# Compiler Application (COMPILER)

**version 4.0**

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

# Contents

1 Compiler Reference Manual 1
1.1 compile . . . . . 2

# Compiler Reference Manual

## Short Summaries


- Erlang Module **compile** [page 2] – Erlang Compiler


### compile

The following functions are exported:


- `file(File)`
  [page 2] Compile a file
- `file(File, Options) -> CompRet`
  [page 2] Compile a file
- `forms(Forms)`
  [page 3] Compile a list of forms
- `forms(Forms, Options) -> CompRet`
  [page 3] Compile a list of forms
- `format_error(ErrorDescriptor) -> string()`
  [page 4] Format an error descriptor

# 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 with no errors.

As a step in the compilation of Erlang code, `erl_lint` is run, resulting in warning and error messages, if appropriate. The options relevant to the syntactic and semantic controls of `erl_lint` are listed in the documentation of the module `erl_lint`.

The elements of `Options` can be selected as follows:

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

`debug_info` Include debug information in the compiled beam module. Currently, the only application that uses the debug information is the new `xref` tool.

Warning: Note that the source code can be reconstructed from the abstract code. Therefore, never include debug information if you want to keep the source code secret.

`'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 This is a short form for both report_errors and report_warnings.

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

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

return This is a short form for both return_errors and return_warnings.

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

{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} Add Dir to the list of directories to be searched when including a file.

{d,Macro}

{d,Macro,Value} Defines a macro Macro to have the value Value. The default 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.

asm The input file is expected to be assembler code (default file suffix ".S"). Note that the format of assembler files is not documented, and may change between releases - this option is primarily for internal debugging use.

Note that all the options except the include path can also be given in the file with a -compile([Option,...]). attribute.

For debugging of the compiler, or for pure curiosity, the intermediate code generated by each compiler pass can be inspected. A complete list of the options to produce list files can be printed by typing compile:options() at the Erlang shell prompt. The options will be printed in 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 below. The file name has been included here as the compiler uses the Erlang pre-processor epp, which allows the code to be included in other files. For this reason, it is important to know to *which* file an error or warning line number refers.

forms(Forms)

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

forms(Forms, Options) -> CompRet

Types:

- Forms = [Form]
- CompRet = ModRet | BinRet | ErrRet
- ModRet = {ok,ModuleName} | {ok,ModuleName,Warnings}
- BinRet = {ok,ModuleName,Binary} | {ok,ModuleName,Binary,Warnings}
- ErrRet = error | {error,Errors,Warnings}

Analogous to `file/1`, but takes a list of forms (in the Erlang abstract format representation) as first argument. The option `binary` is implicit; i.e., no object code file is produced. If the options indicate that a listing file should be produced (e.g., 'E'), the module name is taken as the file name.

```
format_error(ErrorDescriptor) -> string()
```

Types:

- ErrorDescriptor = errordesc()

Uses an `ErrorDescriptor` and returns a string which describes the error. This function is usually called implicitly when an `ErrorInfo` structure is processed. See below.

## 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 will be used as is. If it is not a list, it will be put into a list. The list will be appended to any options given to `file/2` or `forms/2`.

## Inlining

The compiler can now 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 will be reported as occurring in the function the body was inlined into. Also, `function_clause` exceptions will be converted to similar `case_clause` exceptions.

When a function is inlined, the original function may be kept as a separate function as well, because there might still be calls to it. Therefore, inlining almost always increases code size.

Inlining does not necessarily improve running time, especially if large functions are inlined. The increased code size may cause the code to run the slower (because of worse CPU cache performance). Also, inlining may increase Beam stack usage which will probably be 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.

There are two distinct ways to enable inlining (which may be combined).

The first way is to explicitly list the functions to be inlined at all call places. The syntax is `{inline,[{F,A},...]}`, where `F` is a function name and `A` its arity.

Example from an Erlang module:

```
-compile({inline,[{mkop,3},{mkop,2},{line,1}]}).
```

Here the functions `mkop/3`, `mkop/2`, and `line/1` will be inlined every time they are used.

This type of unconditional inlining is useful for small, simple functions as an alternative to macros. The functions mentioned in the example are defined like this:

```
mkop(L, {Op,Pos}, R) -> {op,Pos,Op,L,R}.
mkop({Op,Pos}, A) -> {op,Pos,Op,A}.
line(Tup) -> element(2, Tup).
```

There are other benefits when using explicit inlining instead of macros. The arguments will only be evaluated once, which can be critical if they contain side effects or are large computations, and it also makes it easy to have local variables, which is difficult with macros.

The other type of inlining is conditional inlining. The compiler will search for candidates suitable for inlining. It does this by calculating a weight for each function. The weight is roughly proportional to the size of the function. Given the weight for each function, the compiler will only inline functions lighter than calling function and below a given threshold value.

To enable conditional inlining, you can use the '`inline`' option, which sets a threshold value of 10, or you can explicitly give a threshold value like this: `{inline,Threshold}`.

Example:

```
-compile({inline,1000}).
```

A threshold of 1000 would inline most functions (except for extremely large), provided that the functions are lighter than the functions they are inlined into. It is not clear that this is a good idea. It all depends on your code.

> **Warning:**
> Conditional inlining should be used with caution, since it may actually increase the execution time and make debugging harder. You should only use it for modules that are known to be bottle-necks and measure execution times with and without inlining.

## 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 above is the standard `ErrorInfo` structure which is returned from all IO modules. It has the following format

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

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

```
apply(Module, format_error, ErrorDescriptor)
```

## See Also

epp, erl_id_trans, erl_lint

# Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.

*compile*
    `file/1`, 2
    `file/2`, 2
    `format_error/1`, 4
    `forms/1`, 3
    `forms/2`, 3

`file/1`
    *compile* , 2

`file/2`
    *compile* , 2

`format_error/1`
    *compile* , 4

`forms/1`
    *compile* , 3

`forms/2`
    *compile* , 3