dir}`

Sets a new directory for the object code. The current directory is used for output, except when a directory has been specified with this option.

`export_all`

Causes all functions in the module to be exported.

`{i,Dir}`

Adds `Dir` to the list of directories to be searched when including a file. When encountering an `-include` or `-include_lib` directive, the compiler searches for header files in the following directories:

- `"."`, the current working directory of the file server
- The base name of the compiled file
- The directories specified using option `i`; the directory specified last is searched first

`{d,Macro}`
`{d,Macro,Value}`

Defines a macro `Macro` to have the value `Value`. `Macro` is of type atom, and `Value` can be any term. The default `Value` is `true`.

`{parse_transform,Module}`

Causes the parse transformation function `Module:parse_transform/2` to be applied to the parsed code before the code is checked for errors.

`from_asm`

The input file is expected to be assembler code (default file suffix ".S"). Notice that the format of assembler files is not documented, and can change between releases.

`from_core`

The input file is expected to be core code (default file suffix ".core"). Notice that the format of core files is not documented, and can change between releases.

`no_strict_record_tests`

This option is not recommended.

By default, the generated code for operation `Record#record_tag.field` verifies that the tuple `Record` has the correct size for the record, and that the first element is the tag `record_tag`. Use this option to omit the verification code.

`no_error_module_mismatch`

Normally the compiler verifies that the module name given in the source code is the same as the base name of the output file and refuses to generate an output file if there is a mismatch. If you have a good reason (or other reason) for having a module name unrelated to the name of the output file, this option disables that verification (there will not even be a warning if there is a mismatch).

`{no_auto_import,[{F,A}, ...]}`

Makes the function `F/A` no longer being auto-imported from the `erlang` module, which resolves BIF name clashes. This option must be used to resolve name clashes with BIFs auto-imported before R14A, if it is needed to call the local function with the same name as an auto-imported BIF without module prefix.

> **Note:**
>
> As from R14A and forward, the compiler resolves calls without module prefix to local or imported functions before trying with auto-imported BIFs. If the BIF is to be called, use the `erlang` module prefix in the call, not `{ no_auto_import,[{F,A}, ...]}`.

If this option is written in the source code, as a `-compile` directive, the syntax `F/A` can be used instead of `{F,A}`, for example:

```
-compile({no_auto_import,[error/1]}).
```

`no_auto_import`

Do not auto-import any functions from `erlang` module.

`no_line_info`

Omits line number information to produce a slightly smaller output file.

If warnings are turned on (option `report_warnings` described earlier), the following options control what type of warnings that are generated. Except from `{warn_format,Verbosity}`, the following options have two forms:

- A `warn_xxx` form, to turn on the warning.
- A `nowarn_xxx` form, to turn off the warning.

In the descriptions that follow, the form that is used to change the default value are listed.

`{warn_format, Verbosity}`

Causes warnings to be emitted for malformed format strings as arguments to `io:format` and similar functions.

`Verbosity` selects the number of warnings:

- `0` = No warnings
- `1` = Warnings for invalid format strings and incorrect number of arguments
- `2` = Warnings also when the validity cannot be checked, for example, when the format string argument is a variable.

The default verbosity is `1`. Verbosity `0` can also be selected by option `nowarn_format`.

`nowarn_bif_clash`

This option is removed, it generates a fatal error if used.

> ### Warning:
>
> As from beginning with R14A, the compiler no longer calls the auto-imported BIF if the name clashes with a local or explicitly imported function, and a call without explicit module name is issued. Instead, the local or imported function is called. Still accepting `nowarn_bif_clash` would make a module calling functions clashing with auto-imported BIFs compile with both the old and new compilers, but with completely different semantics. This is why the option is removed.
>
> The use of this option has always been discouraged. As from R14A, it is an error to use it.
>
> To resolve BIF clashes, use explicit module names or the `{no_auto_import,[F/A]}` compiler directive.

`{nowarn_bif_clash, FAs}`

This option is removed, it generates a fatal error if used.

> ### Warning:
>
> The use of this option has always been discouraged. As from R14A, it is an error to use it.
>
> To resolve BIF clashes, use explicit module names or the `{no_auto_import,[F/A]}` compiler directive.

`warn_export_all`

Emits a warning if option `export_all` is also given.

`warn_export_vars`

Emits warnings for all implicitly exported variables referred to after the primitives where they were first defined. By default, the compiler only emits warnings for exported variables referred to in a pattern.

`nowarn_shadow_vars`

Turns off warnings for "fresh" variables in functional objects or list comprehensions with the same name as some already defined variable. Default is to emit warnings for such variables.

`nowarn_unused_function`

Turns off warnings for unused local functions. Default is to emit warnings for all local functions that are not called directly or indirectly by an exported function. The compiler does not include unused local functions in the generated beam file, but the warning is still useful to keep the source code cleaner.

`{nowarn_unused_function, FAs}`

Turns off warnings for unused local functions like `nowarn_unused_function` does, but only for the mentioned local functions. `FAs` is a tuple `{Name,Arity}` or a list of such tuples.

`nowarn_deprecated_function`

Turns off warnings for calls to deprecated functions. Default is to emit warnings for every call to a function known by the compiler to be deprecated. Notice that the compiler does not know about attribute `-deprecated()`, but uses an assembled list of deprecated functions in Erlang/OTP. To do a more general check, the `Xref` tool can be used. See also *xref(3)* and the function *xref:m/1*, also accessible through the function *c:xm/1*.

`{nowarn_deprecated_function, MFAs}`

Turns off warnings for calls to deprecated functions like `nowarn_deprecated_function` does, but only for the mentioned functions. `MFAs` is a tuple `{Module,Name,Arity}` or a list of such tuples.

`nowarn_deprecated_type`

Turns off warnings for use of deprecated types. Default is to emit warnings for every use of a type known by the compiler to be deprecated.

`warn_obsolete_guard`

Emits warnings for calls to old type testing BIFs, such as `pid/1` and `list/1`. See the *Erlang Reference Manual* for a complete list of type testing BIFs and their old equivalents. Default is to emit no warnings for calls to old type testing BIFs.

`warn_unused_import`

Emits warnings for unused imported functions. Default is to emit no warnings for unused imported functions.

`nowarn_unused_vars`

By default, warnings are emitted for unused variables, except for variables beginning with an underscore ("Prolog style warnings"). Use this option to turn off this kind of warnings.

`nowarn_unused_record`

Turns off warnings for unused record types. Default is to emit warnings for unused locally defined record types.

Another class of warnings is generated by the compiler during optimization and code generation. They warn about patterns that will never match (such as `a=b`), guards that always evaluate to false, and expressions that always fail (such as `atom+42`).

Those warnings cannot be disabled (except by disabling all warnings).

> **Note:**
>
> The compiler does not warn for expressions that it does not attempt to optimize. For example, the compiler tries to evaluate `1/0`, detects that it will cause an exception, and emits a warning. However, the compiler is silent about the similar expression, `X/0`, because of the variable in it. Thus, the compiler does not even try to evaluate and therefore it emits no warnings.

> **Warning:**
>
> The absence of warnings does not mean that there are no remaining errors in the code.

> **Note:**
>
> All options, except the include path (`{i,Dir}`), can also be given in the file with attribute `-compile([Option,...])`. Attribute `-compile()` is allowed after the function definitions.

> **Note:**
>
> The options `{nowarn_unused_function, FAs}`, `{nowarn_bif_clash, FAs}`, and `{nowarn_deprecated_function, MFAs}` are only recognized when given in files. They are not affected by options `warn_unused_function`, `warn_bif_clash`, or `warn_deprecated_function`.

For debugging of the compiler, or for pure curiosity, the intermediate code generated by each compiler pass can be inspected. To print a complete list of the options to produce list files, type `compile:options()` at the Erlang shell prompt. The options are printed in the order that the passes are executed. If more than one listing option is used, the one representing the earliest pass takes effect.

Unrecognized options are ignored.

Both `WarningList` and `ErrorList` have the following format:

```
[{FileName,[ErrorInfo]}].
```

`ErrorInfo` is described later in this section. The filename is included here, as the compiler uses the Erlang pre-processor `epp`, which allows the code to be included in other files. It is therefore important to know to *which* file the line number of an error or a warning refers.

`forms(Forms)`

Is the same as `forms(File, [verbose,report_errors,report_warnings])`.

`forms(Forms, Options) -> CompRet`

Types:

`Forms = [Form]`

```
CompRet = BinRet | ErrRet
BinRet = {ok,ModuleName,BinaryOrCode} |
{ok,ModuleName,BinaryOrCode,Warnings}
BinaryOrCode = binary() | term()
ErrRet = error | {error,Errors,Warnings}
```

Analogous to `file/1`, but takes a list of forms (in the Erlang abstract format representation) as first argument. Option `binary` is implicit, that is, no object code file is produced. For options that normally produce a listing file, such as 'E', the internal format for that compiler pass (an Erlang term, usually not a binary) is returned instead of a binary.

`format_error(ErrorDescriptor) -> chars()`

Types:

```
ErrorDescriptor = errordesc()
```

Uses an `ErrorDescriptor` and returns a deep list of characters that describes the error. This function is usually called implicitly when an `ErrorInfo` structure (described in section *Error Information*) is processed.

`output_generated(Options) -> true | false`

Types:

```
Options = [term()]
```

Determines whether the compiler generates a `beam` file with the given options. `true` means that a `beam` file is generated. `false` means that the compiler generates some listing file, returns a binary, or merely checks the syntax of the source code.

`noenv_file(File, Options) -> CompRet`

Works like *file/2*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

`noenv_forms(Forms, Options) -> CompRet`

Works like *forms/2*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

`noenv_output_generated(Options) -> true | false`

Types:

```
Options = [term()]
```

Works like *output_generated/1*, except that the environment variable `ERL_COMPILER_OPTIONS` is not consulted.

Default Compiler Options

The (host operating system) environment variable `ERL_COMPILER_OPTIONS` can be used to give default compiler options. Its value must be a valid Erlang term. If the value is a list, it is used as is. If it is not a list, it is put into a list.

The list is appended to any options given to *file/2*, *forms/2*, and *output_generated/2*. Use the alternative functions *noenv_file/2*, *noenv_forms/2*, or *noenv_output_generated/2* if you do not want the environment variable to be consulted, for example, if you are calling the compiler recursively from inside a parse transform.

Inlining

The compiler can do function inlining within an Erlang module. Inlining means that a call to a function is replaced with the function body with the arguments replaced with the actual values. The semantics are preserved, except if exceptions are generated in the inlined code. Exceptions are reported as occurring in the function the body was inlined into. Also, `function_clause` exceptions are converted to similar `case_clause` exceptions.

When a function is inlined, the original function is kept if it is exported (either by an explicit export or if the option `export_all` was given) or if not all calls to the function are inlined.

Inlining does not necessarily improve running time. For example, inlining can increase Beam stack use, which probably is detrimental to performance for recursive functions.

Inlining is never default. It must be explicitly enabled with a compiler option or a `-compile()` attribute in the source module.

To enable inlining, either use the option `inline` to let the compiler decide which functions to inline, or `{inline, [{Name,Arity},...]}` to have the compiler inline all calls to the given functions. If the option is given inside a `compile` directive in an Erlang module, `{Name,Arity}` can be written as `Name/Arity`.

Example of explicit inlining:

```
-compile({inline,[pi/0]}).

pi() -> 3.1416.
```

Example of implicit inlining:

```
-compile(inline).
```

The option `{inline_size,Size}` controls how large functions that are allowed to be inlined. Default is `24`, which keeps the size of the inlined code roughly the same as the un-inlined version (only relatively small functions are inlined).

Example:

```
%% Aggressive inlining - will increase code size.
-compile(inline).
-compile({inline_size,100}).
```

Inlining of List Functions

The compiler can also inline various list manipulation functions from the module `list` in `STDLIB`.

This feature must be explicitly enabled with a compiler option or a `-compile()` attribute in the source module.

To enable inlining of list functions, use option `inline_list_funcs`.

The following functions are inlined:

- *lists:all/2*
- *lists:any/2*
- *lists:foreach/2*
- *lists:map/2*
- *lists:flatmap/2*
- *lists:filter/2*
- *lists:foldl/3*
- *lists:foldr/3*
- *lists:mapfoldl/3*

- *lists:mapfoldr/3*

Parse Transformations

Parse transformations are used when a programmer wants to use Erlang syntax but with different semantics. The original Erlang code is then transformed into other Erlang code.

Error Information

The `ErrorInfo` mentioned earlier is the standard `ErrorInfo` structure, which is returned from all I/O modules. It has the following format:

```
{ErrorLine, Module, ErrorDescriptor}
```

`ErrorLine` is the atom `none` if the error does not correspond to a specific line, for example, if the source file does not exist.

A string describing the error is obtained with the following call:

```
Module:format_error(ErrorDescriptor)
```

See Also

epp(3), erl_id_trans(3), erl_lint(3), beam_lib(3)