vbcc compiler system

Volker Barthelmann

Table of Contents

1 General

1.1 Introduction

vbcc is a highly optimizing portable and retargetable ISO C compiler. It supports ISO C according to ISO/IEC 9899:1989 and a subset of the new standard ISO/IEC 9899:1999 (C99).

It is split into a target-independent and a target-dependent part, and provides complete abstraction of host- and target-arithmetic. Therefore, it fully supports cross-compiling for 8, 16, 32 and 64bit architectures.

Embedded systems are supported by features like different pointer-sizes (e.g. differently sized function- and object-pointers or near- and far-pointers), ROM-able code, inline-assembly, bit-types, interrupt-handlers, section-attributes, stack-calculation and many others (depending on the backend).

vbcc provides a large set of aggressive high-level optimizations (see Section 3.4 [Optimizations], page 15) as well as target-specific optimizations to produce faster or smaller code. Rather than restricting analysis and optimization to single functions or files, vbcc is able to optimize across functions and even modules. Target-independent optimizations include:

- cross-module function-inlining
- partial inlining of recursive functions
- inter-procedural data-flow analysis
- inter-procedural register-allocation
- register-allocation for global variables
- global common-subexpression-elimination
- global constant-propagation
- global copy-propagation
- dead-code-elimination
- alias-analysis
- loop-unrolling
- induction-variable elimination
- loop-invariant code-motion
- loop-reversal

1.2 Legal

vbcc contains the preprocessor ucpp by Thomas Pornin. Included is the copyright notice of ucpp (note that this license does not apply to vbcc or any other part of this distribution):

```
/*
 * (c) Thomas Pornin 1999, 2000
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 4. The name of the authors may not be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED ''AS IS'' AND WITHOUT ANY EXPRESS OR
 * IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
 * OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
 * OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
```

1.3 Installation

The vbcc directory tree looks as follows:

‘`vbcc/bin`’
: The executables.

‘`vbcc/config`’
: Config files for the frontend.

‘`vbcc/targets/<target>`’
: Subdirectory containing all files specific to a certain target (e.g. m68k-amigaos or ppc-eabi).

1.3.1 Installing for Unix

1. Extract the archive.
2. Set the environment variable `VBCC` to the vbcc directory. Depending on your shell this might be done e.g. by

```
VBCC=<prefix>/vbcc
```

or

```
setenv VBCC <prefix>/vbcc
```

3. Include <prefix>/vbcc/bin to your search-path. Depending on your shell this might be done e.g. by

```
PATH=<prefix>/vbcc/bin:"$PATH"
```

or

```
setenv PATH <prefix>/vbcc/bin:"$PATH"
```

1.3.2 Installing for DOS/Windows

1. Extract the archive.
2. Set the environment variable `VBCC` to the vbcc directory.

```
set VBCC=<prefix>\vbcc
```

3. Include <prefix>/vbcc/bin to your search-path.

```
set PATH=<prefix>\vbcc\bin;%PATH%
```

1.3.3 Installing for AmigaOS

To use `vbcc` on AmigaOS, several assigns have to be set (e.g. in '`s:user-startup`'):

```
assign >NIL: vbcc: <path to vbcc directory>
assign >NIL: C: vbcc:bin add

assign >NIL: vbccm68k: vbcc:targets/m68k-amigaos
assign >NIL: vincludem68k: vbccm68k:include
assign >NIL: vincludem68k: <path to your AmigaOS header files> ADD
;assign >NIL: ixinclude: <path to ixemul header files, if needed>
assign >NIL: vlibm68k: vbccm68k:lib

assign >NIL: vbccppc: vbcc:targets/ppc-powerup
assign >NIL: vincludeppc: vbccppc:include
assign >NIL: vlibppc: vbccppc:lib
assign >NIL: vincludeppc: <path to your AmigaOS header files> ADD

assign >NIL: vbccwos: vbcc:targets/ppc-warpos
assign >NIL: vincludewos: vbccwos:include
assign >NIL: vlibwos: vbccwos:lib
assign >NIL: vincludewos: <path to your AmigaOS header files> ADD

assign >NIL: vbccmos: vbcc:targets/ppc-morphos
assign >NIL: vincludemos: vbccmos:include
```

```
assign >NIL: vlibmos: vbccmos:lib
assign >NIL: vincludemos: <path to your MorphOS header files> ADD
```

Also, the stack-size has to be increased from the default. 40KB is a sensible value, for very large projects higher values might be necessary.

There is a sample script file '`init_vbcc`'. Changing to the vbcc-directory and executing this script will set up a basic vbcc system. However, it is recommended to adapt the script and put it into your '`s:user-startup`'.

There are different configuration files provided in the '`config`'-subdirectory to choose different targets (i.e. the system you want to generate programs for) and hosts (i.e. the system you want the compiler an tools to run on). The general naming-scheme for these files ist <target>_<host>.

The systems available as targets are '`m68k`' (AmigaOS on 68k with standard libraries), '`ixemul`' (AmigaOS on 68k using ixemul library), '`ppc`' (PPC boards using the PowerUp system), '`warpos`' (PPC boards using the WarpOS system) and '`morphos`' (PPC systems running MorphOS).

'`m68k`', '`ppc`' and '`warpos`' are available as host specifiers on AmigaOS.

You can choose one of these systems using the '`+`'-option of `vc`, e.g.

```
vc +m68k_ppc ...
```

will compile for AmigaOS/68k using the compiler running on PowerUp.

You may choose to create copies of some of these configuration files with simpler names. E.g. if you usually want the compiler to run on WarpOS you could copy '`m68k_warpos`' to '`m68k`', '`warpos_warpos`' to '`warpos`' and so on. Then you can just specify the target and your preferred host system will be chosen automatically.

Additionally, you may copy the configuration file for your preferred host/target-combination to '`vc.config`'. This configuration will be chosen by default if you do not specify anything.

By default, the target-only-specifications use 68k-native tools, e.g. '`+warpos`' will create code for WarpOS, but the compiler and tools will run on the 68k. The default '`vc.config`' will create code for 68k using tools running on 68k.

1.4 Tutorial

Now you should be able to use vbcc. To compile and link the program '`hello.c`', type

```
vc hello.c
```

The file '`hello.c`' will be compiled and linked to create the executable a.out in the current directory.

```
vc hello.c -o hello
```

will do the same, but the created executable will be called '`hello`'.

```
vc -c t1.c t2.c
```

will compile '`t1.c`' and '`t2.c`' without linking, creating the object files '`t1.o`' and '`t2.o`'.

```
vc t1.o t2.o -o tt
```

will link them together and create the executable '`tt`'.

If your program uses floating point, you may have to link with a math-library. The details are dependent on the target, but usually '`-lm`' will be suitable (for AmigaOS on m68k choose one of '`-lmieee`', '`-lm881`' or '`-lm040`').

```
vc calc.c -o calc -lmieee
```

There may also be an extra.lib which includes a few functions that are no standard C functions but some people seem to regard them as standard functions. If you use one of these add '`-lextra`' to the commandline.

2 The Frontend

This chapter describes `vc`, the frontend for vbcc. It knows how to deal with different file types and optimization settings and will call the compiler, assembler and linker. It is not recommended to call the different translation-phases directly. `vc` provides an easy-to-use interface which is mostly compatible to Unix `cc`.

2.1 Usage

The general syntax for calling `vc`

```
vc [options] file1 file2 ...
```

processes all files according to their suffix and links all objects together (unless any of '`-E`', '`-S`', '`-c`' is specified). The following file types are recognized:

'`.c`' C source

'`.i`' already preprocessed C source

'`.scs`' assembly source to be fed to the scheduler

'`.asm`'

'`.s`' assembly source

'`.obj`'

'`.o`' object file

Usually pattern matching is supported - however this depends on the port and the host system.

The options recognized by `vc` are:

'`-v`' Verbose mode. Prints all commands before executing them.

'`-vv`' Very verbose. Displays some internals as well.

'`-Ox`' Sets the optimization level.
-O0 is equivalent to -O=0.
-O will activate some optimizations (at the moment -O=991).
-O2 will activate most optimizations (at the moment -O=1023 -schedule).
-O3 will activate all optimizations (at the moment -O=~0 -schedule).
-O4 will activate full cross-module-optimization.

Also, -O3 will activate cross-module-optimizations. All source files specified on the command line will be passed to the compiler at once. Only one assembly/object-file will be produced (by default the name is the name of the first source file with corresponding suffix).

When compiling with -O4 and -c vbcc will not produce real object files but special files containing all necessary information to defer optimization and code-generation to link-time. This is useful to provide all files of a project to the optimizer and make full use of cross-module optimizations. Note that you must

use vc to do the linking. vc will detect and handle these files correctly. They can not be linked directly. Also, make sure to pass all relevant compiler options also to the linker-command.

Higher values may or may not activate even more optimizations. The default is -O=1. It is also possible to specify an exact value with -O=n. However, I do not recommend this unless you know exactly what you are doing.

'`-o file`' Save the target as '`file`' (default for executables is '`a.out`').

'`-E`' Save the preprocessed C sources with .i suffix.

'`-S`' Do not assemble. Save the compiled files with .asm suffix.

'`-SCS`' Do not schedule. Save the compiled files with .scs suffix.

'`-c`' Do not link. Save the compiled files with .o suffix.

'`-k`' Keep all intermediate files. By default all generated files except the source files and the targets are deleted.

'`-Dstr`' `#define` a preprocessor symbol, e.g. -DAMIGA or -DCPU=68000. The former syntax is equivalent to:

```
#define AMIGA 1
```

The latter form is equivalent to:

```
#define CPU 68000
```

'`-Ipath`' Add '`path`' to the include-search-path.

'`-lulib`' Link with library '`ulib`'.

'`-Lpath`' Add '`path`' to the library-search-path. This is passed through to the linker.

'`-nostdlib`'
Do not link with standard-startup/libraries. Useful only for people who know what they are doing.

'`-notmpfile`'
Do not use names from tmpnam() for temporary files.

'`-schedule`'
Invoke the instruction-scheduler, if available.

'`+file`' Use '`file`' as config-file.

All other options are passed through to `vbcc`.

2.2 Configuration

`vc` needs a config file to know how to call all the translation phases (compiler, assembler, linker). Unless a different file is specified using the '`+`'-option, it will look for a file '`vc.config`' ('`vc.cfg`' for DOS/Windows).

On AmigaOS `vc` will search in the current directory, in '`ENV:`' and '`VBCC:`'.

On Unix `vc` will search in the current directory followed by '`/etc/`'.

On DOS/Windows it will search in the current directory.

If the config file was not found in the default search-paths and an environment variable `$VBCC` is set, `vc` will also look in `$VBCC'/config'`.

Once a config file is found, it will be treated as a collection of additional command line arguments. Every line of the file will be used as one argument. So no quoting shall be used and furthermore must each argument be placed on its own line.

The following options can be used to tell `vc` how to call the translation phases (they will usually be contained in the config-file):

'`-pp=string`'
: The preprocessor will be called like in `printf(string,opts,infile,outfile)`, e.g. the default for `vcpp` searching the includes in '`vinclude:`' and defining `__STDC__`) is '`-pp=vcpp -Ivinclude: -D__STDC__=1 %s %s %s`'

'`-cc=string`'
: For the compiler. Note that you cannot use `vc` to call another compiler than `vbcc`. But you can call different versions of `vbcc` this way, e.g.: '`-cc=vbcca68k -quiet`' or '`-cc=vbcci386 -quiet`'

'`-isc=string`'
: The same for the scheduler, e.g.: '`-isc=vscppc -quiet %s %s`' Omit, if there is no scheduler for the architecture.

'`-as=string`'
: The same for the assembler, e.g.: '`-as=PhxAss opt NRQBTLPSM quiet %s to %s`' or '`-as=as %s -o %s`'

'`-rm=string`'
: This is the string for the delete command and takes only one argument, e.g. '`-rm=delete quiet %s`' or '`-rm=rm %s`'

'`-ld=string`'
: This is for the linker and takes three arguments. The first one are the object files (separated by spaces), the second one the user specified libraries and the last one the name of the resulting executable. This has to link with proper startup-code and c-libraries, e.g.: '`-ld=PhxLnk vlib:startup.o %s %s vlib:vc.lib vlib:amiga.lib to %s`' or '`-ld=ld /usr/lib/crt0.o %s %s -lc -o %s`'

'`-l2=string`'
: The same like -ld, but standard-startup and -libraries should not be linked; used when -nostdlib is specified.

All those strings should tell the command to omit any output apart from error messages if possible. However for every of those options there exists one with an additional '`v`', i.e. '`-ppv=`', '`-asv=`', etc. which should produce some output, if possible. If vc is invoked with the '`-vv`' option the verbose commands will be called, if not the quiet ones will be used.

'`-ul=string`'
: Format for additional libraries specified with '`-l<lib>`'. The result of `printf(string,lib)` will be added to the command invoking the linker. Examples are: '`-ul=vlib:%s.lib`' or '`-ul=-l%s`'

3 The Compiler

This chapter describes the target-independent part of the compiler. It documents the options and extensions which are not specific to a certain target. Be sure to also read the chapter on the backend you are using. It will likely contain important additional information like data-representation or additional options.

3.1 General Compiler Options

Usually `vbcc` will be called by `vc`. However, if called directly it expects the following syntax:

```
vbcc<target> [options] file
```

The following options are supported by the machine independent part of `vbcc` (and will be passed through by `vc`):

'`-quiet`' Do not print the copyright notice.

'`-ic1`' Write the intermediate code before optimizing to file.ic1.

'`-ic2`' Write the intermediate code after optimizing to file.ic2.

'`-debug=n`'
Set the debug level to n.

'`-o=ofile`'
Write the generated assembler output to <ofile> rather than the default file.

'`-noasm`' Do not generate assembler output (only for testing).

'`-O=n`' Turns optimizing options on/off; every bit set in n turns on an option. See Section 3.4 [Optimizations], page 15.

'`-speed`' Turns on optimizations which improve speed even if they increase code-size quite a bit.

'`-size`' Turns on optimizations which improve code-size even if they have negative effect on execution-times.

'`-final`' This flag is useful only with higher optimization levels. It tells the compiler that all relevant files have been provided to the compiler (i.e. it is the link-stage). The compiler will try to eliminate all functions and variables which are not referenced.

See Section 3.4.13 [Unused Object Elimination], page 27.

'`-wpo`' Create a high-level pseudo object for cross-module optimization (see Section 3.4.16 [Cross-Module Optimizations], page 29).

'`-g`' Create debug output. Whether this is supported as well as the format of the debug information depends on the backend. Some backends may offer additional options to control the generation of debug output.

Usually DWARF2-output will be generated by default, if possible.

Also, options regarding optimization and code-generation may affect the debug output (see Section 3.4.19 [Debugging Optimized Code], page 31).

'`-c99`' Switch to the 1999 ISO standard for C /ISO/IEC9899:1999). Currently the following changes of C99 are handled:

- long long int (not supported by all backends)
- flexible array members as last element of a struct
- mixed statements and declarations
- declarations within for-loops
- `inline` function-specifier
- `restrict`-qualifier
- new reserved keywords
- `//`-comments
- vararg-macros
- `_Pragma`
- implicit int deprecated
- implicit function-declarations deprecated
- increased translation-limits

'`-maxoptpasses=n`'
Set maximum number of optimizer passes to n. See Section 3.4 [Optimizations], page 15.

'`-inline-size=n`'
Set the maximum 'size' of functions to be inlined. See Section 3.4.11 [Function Inlining], page 25.

'`-inline-depth=n`'
Inline functions up to n nesting-levels (including recursive calls). The default value is 1. Be careful with values greater than 2. See Section 3.4.11 [Function Inlining], page 25.

'`-unroll-size=n`'
Set the maximum 'size' of unrolled loops. See Section 3.4.10 [Loop Unrolling], page 23.

'`-unroll-all`'
Unroll loops with a non-constant number of iterations if the number can be calculated at runtime before entering the loop. See Section 3.4.10 [Loop Unrolling], page 23.

'`-no-inline-peephole`'
Some backends provide peephole-optimizers which perform simple optimizations on the assembly code output by `vbcc`. By default, these optimizations will also be performed on inline-assembly code of the application. This switch turns off this behaviour. See Section 3.5.3 [Inline-Assembly Functions], page 33.

'`-fp-associative`'
: Floating point operations do not obey the law of associativity, e.g. `(a+b)+c==a+(b+c)` is not true for all floating point numbers `a`,`b`,`c`. Therefore certain optimizations depending on this property cannot be performed on floating point numbers.

  This option tells `vbcc` to treat floating point operations as associative and perform those optimizations even if that may change the results in some cases (not ISO conforming).

'`-no-alias-opt`'
: Do not perform type-based alias analysis. See Section 3.4.14 [Alias Analysis], page 27.

'`-no-multiple-ccs`'
: If the backend supports multiple condition code registers, `vbcc` will try to use them when optimizing. This flag prevents `vbcc` from using them.

'`-double-push`'
: On targets where function-arguments are passed in registers but also stack-slots are left empty for such arguments, pass those arguments both in registers and on the stack.

  This generates less efficient code but some broken code (e.g. code which calls varargs functions without correct prototypes in scope) may work.

'`-stack-check`'
: Insert code for dynamic stack checking/extending if the backend and the environment support this feature.

'`-ansi`'
'`-iso`'
: Switch to ANSI/ISO mode.
  - In ISO mode warning 209 will be printed by default.
  - Inline-assembly functions are not recognized.
  - Assignments between pointers to <type> and pointers to unsigned <type> will cause warnings.

'`-maxerrors=n`'
: Abort the compilation after n errors; do not stop if n==0.

'`-dontwarn=n`'
: Suppress warning number n; suppress all warnings if n<0. See Section 3.2 [Errors and Warnings], page 14

'`-warn=n`'
: Turn on warning number n; turn on all warnings if n<0. See Section 3.2 [Errors and Warnings], page 14

'`-strip-path`'
: Strip the path of filenames from error messages. Error messages may look more convenient that way, but message browsers or similar programs might need full paths.

'`-+`'
'`-cpp-comments`'
: Allow C++ style comments (not ISO89 conforming).

'`-no-trigraphs`'
: Do not recognize trigraphs (not ISO conforming).

'`-E`'
: Write the preprocessor output to <file>.i.

'`-dontkeep-initialized-data`'
: By default `vbcc` keeps all data of initializations in memory during the whole compilation (it can sometimes make use of this when optimizing). This can take some amount of memory, though. This options tells `vbcc` to keep as little of this data in memory as possible. This has not yet been tested very well.

The assembler output will be saved to '`file.asm`' (if '`file`' already contained a suffix, this will first be removed; same applies to .ic1/.ic2)

3.2 Errors and Warnings

`vbcc` knows the following kinds of messages:

Fatal Errors
: Something is badly wrong and further compilation is impossible or pointless. `vbcc` will abort. E.g. no source file or really corrupt source.

Errors
: There was an error and `vbcc` cannot generate useful code. Compilation continues, but no code will be generated. E.g. unknown identifiers.

Warnings (1)
: Warnings with ISO-violations. The program is not ISO-conforming, but `vbcc` will generate code that could be what you want (or not). E.g. missing semicolon.

Warnings (2)
: The code has no ISO-violations, but contains some strange things you should perhaps look at. E.g. unused variables.

Errors or the first kind of warnings are always displayed and cannot be suppressed.

Only some warnings of the second kind are turned on by default. Many of them are very useful for some but annoying to others, and their usability may depend on programming style. Everybody is recommended to find their own preferences.

A good way to do this is starting with all warnings turned on by '`-warn=-1`'. Now all possible warnings will be issued. Everytime a warning that is not considered useful appears, turn that one off with '`-dontwarn=n`'.

See Chapter 8 [List of Errors], page 63 for a list of all diagnostic messages available.

See Chapter 2 [The Frontend], page 7 to find out how to configure `vc` to your preferences.

3.3 Data Types

`vbcc` can handle the following atomic data types:

`signed char`

`unsigned char`

`signed short`

`unsigned short`

`signed int`

`unsigned int`

`signed long int`

`unsigned long int`

`signed long long int`
(with '`-c99`')

`unsigned long long int`
(with '`-c99`')

`float`

`double`

`long double`

The default signedness for integer types is `signed`.

Depending on the backend, some of these types can have identical representation. The representation (size, alignment etc.) of these types usually varies between different backends. `vbcc` is able to support arbitrary implementations.

Backends may be restricted and omit some types (e.g. floating point on small embedded architectures) or offer additional types. E.g. some backends may provide special bit types or different pointer types.

3.4 Optimizations

`vbcc` offers different levels of optimization, ranging from fast compilation with straightforward code suitable for easy debugging to highly aggressive cross-module optimizations delivering very fast and/or tight code.

This section describes the general phases of compilation and gives a short overview on the available optimizations.

In the first compilation phase every function is parsed into a tree structure one expression after the other. Type-checking and some minor optimizations like constant-folding or some algebraic simplifications are done on the trees. This phase of the translation is identical in optimizing and non-optimizing compilation.

Then intermediate code is generated from the trees. In non-optimizing compilation temporaries needed to evaluate the expression are immediately assigned to registers while in optimizing compilation, a new variable is generated for each temporary. Slightly different intermediate code is produced in optimizing compilation. Some minor optimizations are performed while generating the intermediate code (simple elimination of unreachable code, some optimizations on branches etc.).

After intermediate code for the whole function has been generated, simple register allocation may be done in non-optimizing compilation if bit 1 has been set in the '`-O`' option.

Afterwards, the intermediate code is passed to the code generator and then all memory for the function, its variables etc. is freed.

In optimizing compilation flowgraphs are constructed, data flow analysis is performed and many passes are made over the function's intermediate code. Code may be moved around, new variables may be added, other variables removed etc. etc. (for more detailed information on the optimizations look at the description for the '`-O`' option below).

Many of the optimization routines depend on each other. If one routine finds an optimization, this often enables other routines to find further ones. Also, some routines only do a first step and let other routines 'clean up' afterwards. Therefore `vbcc` usually makes many passes until no further optimizations are found. To avoid possible extremely long optimization times, the number of those passes can be limited with '`-maxoptpasses`' (the default is max. 10 passes). `vbcc` will display a warning if more passes might be useful.

Depending on the optimization level, a whole translation-unit or even several translation-units will be read at once. Also, the intermediate code for all functions may be kept in memory during the entire compilation. Be aware that higher optimization levels can take much more time and memory to complete.

The following table lists the optimizations which are activated by bits in the argument of the '`-O`' option. Note that not all combinations are valid. It is heavily recommended not to fiddle with this option but just use one of the settings provided by `vc` (e.g. '`-O0`' - '`-O4`'). These options also automatically handle actions like invoking the scheduler or cross-module optimizer.

Bit 0 (1) Perform Register allocation. See Section 3.4.1 [Register Allocation], page 17.

Bit 1 (2) This flag turns on the optimizer. If it is set to zero, no global optimizations will be performed, no matter what the other flags are set to. Slightly different intermediate code will be generated by the first translation phases and a flowgraph will be constructed. See Section 3.4.2 [Flow Optimizations], page 18.

Bit 2 (4) Perform common subexpression elimination (see Section 3.4.3 [Common Subexpression Elimination], page 19) and copy propagation (see Section 3.4.4 [Copy Propagation], page 19). This can be done globally or only within basic blocks depending on bit 5.

Bit 3 (8) Perform constant propagation (see Section 3.4.5 [Constant Propagation], page 20). This can be done globally or only within basic blocks depending on bit 5.

Bit 4 (16) Perform dead code elimination (see Section 3.4.6 [Dead Code Elimination], page 20).

Bit 5 (32) Some optimizations are available in local and global versions. This flag turns on the global versions. Several major optimizations will not be performed and only one optimization pass is done unless this flag is set.

Bit 6 (64) Reserved.

Bit 7 (128)
`vbcc` will try to identify loops and perform some loop optimizations. See Section 3.4.8 [Strength Reduction], page 21 and Section 3.4.7 [Loop-Invariant Code Motion], page 21. These only work if bit 5 (32) is set.

Bit 8 (256)
: `vbcc` tries to place variables at the same memory addresses if possible (see Section 3.4.13 [Unused Object Elimination], page 27).

Bit 9 (512)
: Reserved.

Bit 10 (1024)
: Pointers are analyzed and more precise alias-information is generated (see Section 3.4.14 [Alias Analysis], page 27). Using this information, better data-flow analysis is possible.

  Also, `vbcc` tries to place global/static variables and variables which have their address taken in registers, if possible (see Section 3.4.1 [Register Allocation], page 17).

Bit 11 (2048)
: More aggressive loop optimizations are performed (see Section 3.4.10 [Loop Unrolling], page 23 and Section 3.4.9 [Induction Variable Elimination], page 22). Only works if bit 5 (32) and bit 7 (128) are set.

Bit 12 (4096)
: Perform function inlining (see Section 3.4.11 [Function Inlining], page 25).

Bit 13 (8192)
: Reserved.

Bit 14 (16384)
: Perform inter-procedural analysis (see Section 3.4.15 [Inter-Procedural Analysis], page 29) and cross-module optimizations (see Section 3.4.16 [Cross-Module Optimizations], page 29).

Also look at the documentation for the target-dependent part of `vbcc`. There may be additional machine specific optimization options.

3.4.1 Register Allocation

This optimization tries to assign variables or temporaries into machine registers to save time and space. The scope and details of this optimization vary on the optimization level.

With '`-O0`' only temporaries during expression-evaluation are put into registers. This may be useful for debugging.

At the default level (without the optimizer), additionally local variables whose address has not been taken may be put into registers for a whole function. The decision which variables to assign to registers is based on very simple heuristics.

In optimizing compilation a different algorithm will be used which uses hierarchical live-range-splitting. This means that variables may be assigned to different registers at different time. This typically allows to put the most used variables into registers in all inner loops. Note that this means that a variable can be located in different registers at different locations. Most debuggers can not handle this.

Also, the use of registers can be guided by information provided by the backend, if available. For architectures which are not very orthogonal this allows to choose registers which are better suited to certain operations. Constants can also be assigned to registers, if this is beneficial for the architecture.

The options '`-speed`' and '`-size`' change the behaviour of the register-allocator to optimize for speed or size of the generated code.

On low optimization levels, only local variables whose address has not been taken will be assigned to registers. On higher optimization levels, `vbcc` will also try to assign global/static variables and variables which had their address taken, to registers. Typically, this occurs during loops. The variables will be loaded into a register before entering a loop and stored back after the loop. However, this can only be done if `vbcc` can detect that the variable is not modified in unpredictable ways. Therefore, alias-analysis is crucial for this optimization.

During register-allocation `vbcc` will use information on register usage of functions to minimize loading/saving of registers between function-calls. Therefore, other optimizations will affect register allocation. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.2 Flow Optimizations

When optimizing `vbcc` will construct a flowgraph for every function and perform optimizations based on control-flow. For example, code which is unreachable will be removed and branches to other branches or branches around branches will be simplified.

Also, unused labels will be removed and basic blocks united to allow further optimizations.

For example, the following code

```
void f(int x, int y)
{
  if(x > y)
    goto label1;
  q();
label1:
  goto label2;
  r();
label2:
}
```

will be optimized like:

```
void f(int x, int y)
{
  if(x <= y)
    q();
}
```

Identical code at the beginning or end of basic blocks will be moved to the successors/predecessors under certain conditions.

3.4.3 Common Subexpression Elimination

If an expression has been computed on all paths leading to a second evaluation and `vbcc` knows that the operands have not been changed, then the result of the original evaluation will be reused instead of recomputing it. Also, memory operands will be loaded into registers and reused instead of being reloaded, if possible.

For example, the following code

```
void f(int x, int y)
{
  q(x * y, x * y);
}
```

will be optimized like:

```
void f(int x, int y)
{
  int tmp;

  tmp = x * y;
  q(tmp, tmp);
}
```

Depending on the optimization level, `vbcc` will perform this optimization only locally within basic blocks or globally across an entire function.

As this optimization requires detecting whether operand of an expression may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.4 Copy Propagation

If a variable is assigned to another one, the original variable will be used as long as it is not modified. This is especially useful in conjunction with other optimizations, e.g. common subexpression elimination.

For example, the following code

```
int y;

int f()
{
  int x;
  x = y;
  return x;
}
```

will be optimized like:

```
int y;

int f()
```

```
{
  return y;
}
```

Depending on the optimization level, `vbcc` will perform this optimization only locally within basic blocks or globally across an entire function.

As this optimization requires detecting whether a variable may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.5 Constant Propagation

If a variable is known to have a constant value (this includes addresses of objects) at some use, it will be replaced by the constant.

For example, the following code

```
int f()
{
  int x;
  x = 1;
  return x;
}
```

will be optimized like:

```
int f()
{
  return 1;
}
```

Depending on the optimization level, `vbcc` will perform this optimization only locally within basic blocks or globally across an entire function.

As this optimization requires detecting whether a variable may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.6 Dead Code Elimination

If a variable is assigned a value which is never used (either because it is overwritten or its lifetime ends), the assignment will be removed. This optimization is crucial to remove code which has become dead due to other optimizations.

For example, the following code

```
int x;

void f()
{
  int y;
```

```
  x = 1;
  y = 2;
  x = 3;
}
```

will be optimized like:

```
int x;

void f()
{
  x = 3;
}
```

As this optimization requires detecting whether a variable may be read, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.7 Loop-Invariant Code Motion

If the operands of a computation within a loop will not change during iterations, the computation will be moved outside of the loop.

For example, the following code

```
void f(int x, int y)
{
  int i;

  for (i = 0; i < 100; i++)
    q(x * y);
}
```

will be optimized like:

```
void f(int x, int y)
{
  int i, tmp = x * y;

  for (i = 0; i < 100; i++)
    q(tmp);
}
```

As this optimization requires detecting whether operands of an expression may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.8 Strength Reduction

This is an optimization applied to loops in order to replace more costly operations (usually multiplications) by cheaper ones (typically additions). Linear functions of an induction

variable (a variable which is changed by a loop-invariant value in every iteration) will be replaced by new induction variables. If possible, the original induction variable will be eliminated.

As array accesses are actually composed of multiplications and additions, they often benefit significantly by this optimization.

For example, the following code

```
void f(int *p)
{
  int i;

  for (i = 0; i < 100; i++)
    p[i] = i;
}
```

will be optimized like:

```
void f(int *p)
{
  int i;

  for (i = 0; i < 100; i++)
    *p++ = i;
}
```

As this optimization requires detecting whether operands of an expression may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.9 Induction Variable Elimination

If an induction variable is only used to determine the number of iterations through the loop, it will be removed. Instead, a new variable will be created which counts down to zero. This is generally faster and often enables special decrement-and-branch or decrement-and-compare instructions.

For example, the following code

```
void f(int n)
{
  int i;

  for (i = 0; i < n; i++)
    puts("hello");
}
```

will be optimized like:

```
void f(int n)
{
  int tmp;
```

```
    for(tmp = n; tmp > 0; tmp--)
      puts("hello");

}
```

As this optimization requires detecting whether operands of an expression may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.10 Loop Unrolling

`vbcc` reduces the loop overhead by replicating the loop body and reducing the number of iterations. Also, additional optimizations between different iterations of the loop will often be enabled by creating larger basic blocks. However, code-size as well as compilation-times can increase significantly.

This optimization can be controlled by '`-unroll-size`' and '`-unroll-all`'. '`-unroll-size`' specifies the maximum number of intermediate instructions for the unrolled loop body. `vbcc` will try to unroll the loop as many times to suit this value.

If the number of iterations is constant and the size of the loop body multiplied by this number is less or equal to the value specified by '`-unroll-size`', the loop will be unrolled completely. If the loop is known to be executed exactly once, it will always be unrolled completely.

For example, the following code

```
void f()
{
  int i;

  for (i = 0; i < 4; i++)
    q(i);
}
```

will be optimized like:

```
void f()
{
  q(0);
  q(1);
  q(2);
  q(3);
}
```

If the number of iteration is constant the loop will be unrolled as many times as permitted by the size of the loop and '`-unroll-size`'. If the number of iterations is not a multiple of the number of replications, the remaining iterations will be unrolled separately.

For example, the following code

```
void f()
{
  int i;
```

```
  for (i = 0; i < 102; i++)
    q(i);
}
```

will be optimized like:

```
void f()
{
  int i;
  q(0);
  q(1);
  for(i = 2; i < 102;){
    q(i++);
    q(i++);
    q(i++);
    q(i++);
  }
}
```

By default, only loops with a constant number of iterations will be unrolled. However, if '`-unroll-all`' is specified, `vbcc` will also unroll loops if the number of iterations can be calculated at entry to the loop.

For example, the following code

```
void f(int n)
{
  int i;

  for (i = 0; i < n; i++)
    q(i);
}
```

will be optimized like:

```
void f(int n)
{
  int i, tmp;

  i = 0;
  tmp = n & 3;
  switch(tmp){
  case 3:
    q(i++);
  case 2:
    q(i++);
  case 1:
    q(i++);
  }
  while(i < n){
    q(i++);
    q(i++);
    q(i++);
```

```
    q(i++);
  }
}
```

As this optimization requires detecting whether operands of an expression may have changed, it will be affected by other optimizations. See Section 3.4.14 [Alias Analysis], page 27, Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.11 Function Inlining

To reduce the overhead, a function call can be expanded inline. Passing parameters can be optimized as the arguments can be directly accessed by the inlined function. Also, further optimizations are enabled, e.g. constant arguments can be evaluated or common subexpressions between the caller and the callee can be eliminated. An inlined function call is as fast as a macro. However (just as with using large macros), code size and compilation time can increase significantly.

Therefore, this optimization can be controlled with '`-inline-size`' and '`-inline-depth`'. `vbcc` will only inline functions which contain less intermediate instructions than specified with this option.

For example, the following code

```
int f(int n)
{
  return q(&n,1);
}

void q(int *x, int y)
{
  if(y > 0)
    *x = *x + y;
  else
    abort();
}
```

will be optimized like:

```
int f(int n)
{
  return n + 1;
}

void q(int *x, int y)
{
  if(y > 0)
    *x = *x + y;
  else
    abort();
}
```

If a function to be inlined calls another function, that function can also be inlined. This also includes a recursive call of the function.

For example, the following code

```
int f(int n)
{
  if(n < 2)
    return 1;
  else
    return f(n - 1) + f(n - 2);
}
```

will be optimized like:

```
int f(int n)
{
  if(n < 2)
    return 1;
  else{
    int tmp1 = n - 1, tmp2, tmp3 = n - 2, tmp4;
    if(tmp1 < 2)
      tmp2 = 1;
    else
      tmp2 = f(tmp1 - 1) + f(tmp2 - 2);
    if(tmp3 < 2)
      tmp4 = 1;
    else
      tmp4 = f(tmp3 - 1) + f(tmp3 - 2);
    return tmp2 + tmp4;
  }
}
```

By default, only one level of inlining is done. The maximum nesting of inlining can be set with '`-inline-depth`'. However, this option should be used with care. The code-size can increase very fast and in many cases the code will be slower. Only use it for fine-tuning after measuring if it is really beneficial.

At lower optimization levels a function must be defined in the same translation-unit as the caller to be inlined. With cross-module optimizations, `vbcc` will also inline functions which are defined in other files. See Section 3.4.16 [Cross-Module Optimizations], page 29.

See also Section 3.5.3 [Inline-Assembly Functions], page 33.

3.4.12 Intrinsic Functions

This optimization will replace calls to some known functions (usually library functions) with calls to different functions or special inline-code. This optimization usually depends on the arguments to a function. Typical candidates are the `printf` family of functions and string-functions applied to string-literals.

For example, the following code

```
int f()
```

```
{
  return strlen("vbcc");
}
```

will be optimized like:

```
int f()
{
  return 4;
}
```

Note that there are also other possibilities of providing specially optimized library functions. See Section 3.5.3 [Inline-Assembly Functions], page 33 and Section 3.4.11 [Function Inlining], page 25.

3.4.13 Unused Object Elimination

Depending on the optimization level, `vbcc` will try to eliminate different objects and reduce the size needed for objects.

Generally, `vbcc` will try to use common storage for local non-static variables with non-overlapping live-ranges .

At some optimization levels and with '`-size`' specified, `vbcc` will try to order the placement of variables with static storage-duration to minimize padding needed due to different alignment requirements. This optimization generally benefits from an increased scope of optimization. See Section 3.4.16 [Cross-Module Optimizations], page 29.

At higher optimization levels objects and functions which are not referenced are eliminated. This includes functions which have always been inlined or variables which have always been replaced by constants.

When using separate compilation, objects and functions with external linkage usually cannot be eliminated, because they might be referenced from other translation-units. This precludes also elimination of anything referenced by such an object or function.

However, unused objects and functions with external linkage can be eliminated if '`-final`' is specified. In this case `vbcc` will assume that basically the entire program is presented and eliminate everything which is not referenced directly or indirectly from main(). If some objects are not referenced but must not be eliminated, they have to be declared with the `__entry` attribute. Typical examples are callback functions which are called from a library function or from anywhere outside the program, interrupt-handlers or other data which should be preserved. See Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.14 Alias Analysis

Many optimizations can only be done if it is known that two expressions are not aliased, i.e. they do not refer to the same object. If such information is not available, worst-case assumptions have to be made in order to create correct code. In the C language aliasing can occur by use of pointers. As pointers are generally a very frequently used feature of C and also array accesses are just disguised pointer arithmetic, alias analysis is very important.

`vbcc` uses the following methods to obtain aliasing information:

– The C language does not allow accessing an object using an lvalue of a different type. Exceptions are accessing an object using a qualified version of the same type and accessing an object using a character type. In the following example `p1` and `p2` must not point to the same object:

```
f(int *p1, long *p2)
{
  ...
}
```

`vbcc` will assume that the source is correct and does not break this requirement of the C language. If a program does break this requirement and cannot be fixed, then `-no-alias-opt` must be specified and some performance will be lost.

– At higher optimization levels, `vbcc` will try to keep track of all objects a pointer can point to. In the following example, `vbcc` will see that `p1` can only point to `x` or `y` whereas `p2` can only point to `z`. Therefore it knows that `p1` and `p2` are not aliased.

```
int x[10], y[10], z[10];

int f(int a, int b, int c)
{
  int *p1, *p2;

  if(a < b)
    p1 = &x[a];
  else
    p1 = &y[b];

  p2 = &z[c];

  ...
}
```

As pointers itself may be aliased and function calls might modify pointers, this analysis sometimes benefits from a larger scope of optimization. See Section 3.4.15 [Inter-Procedural Analysis], page 29 and Section 3.4.16 [Cross-Module Optimizations], page 29.

This optimization will alter the behaviour of broken code which uses pointer arithmetic to step from one object into another.

– The 1999 C standard provides the `restrict`-qualifier to help alias analysis. If a pointer is declared with this qualifier, the compiler may assume that the object pointed to by this pointer is only aliased by pointers which are derived from this pointer. For a formal definition of the rules for `restrict` please consult ISO/IEC9899:1999.

`vbcc` will make use of this information at higher optimization levels (‘`-c99`’ must be used to use this new keyword).

A very useful application for `restrict` are function parameters. Consider the following example:

```
void cross_prod(float *restrict res,
                float *restrict x,
                float restrict *y)
```

```
{
  res[0] = x[1] * y[2] - x[2] * y[1];
  res[0] = x[2] * y[0] - x[0] * y[2];
  res[0] = x[0] * y[1] - x[1] * y[0];
}
```

Without `restrict`, a compiler has to assume that writing the results through `res` can modify the object pointed to by `x` and `y`. Therefore, the compiler has to reload all the values on the right side twice. With `restrict` `vbcc` will optimize this code like:

```
void cross_prod(float *restrict res,
                float *restrict x,
                float restrict *y)
{
  float x0 = x[0], x1 = x[1], x2 = x[2];
  float y0 = y[0], y1 = x[1], y2 = y[2];

  res[0] = x1 * y2 - x2 * y1;
  res[0] = x2 * y0 - x0 * y2;
  res[0] = x0 * y1 - x1 * y0;
}
```

3.4.15 Inter-Procedural Analysis

Apart from the number of different optimizations a compiler offers, another important point is the scope of the underlying analysis. If a compiler only looks at small parts of code when deciding whether to do an optimization, it often cannot prove that a transformation does not change the behaviour and therefore has to reject it.

Simple compilers only look at single expressions, simple optimizing compilers often restrict their analysis to basic blocks or extended basic blocks. Analyzing a whole function is common in today's optimizing compilers.

This already allows many optimizations but often worst-case assumptions have to be made when a function is called. To avoid this, `vbcc` will not restrict its analysis to single functions at higher optimization levels. Inter-procedural data-flow analysis often allows for example to eliminate more common subexpressions or dead code. Register allocation and many other optimizations also sometimes benefit from inter-procedural analysis.

Further extension of the scope of optimizations is possible by activating cross-module optimizations. See Section 3.4.16 [Cross-Module Optimizations], page 29.

3.4.16 Cross-Module Optimizations

Separate compilation has always been an important feature of the C language. Splitting up an application into several modules does not only reduce turn-around times and resource-requirements for compilation, but it also helps writing reusable well-structured code.

However, an optimizer has much more possibilities when it has access to the entire source code. In order to provide maximum possible optimizations without sacrificing structure and

modularity of code, `vbcc` can do optimizations across different translation-units. Another benefit is that cross-module analysis also will detect objects which are declared inconsistently in different translation-units.

Unfortunately common object-code does not contain enough information to perform aggressive optimization, To overcome this problem, `vbcc` offers two solutions:

- If cross-module optimizations are enabled and several files are passed to `vbcc`, it will read in all files at once, perform optimizations across these files and generate a single object file as output. This file is similar to what would have been obtained by separately compiling the files and linking the resulting objects together.
- The method described above often requires changes in makefiles and somewhat different handling. Therefore `vbcc` also provides means to generate some kind of special pseudo object files which pretain enough high-level information to perform aggressive optimizations at link time.

  If '`-wpo`' is specified (which will automatically be done by `vc` at higher optimization levels) `vbcc` will generate such files rather than normal assembly or object files. These files can not be handled by normal linkers. However, `vc` will detect these files and before linking it will pass all such files to `vbcc` again. `vbcc` will optimize the entire code and generate real code which is then passed to the linker.

  It is possible to pass `vc` a mixture of real and pseudo object files. `vc` will detect the pseudo objects, compile them and link them together with the real objects. Obviously, `vc` has to be used for linking. Directly calling the linker with pseudo objects will not work.

  Please note that optimization and code generation is deferred to link-time. Therefore, all compiler options related to optimization and code generation have to be specified at the linker command as well. Otherwise they would be ignored. Other options (e.g. setting paths or defining macros) have to be specified when compiling.

  Also, turn-around times will obviously increase as usually everything will be rebuild even if makefiles are used. While only the corresponding pseudo object may be rebuilt if one file is changed, all the real work will be done at the linking stage.

3.4.17 Instruction Scheduling

Some backends provide an instruction scheduler which is automatically run by `vc` at higher optimization levels. The purpose is to reorder instructions to make better use of the different pipelines a CPU may offer.

The exact details depend heavily on the backend, but in general the scheduler will try to place instructions which can be executed in parallel (e.g. on super-scalar architectures) close to each other. Also, instructions which depend on the result of another instruction will be moved further apart to avoid pipeline-stalls.

Please note that it may be crucial to specify the correct derivate of a CPU family in order to get best results from the sceduler. Different variants of an architecture may have a different number and behaviour of pipelines requiring different scheduling decisions.

Consult the backend documentation for details.

3.4.18 Target-Specific Optimizations

In addition to those optimzations which are available for all targets, every backend will provide a series of additional optimizations. These vary between the different backends, but optimizations frequently done by backends are:

- use of complex or auto-increment addressing-modes
- implicit setting of condition-codes
- instruction-combining
- delayed popping of stack-slots
- optimized function entry- and exit-code
- elimination of a frame pointer
- optimized multiplication/division by constants
- inline code for block-copying

3.4.19 Debugging Optimized Code

Debugging of optimized code is usually not possible without problems. Many compilers turn off almost all optimizations when debugging. `vbcc` allows debugging output together with optimizations and tries to still do all optimizations (some restrictions have to be made regarding instruction-scheduling).

However, depending on the debugger and debugging-format used, the information displayed in the debugger may differ from the real situation. Typical problems are:

- Incorrectly displayed values of variables.

  When optimizing vbcc will often remove certain variables or eliminate code which sets them. Sometimes it is possible, to tell the debugger that a variable has been optimized away, but most of the time the debugger does not allow this and you will just get bogus values when trying to inspect a variable.

  Also, variables whose locations differs at various locations of the program (e.g. a variable is in a register at one place and in memory at another) can only be correctly displayed, if the debugger supports this.

  Sometimes, this can even occur in non-optimized code (e.g. with register-parameters or a changing stack-pointer).
- Strange program flow.

  When stepping through a program, you may see lines of code be executed out-of-order or parts of the code skipped. This often occurs due to code being moved around or eliminated/combined.
- Missed break-points.

  Setting break-points (especially on source-lines) needs some care when optimized code is debugged. E.g. code may have been moved or even replicated at different parts. A break-point set in a debugger will usually only be set on one instance of the code. Therefore, a different instance of the code may have been executed although the break-point was not hit.

3.5 Extensions

This section lists and describes all extensions to the C language provided by `vbcc`. Most of them are implemented in a way which does not break correct C code and still allows all diagnostics required by the C standard by using reserved identifiers.

The only exception (see Section 3.5.3 [Inline-Assembly Functions], page 33) can be turned off using '`-iso`' or '`-ansi`'.

3.5.1 Pragmas

`vbcc` accepts the following `#pragma`-directives:

`#pragma printflike <function>`
`#pragma scanflike <function>`
: `vbcc` will handle `<function>` specially. `<function>` has to be an already declared function, with external linkage, that takes a variable number of arguments and a `const char *` as the last fixed parameter.

  If such a function is called with a string-constant as format-string, `vbcc` will check if the arguments seem to match the format-specifiers in the format-string, according to the rules of printf or scanf. Also, `vbcc` will replace the call by a call to a simplified version according to the following rules, if such a function has been declared with external linkage:

  - If no format-specifiers are used at all, `__v0<function>` will be called.
  - If no qualifiers are used and only `d,i,x,X,o,s,c` are used, `__v1<function>` will be called.
  - If no floating-point arguments are used, `__v2<function>` will be called.

`#pragma dontwarn <n>`
: Disables warning number n. Must be followed by `#pragma popwarn`.

`#pragma warn <n>`
: Enables warning number n. Must be followed by `#pragma popwarn`.

`pragma popwarn`
: Undoes the last modification done by `#pragma warn` or `#pragma dontwarn`.

`#pragma only-inline on`
: The following functions will be parsed and are available for inlining (see Section 3.4.11 [Function Inlining], page 25), but no out-of-line code will be generated, even if some calls could not be inlined.

  Do not use this with functions that have local static variables!

`#pragma only-inline off`
: The following functions are translated as usual again.

`#pragma opt <n>`
: Sets the optimization options to `<n>` (similar to -O=`<n>`) for the following functions. This is only used for debugging purposes. Do not use!

`#pragma begin_header`
: Used to mark the beginning of a system-header. Must be followed by `#pragma end_header`. Not for use in applications!

`#pragma end_header`
: The counterpart to `#pragma begin_header`. Marks the end of a system-header. Not for use in applications!

3.5.2 Register Parameters

If the parameters for certain functions should be passed in certain registers, it is possible to specify the registers using `__reg("<reg>")` in the prototype, e.g.

```
void f(__reg("d0") int x, __reg("a0") char *y)  ...
```

The names of the available registers depend on the backend and will be listed in the corresponding part of the documentation. Note that a matching prototype must be in scope when calling such a function - otherwise wrong code will be generated. Therefore it is not useful to use register parameters in an old-style function-definition.

If the backend cannot handle the specified register for a certain type, this will cause an error. Note that this may happen although the register could store that type, if the backend does not provide the necessary support.

Also note that this may force `vbcc` to create worse code.

3.5.3 Inline-Assembly Functions

Only use them if you know what you are doing!

A function-declaration may be followed by '=' and a string-constant. If a function is called with such a declaration in scope, no function-call will be generated but the string-constant will be inserted in the assembly-output. Otherwise the compiler and optimizer will treat this like a function-call, i.e. the inline-assembly must not modify any callee-save registers without restoring them. However, it is also possible to specify the side-effects of inline-assembly functions like registers used or variables used and modified (see Section 3.5.9 [Specifying side-effects], page 35).

Example:

```
double sin(__reg("fp0") double) = "\tfsin.x\tfp0\n";
```

There are several issues to take care of when writing inline-assembly.

- As inline-assembly is subject to loop unrolling or function inlining it may be replicated at different locations. Unless it is absolutely known that this will not happen, the code should not define any labels (e.g. for branches). Use offsets instead.
- If a backend provides an instruction scheduler, inline-assembly code will also be scheduled. Some schedulers make assumptions about their input (usually compiler-generated code) to improve the code. Have a look at the backend documentation to see if there are any issues to consider.
- If a backend provides a peephole optimizer which optimizes the assembly output, inline-assembly code will also be optimized unless '`-no-inline-peephole`' is specified. Have a look at the backend documentation to see if there are any issues to consider.

– `vbcc` assumes that inline-assembly does not introduce any new control-flow edges. I.e. control will only enter inline-assembly if the function call is reached and if control leaves inline-assembly it will continue after the call.

Inline-assembly-functions are not recognized when ANSI/ISO mode is turned on.

3.5.4 Variable Attributes

`vbcc` offers attributes to variables or functions. These attributes can be specified at the declaration of a variable or function and are syntactically similar to storage-class-specifiers (e.g. `static`).

Often, these attributes are specific to one backend and will be documented in the backend-documentation (typical attributes would e.g. be `__interrupt` or `__section`). Attributes may also have parameters. A generally available attribute s `__entry` which is used to preserve unreferenced objects and functions (see Section 3.4.13 [Unused Object Elimination], page 27):

```
__entry __interrupt __section("vectab") void my_handler()
```

Additional non-target-specific attributes are available to specify side-effects of functions (see Section 3.5.9 [Specifying side-effects], page 35).

Please note that some common extensions like `__far` are variable attributes on some architectures, but actually type attributes (see Section 3.5.5 [Type Attributes], page 34) on others. This is due to significantly different meanings on different architectures.

3.5.5 Type Attributes

Types may be qualified by additional attributes, e.g. `__far`, on some backends. Regarding the availability of type attributes please consult the backend documentation.

Syntactically type attributes have to be placed like a type-qualifier (e.g. `const`). As example, some backends know the attribute `__far`.

Declaration of a pointer to a far-qualified character would be

```
__far char *p;
```

whereas

```
char * __far p;
```

is a far-qualified pointer to an unqualified char.

Please note that some common extensions like `__far` are type attributes on some architectures, but actually variable attributes (see Section 3.5.4 [Variable Attributes], page 34) on others. This is due to significantly different meanings on different architectures.

3.5.6 `__typeof`

`__typeof` is syntactically equivalent to sizeof, but its result is of type int and is a number representing the type of its argument. This may be necessary for implementing '`stdarg.h`'.

3.5.7 __alignof

__alignof is syntactically equivalent to sizeof, but its result is of type int and is the alignment in bytes of the type of the argument. This may be necessary for implementing 'stdarg.h'.

3.5.8 __offsetof

__offsetof is a builtin version of the offsetof-macro as defined in the C language. The first argument is a structure type and the second a member of the structure type. The result will be a constant expression representing the offset of the specified member in the structure.

3.5.9 Specifying side-effects

Only use if you know what you are doing!

When optimizing and generating code, vbcc often has to take into account side-effects of function-calls, e.g. which registers might be modified by this function and what variables are read or modified.

A rather imprecise way to make assumptions on side-effects is given by the ABI of a certain system (that defines which registers have to be preserved by functions) or rules derived from the language (e.g. local variables whose address has not been taken cannot be accessed by another function).

On higher optimization levels (see Section 3.4.15 [Inter-Procedural Analysis], page 29 and see Section 3.4.16 [Cross-Module Optimizations], page 29)) vbcc will try to analyse functions and often gets much more precise informations regarding side-effects.

However, if the source code of functions is not visible to vbcc, e.g. because the functions are from libraries or they are written in assembly (see Section 3.5.3 [Inline-Assembly Functions], page 33), it is obviously not possible to analyze the code. In this case, it is possible to specify these side-effects using the following special variable-attributes (see Section 3.5.4 [Variable Attributes], page 34).

The __regsused(<register-list>) attribute specifies the registers used or modified by a function. The register list is a list of register names (as defined in the backend-documentation) separated by slashes and enclosed in double-quotes, e.g.

```
__regsused("d0/d1") int abs();
```

declares a function abs which only uses registers d0 and d1.

__varsmodified(<variable-list>) specifies a list of variables with external linkage which are modified by the function. __varsused is similar, but specifies the external variables read by the function. If a variable is read and written, both attributes have to be specified. The variable-list ist a list of identifiers, separated by slashes and enclosed in double quotes.

The attribute __writesmem(<type>) is used to specify that the function accesses memory using a certain type. This is necessary if the function modifies memory accessible to the calling function which cannot be specified using __varsmodified (e.g. because it is accessed via pointers). __readsmem is similar, but specifies memory which is read.

If one of `__varsused`, `varsmodified`, `__readsmem` and `__writesmem` is specified, all relevant side-effects must be specified. If, for example, only `__varsused("my_global")` is specified, this implies that the function only reads `my_global` and does not modify any variable accessible to the caller.

All of these attributes may be specified multiple times.

3.6 Known Problems

Some known target-independent problems of `vbcc` at the moment:

- Bitfields are not really supported (they are always used as int).
- `volatile` is sometimes ignored by the optimizer.
- Some exotic scope-rules are not handled correctly.
- Debugging-infos cannot be used on higher optimization-levels.
- String-constants are not merged.

3.7 Credits

All those who wrote parts of the `vbcc` distribution, made suggestions, answered my questions, tested `vbcc`, reported errors or were otherwise involved in the development of `vbcc` (in descending alphabetical order, under work, not complete):

Frank Wille
Gary Watson
Andrea Vallinotto
Johnny Tevessen
Gabriele Svelto
Dirk Stoecker
Ralph Schmidt
Markus Schmidinger
Thorsten Schaaps
Anton Rolls
Michaela Pruess
Thomas Pornin
Joerg Plate
Gilles Pirio
Bartlomiej Pater
Gunther Nikl
Robert Claus Mueller
Joern Maass
Aki M Laukkanen
Kai Kohlmorgen
Uwe Klinger

Andreas Kleinert
Julian Kinraid
Acereda Macia Jorge
Dirk Holtwick
Tim Hanson
Kasper Graversen
Jens Granseuer
Volker Graf
Marcus Geelnard
Matthias Fleischer
Alexander Fichtner
Olivier Fabre
Robert Ennals
Thomas Dorn
Walter Doerwald
Aaron Digulla
Lars Dannenberg
Sam Crow
Michael Bode
Michael Bauer
Juergen Barthelmann
Thomas Arnhold
Alkinoos Alexandros Argiropoulos
Thomas Aglassinger

4 M68k/Coldfire Backend

This chapter documents the backend for the M68k and Coldfire processor families.

4.1 Additional options

This backend provides the following additional options:

'`-cpu=n`' Generate code for cpu n (e.g. -cpu=68020), default: 68000.

'`-fpu=n`' Generate code for fpu n (e.g. -fpu=68881), default: 0.

'`-sd`' Use small data model (see below).

'`-sc`' Use small code model (see below).

'`-prof`' Insert code for profiling.

'`-const-in-data`'
By default constant data will be placed in the code section (and therefore is accessable with faster pc-relative addressing modes). Using this option it will be placed in the data section.

This could e.g. be useful if you want to use small data and small code, but your code gets too big with all the constant data.

Note that on operating systems with memory protection this option will disable write-protection of constant data.

'`-use-framepointer`'
By default automatic variables are addressed through a7 instead of a5. This generates slightly better code, because the function entry and exit overhead is reduced and a5 can be used as register variable etc.

However this may be a bit confusing when debugging and you can force `vbcc` to use a5 as a fixed framepointer.

'`-no-peephole`'
Do not perform peephole-optimizations.

'`-no-delayed-popping`'
By default arguments of function calls are not always popped from the stack immediately after the call, so that the arguments of several calls may be popped at once. With this option `vbcc` can be forced to pop them after every function call. This may simplify debugging and reduce the stack size needed by the compiled program.

'`-gas`' Create output suitable for the GNU assembler.

'`-no-fp-return`'
Do not return floats and doubles in floating-point registers even if code for an fpu is generated.

‘-no-mreg-return’
: Do not use multiple registers to return types that do not fit into a single register. This is mainly for backwards compatibility with certain libraries.

‘-hunkdebug’
: When creating debug-output (‘-g’ option) create Amiga debug hunks rather than DWARF2. Does not work with ‘-gas’.

‘-no-intz’
: When generating code for FPU do quick&dirty conversions from floating-point to integer. The code may be somewhat faster but will not correctly round to zero. Only use it if you know what you are doing.

4.2 ABI

The current version generates assembler output for use with the PhxAss assembler (c) by Frank Wille. Most peephole optimizations are done by the assembler so vbcc only does some that the assembler cannot make. The generated executables will probably only work with OS2.0 or higher.

With ‘-gas’ assembler output suitable for the GNU assembler is generated (the version must understand the Motorola syntax - some old ones do not). The output is only slightly modified from the PhxAss-output and will therefore result in worse code on gas.

The register names provided by this backend are:

```
 a0,  a1,  a2,  a3,  a4,  a5,  a6,  a7
 d0,  d1,  d2,  d3,  d4,  d5,  d6,  d7
fp0, fp1, fp2, fp3, fp4, fp5, fp6, fp7
```

The registers a0 - a7 are supported to hold pointer types. d0 - d7 can be used for integers types excluding long long, pointers and float if no FPU code is generated. fp0 - fp7 can be used for all floating point types if FPU code is generated.

Additionally the following register pairs can be used for long long:

```
d0/d1, d2/d3, d4/d5, d6/d7
```

The registers d0, d1, a0, a1, fp0 and fp1 are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved.

By default, all function arguments are passed on the stack.

All scalar types up to 4 bytes are returned in register d0, long long is returned in d0/d1. If compiled for FPU, floating point values are returned in fp0 unless ‘-no-fpreturn’ is specified. Types which are 8, 12 or 16 bytes large will be returned in several registers (d0/d1/a0/a1) unless ‘-no-mreg-return’ is specified. All other types are returned by passing the function the address of the result as a hidden argument - such a function must not be called without a proper declaration in scope.

Objects which have been compiled with different settings must not be linked together.

a7 is used as stack pointer. If ‘-sd’ is used, a4 will be used as small data pointer. If ‘-use-framepointer’ is used, a5 will be used as frame pointer. All other registers will be used by the register allocator and can be used for register parameters.

The size of the stack frame is limited to 32KB for early members of the 68000 family prior to 68020.

The basic data types are represented like:

type	size in bits	alignment in bytes	
char	8	1	
short	16	2	
int	32	2	
long	32	2	
long long	64	2	
all pointers	32	2	
float(fpu)	32	2	see below
double(fpu)	64	2	see below
long double(fpu)	64	2	see below

4.3 Small data

`vbcc` can access static data in two ways. By default all such data will be accessed with full 32bit addresses (large data model). However there is a second way. You can set up an address register (`a4`) to point into the data segment and then address data with a 16bit offset through this register.

The advantages of the small data model are that the program will usually be smaller (because the 16bit offsets use less space and no relocation information is needed) and faster.

The disadvantages are that one address register cannot be used by the compiler and that it can only be used if all static data occupies less than 64kb. Also object modules and libraries that have been compiled with different data models must not be mixed (it is possible to call functions compiled with large data model from object files compiled with small data model, but not vice versa and only functions can be called that way - other data cannot be accessed).

If small data is used with functions which are called from functions which have not been compiled with `vbcc` or without the small data model then those functions must be declared with the `__saveds` attribute or call `geta4()` as the first statement (do not use automatic initializations prior to the call to `geta4`). Note that `geta4()` must not be called through a function pointer!

4.4 Small code

In the small code model calls to external functions (i.e. from libraries or other object files) are done with 16bit offsets through the program counter rather than with absolute 32bit addresses.

The advantage is slightly smaller and faster code. The disadvantages are that all the code (including library functions) must be small enough. Objects/libraries can be linked together if they have been compiled with different code models.

4.5 CPUs

The values of '`-cpu=n`' have those effects:

'`n<68000`' Code for the Coldfire family is generated.

'n>=68000'
Code for the 68k family is generated.

'n>=68020'
- 32bit multiplication/division/modulo is done with the mul?.l, div?.l and div?l.l instructions.
- tst.l ax is used.
- extb.l dx is used.
- 16/32bit offsets are used in certain addressing modes.
- link.l is used.
- Addressing modes with scaling are used.

'n==68040'
- 8bit constants are not copied in data registers.
- Static memory is not subject to common subexpression elimination.

4.6 FPUs

At the moment the values of -fpu=n have those effects:

'n>68000' Floating point calculations are done using the FPU.

'n=68040'
'n=68060' Instructions that have to be emulated on these FPUs will not be used; at the moment this only includes the `fintrz` instruction in case of the 040.

4.7 Math

Long multiply on CPUs <68020 uses inline routines. This may increase code size a bit, but it should be significantly faster, because function call overhead is not necessary. Long division and modulo is handled by calls to library functions. (Some operations involving constants (e.g. powers of two) are always implemented by more efficient inline code.)

If no FPU is specified floating point math is done using math libraries. 32bit IEEE format is used for float and 64bit IEEE for double and long double.

If floating point math is done with the FPU floating point values are kept in registers and therefore may have extended precision sometimes. This is not ANSI compliant but will usually cause no harm. When floating point values are stored in memory they use the same IEEE formats as without FPU. Return values are passed in `fp0`.

Note that you must not link object files together if they were not compiled with the same `-fpu` settings and that a proper math library must be linked.

4.8 Target-Specific Variable Attributes

This backend offers the following variable attributes:

`__saveds` Load the pointer to the small data segment at function-entry. Applicable only to functions.

`__chip` Place variable in chip-memory. Only applicable on AmigaOS to variables with static storage-duration.

`__far` Do not place this variable in the small-data segment in small data mode. No effect in large data mode. Only applicable to variables with static storage-duration.

`__near` Currently ignored.

`__interrupt`
This is used to declare interrupt-handlers. The function using this attribute will save all registers it destroys (including scratch-registers) and return with `rte` rather than `rts`.

`__amigainterrupt`
Used to write interrupt-handlers for AmigaOS. Stack-checking for a function with this attribute will be disabled and if a value is returned in d0, the condition codes will be set accordingly.

`__section(<string-literal>)`
Places the variable/function in a section named according to the argument.

4.9 Predefined Macros

This backend defines the following macros:

`__M68K__`

`__M680x0` (Depending on the settings of '`-cpu`', e.g. `__M68020`.)

`__COLDFIRE`
(If a Coldfire CPU is selected.)

`__M68881` (If '`-fpu=68881`' is selected.)

`__M68882` (If code for another FPU is selected; '`-fpu=68040`' or '`-fpu=68060`' will set `__M68882`.)

4.10 Stack

If the '`-stack-check`' option is used, every function-prologue will call the function `__stack_check` with the stacksize needed by the current function on the stack. This function has to consider its own stacksize and must restore all registers.

If the compiler is able to calculate the maximum stack-size of a function including all callees, it will add a comment in the generated assembly-output (subject to change to labels).

4.11 Stdarg

A possible '`<stdarg.h>`' could look like this:

```
typedef unsigned char *va_list;
```

```
#define __va_align(type) (__alignof(type)>=4?__alignof(type):4)

#define __va_do_align(vl,type) ((vl)=(char *)((((unsigned int)(vl))+__va_align(type)

#define __va_mem(vl,type) (__va_do_align((vl),type),(vl)+=sizeof(type),((type*)(vl))

#define va_start(ap, lastarg) ((ap)=(va_list)(&lastarg+1))

#define va_arg(vl,type) __va_mem(vl,type)

#define va_end(vl) ((vl)=0)

#define va_copy(new,old) ((new)=(old))

#endif
```

4.12 Known problems

- Converting floating point values to unsigned integers is not correct if the value is >LONG_MAX and FPU code is generated.
- The extended precision of the FPU registers can cause problems if a program depends on the exact precision. Most programs will not have trouble with that, but programs which do exact comparisons with floating point types (e.g. to try to calculate the number of significant bits) may not work as expected (especially if the optimizer was turned on).

5 PowerPC Backend

This chapter documents the Backend for the PowerPC processor family.

5.1 Additional options for this version

This backend provides the following additional options:

'`-merge-constants`'
: Place identical floating point constants at the same memory location. This can reduce program size.

'`-const-in-data`'
: By default constant data will be placed in the `.rodata` section. Using this option it will be placed in the `.data` section. Note that on operating systems with memory protection this option will disable write-protection of constant data.

'`-fsub-zero`'
: Use fsub to load a floating-point-register with zero. This is faster but requires all registers to always contain valid values (i.e. no NaNs etc.) which may not be the case depending on startup-code, libraries etc.

'`-amiga-align`'
: Do not require any alignments greater than 2 bytes. This is needed when accessing Amiga system-structures, but can cause a performance penalty.

'`-elf`'
: Do not prefix symbols with '_'. Prefix labels with '.'.

'`-poweropen`'
: Generate code for the PowerOpen ABI like used in AIX. This does not work correctly yet.

'`-sc`'
: Generate code for the modified PowerOpen ABI used in the StormC compiler.

'`-no-regnames`'
: Do not use register names but only numbers in the assembly output. This is necessary to avoid name-conflicts when using '`-elf`'.

'`-setccs`'
: The V.4 ABI requires signalling (in a bit of the condition code register) when arguments to varargs-functions are passed in floating-point registers. vbcc usually does not make use of this and therefore does not set that bit by default. This may lead to problems when linking objects compiled by vbcc to objects/libraries created by other compilers and calling varargs-functions with floating-point arguments. '`-setccs`' will fix this problem.

'`-no-peephole`'
: Do not perform several peephole optimizations. Currently includes:

- better use of d16(r) addressing
- use of indexed addressing modes
- use of update-flag
- use of record-flag
- use of condition-code-registers to avoid certain branches

'`-use-lmw`'
: Use `lmw/stmw`-instructions. This can significantly reduce code-size. However these instructions may be slower on certain PPCs.

'`-madd`'
: Use the `fmadd/fmsub` instructions for combining multiplication with addition/subtraction in one instruction. As these instructions do not round between the operations, they have increased precision over separate addition and multiplication.

  While this usually does no harm, it is not ISO conforming and therefore not the default behaviour.

'`-eabi`'
: Use the PowerPC Embedded ABI (eabi).

'`-sd`'
: Place all objects in small data-sections.

'`-gas`'
: Create code suitable for the GNU assembler.

'`-no-align-args`'
: Do not align function arguments on the stack stricter than 4 bytes. Default with '`-poweropen`'.

5.2 ABI

This backend supports the following registers:

- `r0` through `r31` for the general purpose registers,
- `f0` through `f31` for the floating point registers and
- `cr0` through `cr7` for the condition-code registers.

Additionally, the register pairs `r3/r4, r5/r6, r7/r8, r9/r10, r14/r15, r16/r17, r18/r19, r20/r21, r22/r23, r24/r25, r26/r27, r28/r29` and `r30/r31` are available.

`r0, r11, r12, f0, f12` and `f13` are reserved by the backend.

The current version generates assembly output for use with the "pasm" assembler by Frank Wille or the GNU assembler. The generated code should work on 32bit systems based on a PowerPC CPU using the V.4 ABI or the PowerPC Embedded ABI (eabi).

The registers r0, r3-r12, f0-f13 and cr0-cr1 are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved. r1 is the stack-pointer and r13 is the small-data-pointer if small-data-mode is used.

The first 8 function arguments which have integer or pointer types are passed in registers r3 through r10 and the first 8 floating-point arguments are passed in registers f1 through f8. All other arguments are passed on the stack.

Integers and pointers are returned in r3 (and r4 for long long), floats and doubles in f1. All other types are returned by passing the function the address of the result as a hidden

argument - so when you call such a function without a proper declaration in scope you can expect a crash.

The elementary data types are represented like:

```
type          size in bits        alignment in bytes (-amiga-align)

char                 8                        1 (1)
short               16                        2 (2)
int                 32                        4 (2)
long                32                        4 (2)
long long           64                        8 (2)
all pointers        32                        4 (2)
float               32                        4 (2)
double              64                        8 (2)
```

5.3 Target-specific variable-attributes

The PPC-backend offers the following variable-attributes:

`__saveds` Load the pointer to the small data segment at function-entry. Applicable only to functions.

`__chip` Place variable in chip-memory. Only applicable on AmigaOS to variables with static storage-duration.

`__far` Do not place this variable in the small-data segment in small-data-mode. No effect in large-data-mode. Only applicable to variables with static storage-duration.

`__near` Currently ignored.

`__interrupt`
Return with rfi rather than blr.

`__section("name","attr")`
Place this function/object in section "name" with attributes "attr".

5.4 Target-specific pragmas

The PPC-backend offers the following #pragmas:

`#pragma amiga-align`
Set alignment like -amiga-alignment option.

`#pragma natural-align`
Align every type to its own size.

`#pragma default-align`
Set alignment according to command-line options.

5.5 Predefined Macros

This backend defines the following macros:

`__PPC__`

5.6 Stack

If the '`-stack-check`' option is used, every function-prologue will call the function `__stack_check` with the stacksize needed by this function in register r12. This function has to consider its own stacksize and must restore all registers.

If the compiler is able to calculate the maximum stack-size of a function including all callees, it will add a comment in the generated assembly-output (subject to change to labels).

5.7 Stdarg

A possible <stdarg.h> for V.4 ABI could look like this:

```
typedef struct
  int gpr;
  int fpr;
  char *regbase;
  char *membase;
 va_list;

char *__va_start(void);
char *__va_regbase(void);
int __va_fixedgpr(void);
int __va_fixedfpr(void);

#define va_start(vl,dummy) \
  ( \
    vl.gpr=__va_fixedgpr(), \
    vl.fpr=__va_fixedfpr(), \
    vl.regbase=__va_regbase(), \
    vl.membase=__va_start() \
  )

#define va_end(vl) ((vl).regbase=(vl).membase=0)

#define va_copy(new,old) ((new)=(old))

#define __va_align(type) (__alignof(type)>=4?__alignof(type):4)

#define __va_do_align(vl,type) ((vl).membase=(char *)((((unsigned int)((vl).membase

#define __va_mem(vl,type) (__va_do_align((vl),type),(vl).membase+=sizeof(type),((ty

#define va_arg(vl,type) \
```

```
( \
 (__typeof(type)&127)>10? \
   __va_mem((vl),type) \
 : \
  ( \
   (((__typeof(type)&127)>=6&&(__typeof(type)&127)<=8)) ? \
    ( \
     ++(vl).fpr<=8 ? \
       ((type*)((vl).regbase+32))[(vl).fpr-1] \
     : \
       __va_mem((vl),type) \
    ) \
   : \
    ( \
     ++(vl).gpr<=8 ? \
       ((type*)((vl).regbase+0))[(vl).gpr-1] \
     : \
       __va_mem((vl),type) \
    ) \
  ) \
)
```

A possible <stdarg.h> for V.4 ABI could look like this:

```
typedef unsigned char *va_list;

#define __va_align(type) (4)

#define __va_do_align(vl,type) ((vl)=(char *)((((unsigned int)(vl))+__va_align(type

#define __va_mem(vl,type) (__va_do_align((vl),type),(vl)+=sizeof(type),((type*)(vl)

#define va_start(ap, lastarg) ((ap)=(va_list)(&lastarg+1))

#define va_arg(vl,type) __va_mem(vl,type)

#define va_end(vl) ((vl)=0)

#define va_copy(new,old) ((new)=(old))
```

5.8 Known problems

- composite types are put on the stack rather than passed via pointer
- indication of fp-register-args with bit 6 of cr is not done well
- __interrupt does not save all modified registers

6 Instruction Scheduler

vsc - scheduler for vbcc (c) in 1997-99 by Volker Barthelmann

6.1 Introduction

vsc is an instruction-scheduler which reorders the assembly output of vbcc and tries to improve performance of the generated code by avoiding pipeline stalls etc.

Like the compiler vbcc it is split into a target independent and a target dependent part. However there may be code-generators for vbcc which do not have a corresponding scheduler.

This document only deals with the target independent parts of vsc. Be sure to read all the documents for your machine.

6.2 Usage

Usually `vsc` will be called by a frontend. However if you call it directly, it has to be done like this:

```
vsc [options] input-file output-file
```

The following options are supported:

'`-quiet`' Do not print the copyright notice.

'`-debug=<n>`'
Set debug-level to `<n>`.

Note that depending on the target `vbcc` may insert hints into the generated code to tell vsc what CPU to schedule for. Code scheduled for a certain CPU may run much slower on slightly different CPUs. Therefore it is especially important to specify the correct target-CPU when compiling.

6.3 Known problems

– works only on basic-blocks

7 C Library

This chapter describes the C library usually provided with `vbcc`.

7.1 Introduction

To execute code compiled by `vbcc`, a library is needed. It provides basic interfaces to the underlying operating system or hardware as well as a set of often used functions.

A big part of the library is portable across all architectures. However, some functions (e.g. for input/output or memory allocation) are naturally dependent on the operating system or hardware. There are several sections in this chapter dealing with different versions of the library.

The library itself often is split into several parts. A startup-code will do useful initializations, like setting up IO, parsing the command line or initializing variables and hardware.

The biggest part of the functions will usually be stored in one library file. The name and format of this file depends on the conventions of the underlying system (e.g. '`vc.lib`' or '`libvc.a`').

Often, floating point code (if available) is stored in a different file (e.g. '`m.lib`' or '`libm.a`'). If floating point is used in an application, it might be necessary to explicitly link with this library (e.g. by specifying '`-lm`').

In many cases, the include files provide special inline-code or similar optimizations. Therefore, it is recommended to always include the corresponding include file when using a library function. Even if it is not necessary in all cases, it may affect the quality of the generated code.

The library implements the functions specified by ISO9899:1989 as well as a small number of the new functions from ISO9899:1999.

7.2 Legal

Most parts of this library are public domain. However, for some systems, parts may be under a different license. Please consult the system specific documentation. Usually, linking against this library will not put any restrictions on the created executable unless otherwise mentioned.

7.3 AmigaOS/68k

This section describes specifics of the C library for AmigaOS/68k. The relevant files are '`startup.o`', '`minstart.o`', '`vc.lib`', '`vcs.lib`', '`mieee.lib`', '`m881.lib`', '`m040.lib`', '`amiga.lib`', '`amigas.lib`', '`extra.lib`' and '`extras.lib`'.

7.3.1 Startup

The startup code currently consists of a slightly modified standard Amiga startup ('`startup.o`'). The startup code sets up some global variables and initializes stdin, stdout and stderr. The exit code closes all open files and frees all memory. If you link with a math library the startup/exit code will be taken from there if necessary.

7.3.2 Floating point

Note that you have to link with a math library if you want to use floating point. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math libraries only. At the moment there are three math libraries:

'`mieee.lib`'
: This one uses the C= math libraries. The startup code will always open MathIeeeSingBas.library, MathIeeeDoubBas.library and MathIeeeDoubTrans.library. Float return values are passed in d0, double return values are passed in d0/d1. A 68000 is sufficient to use this library. You must not specify '`-fpu=...`' when you use this library.

'`m881.lib`'
: This one uses direct FPU instructions and function return values are passed in fp0. You must have a 68020 or higher and an FPU to use this library. You also have to specify '`-fpu=68881`'. Several FPU instructions that have to be emulated on 040/060 may be used.

'`m040.lib`'
: This one uses only direct FPU instructions that do not have to be emulated on a 040/060. Other functions use the Motorola emulation routines modified by Aki M Laukkanen. It should be used for programs compiled for 040 or 060 with FPU. Return values are passed in fp0.

Depending on the CPU/FPU selected, #including '`math.h`' will cause inline-code generated for certain math functions.

7.3.3 Stack

An application can specify the stack-size needed by defining a variable `__stack` (of type `size_t`) with external linkage, e.g.

```
size_t __stack=65536; /* 64KB stack-size */
```

The startup code will check whether the stack-size specified is larger than the default stack-size (as set in the shell) and switch to a new stack of appropriate size, if necessary.

If the '`-stack-check`' option is specified when compiling, the library will check for a stack overflow and abort the program, if the stack overflows. Note, however, that only code compiled with this option will be checked. Calls to libraries which have not been compiled with '`-stack-check`' or calls to OS function may cause a stack overflow which is not noticed.

Additionally, if '`-stack-check`' is used, the maximum stack-size used can be read by querying the external variable `__stack_usage`.

```
#include <stdio.h>

extern size_t __stack_usage;

main()
{
    do_program();
```

```
        printf("stack used: %lu\n",(unsigned long)__stack_usage);
    }
```

Like above, the stack used by functions not compiled using '`-stack-check`' or OS functions is ignored.

7.3.4 Small data model

When using the small data model of the 68k series CPUs, you also have to link with appropriate libraries. Most libraries documented here are also available as small data versions (with an 's' attached to the file name). Exceptions are the math libraries.

To compile and link a program using the small data model a command like

```
vc test.c -o test -sd -lvcs -lamigas
```

might be used.

7.3.5 Restrictions

The following list contains some restrictions of this version of the library:

`tmpfile()`
: The `tmpfile()` function always returns an error.

`clock()`
: The `clock()` function always returns -1. This is correct, according to the C standard, because on AmigaOS it is not possible to obtain the time used by the calling process. For programs which cannot deal with this, a special version of `clock()` is provided in '`extra.lib`' (See Section 7.3.9 [extra.lib], page 57).

7.3.6 Minimal startup

If you want to write programs that use only Amiga functions and none from vc.lib you can use '`minstart.o`' instead of '`startup.o`' and produce smaller executables.

This is only useful for people who know enough about the Amiga shared libraries, the stubs in amiga.lib etc. If you do not know enough about those things better forget minstart at all.

This startup code does not set up all the things needed by vc.lib, so you must not use most of those functions (string and ctype funtions are ok, but most other functions - especially I/O and memory handling - must not be used). `exit()` is supplied by minstart and can be used.

The command line is not parsed, but passed to `main()` as a single string, so you can declare main as `int main(char *command)` or `int main(void)`.

Also no Amiga libraries are opened (but `SysBase` ist set up), so you have to define and open `DOSBase` yourself if you need it. If you want to use floating point with the IEEE libraries you have to define and open MathIeeeSingBas.library, MathIeeeDoubBas.library and MathIeeeDoubTrans.library (in this order!) and link with mieee.lib (if compiled for FPU this is not needed).

A hello world using minstart could look like this:

```
#include <exec/libraries.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

struct Library *DOSBase;

main()
{
    if(DOSBase=OpenLibrary("dos.library",0)){
        Write(Output(),"Hello, world!\n",14);
        CloseLibrary(DOSBase);
    }
    return 0;
}
```

This can yield an executable of under 300 bytes when compiled with '`-sc -sd -O2`' and linked with '`minstart.o`' and `amigas.lib` (using `vlink` - may not work with other linkers).

7.3.7 amiga.lib

To write programs accessing AmigaOS (rather than standard C functions only), a replacement for the original (copyrighted) '`amiga.lib`' is provided with `vbcc`. This replacement is adapted to vbcc, does not cause collisions with some functions (e.g. `sprintf`) provided by the original '`amiga.lib`' and is available in small data. It is recommended to always use this library rather than the original version.

Additionally, there are header files (in the '`proto`'- and '`inline`'-subdirectories) which cause inlined calls to Amiga library functions.

For AmigaOS/68k, '`amiga.lib`' is linked by default.

7.3.8 auto.lib

To link with '`auto.lib`' (or the small data version '`autos.lib`') specify the '`-lauto`' or '`-lautos`' option to `vc`.

When you are calling a standard Amiga library function and do not have defined the corresponding library base then the library base as well as code to open/close it will be taken from '`auto.lib`'.

By default, '`auto.lib`' will try to open any library version. If you need at least a certain version you can define and set a variable _<library-base>Ver with external linkage, e.g. (on file-scope):

```
int _IntuitionBaseVer = 39;
```

Note that your program will abort before reaching `main()` if one of the libraries cannot be opened. Also note that the dos.library will be openened by the standard startup code, not by auto.lib. This means you have to open dos.library yourself, when linking with '`minstart.o`'.

7.3.9 extra.lib

To link with '`extra.lib`' (or the small data version '`extras.lib`'), specify '`-lextra`' or '`-lextras`' to `vc`. There is also a header file '`extra.h`'.

At the moment extra.lib contains the following functions:

`getch()` Similar to `getchar()` but sets the console to raw mode and does not wait for return.

`stricmp()`
Case-insensitive variant of `strcmp()`.

`strnicmp()`
Case-insensitive variant of `strncmp()`.

`chdir()` Changes the current directory.

`clock()` This is an alternative version of the `clock()` function in '`vc.lib`'. `clock()` is supposed to return the amount of cpu-time spent by the current program. As this is generally not possible on the Amiga `clock()` from '`vc.lib`' always returns -1. However there seem to be several badly written programs that do not check this. So this version of `clock()` returns values which are strictly ascending for successive calls to `clock()` and the difference between them is the time difference between the calls.

`iswhitespace()`
Variant of `isspace()`.

`isseparator()`
Tests for ',' or '`|`'.

7.3.10 ixemul

7.3.10.1 Introduction

ixemul.library is a shared Amiga library which is covered by the GNU license and includes standard ANSI and POSIX functions as well as some functions common in Unix, BSD and similar operating systems. Have a look at the ixemul-archives if you want to find out more.

The link library ixemul.lib provided with vbcc only contains stub functions which call the functions in the shared library plus some auxiliary functions for vbcc.

What are the main differences between vc.lib and ixemul.lib?

- vc.lib contains (almost) only standard ANSI functions. If you want to port Unix programs you will probably miss a lot of functions. Also ixemul supports things like mapping Unix directory paths to Amiga paths or expanding wildcards in command lines automatically.
- Programs compiled for ixemul will be shorter because the code for all functions is not contained in the executable itself.
- Programs compiled for ixemul will need the ixemul.library present when started.

– Programs compiled for ixemul will probably need more memory because the entire (rather large) ixemul.library will be loaded into memory. With vc.lib only the functions your program uses will be in ram. However if you have several programs using ixemul.library at the same time only one copy of ixemul.library should be loaded.

Things you should note:

– With ixemul you do not need extra math-libraries. Floating point values are always returned in d0/d1 therefore -no-fp-return must be specified. The config-file will usually take care of this.
– You must link with vlibm68k:crt0.o as startup code. The config-file will usually take care of this.
– You _must_ use the ixemul-includes (they are currently not part of this archive) rather than the ones which are for vc.lib. The config-file will usually take care of this.
– I recommend you get the ixemul-distribution from aminet (in dev/gcc), and have a look at it (although a lot is gcc-specific).

7.3.10.2 Legal

The '`crt0.o`' and '`ixemul.lib`' in this archive are public domain. Have a look at the ixemul-archives to find out about ixemul.library.

7.3.10.3 Usage

First you must have the ixemul.library and the ixemul-includes (they should e.g. be on aminet in dev/gcc/ixemul-sdk.lha) and assign ixinclude: to the directory where they can be found.

To compile a program to use ixemul you must make sure the proper config-file ('`ixemul`') is used, e.g.

```
vc +ixemul hello.c
```

or

The small data model is currently not supported with '`ixemul.lib`', but the small code model is.

If your program needs a certain version of the ixemul.library you can define a variable `__ixemulVer` with external linkage which specifies the minimum required version, e.g.

```
int __ixemulVer = 45;
```

7.4 PowerUp/PPC

This section describes specifics of the C library for PowerUp/PPC. The relevant files are '`startup.o`', '`minstart.o`', '`libvc.a`', '`libvcs.a`', '`libm.a`', '`libamiga.a`', '`libamigas.a`', '`libextra.a`' and '`libextras.a`'.

7.4.1 Startup

The startup code '`startup.o`' sets up some global variables and initializes stdin, stdout and stderr. The exit code closes all open files and frees all memory. If you link with a math library the startup/exit code will be taken from there if necessary.

7.4.2 Floating point

Note that you have to link with a math library if you want to use floating point. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math libraries only.

The math library ('`libm.a`') is linked against the floating point library libmoto by Motorola.

Depending on the CPU/FPU selected, #including '`math.h`' will cause inline-code generated for certain math functions.

7.4.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 7.3.3 [amiga-stack], page 54). The only difference is that stack-swapping cannot be done. If the default stack-size is less than the stack-size specified with `__stack` the program will abort.

7.4.4 Small data model

When using the small data model of the PPC series CPUs, you also have to link with appropriate libraries. Most libraries documented here are also available as small data versions (with an 's' attached to the file name). Exceptions are the math libraries.

To compile and link a program using the small data model a command like

```
vc test.c -o test -sd -lvcs -lamigas
```

might be used.

7.4.5 Restrictions

The following list contains some restrictions of this version of the library:

`tmpfile()`
: The `tmpfile()` function always returns an error.

`clock()`
: The `clock()` function always returns -1. This is correct, according to the C standard, because on AmigaOS it is not possible to obtain the time used by the calling process. For programs which cannot deal with this, a special version of `clock()` is provided in '`extra.lib`' (See Section 7.3.9 [extra.lib], page 57).

7.4.6 Minimal startup

The provided minimal startup code ('`minstart.o`') is used similarly like the one for 68k (See Section 7.3.6 [Minimal startup], page 55). Only use it if you know what you are doing.

7.4.7 libamiga.a

To write programs accessing AmigaOS (rather than standard C functions only), a replacement for the original (copyrighted) '`amiga.lib`' is provided with `vbcc`. This replacement ('`libamiga.a`') automatically performs a necessary context switch to the 68k to execute the system call. Furthermore, it is adapted to vbcc, does not cause collisions with some functions (e.g. `sprintf`) provided by the original '`amiga.lib`' and is available in small data.

Specify '`-lamiga`' to link with '`libamiga.a`'.

7.4.8 libauto.a

This library corresponds to the AmigaOS/68k version (See Section 7.3.8 [auto.lib], page 56).

7.4.9 libextra.a

This library corresponds to the AmigaOS/68k version (See Section 7.3.9 [extra.lib], page 57).

7.5 WarpOS/PPC

This section describes specifics of the C library for WarpOS/PPC. The relevant files are '`startup.o`', '`vc.lib`', '`m.lib`' '`amiga.lib`' and '`extra.lib`'.

7.5.1 Startup

The startup code '`startup.o`' sets up some global variables and initializes stdin, stdout and stderr. The exit code closes all open files and frees all memory. If you link with a math library the startup/exit code will be taken from there if necessary.

7.5.2 Floating point

Note that you have to link with a math library if you want to use floating point. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math libraries only.

The math library ('`m.lib`') contains functions from Sun's portable floating point library. Additionally, there is a `vbcc` version of Andreas Heumann's '`ppcmath.lib`'. These routines are linked against Motorola's floating point routines optimized for PowerPC and therefore are much faster.

To make use of this library, link with '`ppcmath.lib`' before '`m.lib`', e.g.

```
vc test.c -lppcmath -lm
```

Depending on the CPU/FPU selected, #including '`math.h`' will cause inline-code generated for certain math functions.

7.5.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 7.3.3 [amiga-stack], page 54).

7.5.4 Restrictions

The following list contains some restrictions of this version of the library:

`tmpfile()`
: The `tmpfile()` function always returns an error.

`clock()`
: The `clock()` function always returns -1. This is correct, according to the C standard, because on AmigaOS it is not possible to obtain the time used by the calling process. For programs which cannot deal with this, a special version of `clock()` is provided in '`extra.lib`' (See Section 7.3.9 [extra.lib], page 57).

7.5.5 amiga.lib

To write programs accessing AmigaOS (rather than standard C functions only), a replacement for the original (copyrighted) '`amiga.lib`' is provided with `vbcc`. This replacement automatically performs a necessary context switch to the 68k to execute the system call. Furthermore, it is adapted to vbcc, does not cause collisions with some functions (e.g. `sprintf`) provided by the original '`amiga.lib`' and is available in small data.

Specify '`-lamiga`' to link with '`amiga.lib`'.

7.5.6 auto.lib

This library corresponds to the AmigaOS/68k version (See Section 7.3.8 [auto.lib], page 56).

7.5.7 extra.lib

This library corresponds to the AmigaOS/68k version (See Section 7.3.9 [extra.lib], page 57).

7.6 MorphOS/PPC

This section describes specifics of the C library for MorphOS/PPC. The relevant files are '`startup.o`', '`minstart.o`', '`libvc.a`', '`libvcs.a`', '`libm.a`', '`libamiga.a`', '`libamigas.a`', '`libextra.a`' and '`libextras.a`'.

7.6.1 Startup

The startup code '`startup.o`' sets up some global variables and initializes stdin, stdout and stderr. The exit code closes all open files and frees all memory. If you link with a math library the startup/exit code will be taken from there if necessary.

7.6.2 Floating point

Note that you have to link with a math library if you want to use floating point. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math libraries only.

The math library ('`libm.a`') is linked against the floating point library libmoto by Motorola.

Depending on the CPU/FPU selected, #including '`math.h`' will cause inline-code generated for certain math functions.

7.6.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 7.3.3 [amiga-stack], page 54).

7.6.4 Small data model

When using the small data model of the PPC series CPUs, you also have to link with appropriate libraries. Most libraries documented here are also available as small data versions (with an 's' attached to the file name). Exceptions are the math libraries.

To compile and link a program using the small data model a command like

```
vc test.c -o test -sd -lvcs -lamigas
```

might be used.

7.6.5 Restrictions

The following list contains some restrictions of this version of the library:

`tmpfile()`
: The `tmpfile()` function always returns an error.

`clock()`
: The `clock()` function always returns -1. This is correct, according to the C standard, because on MorphOS it is not possible to obtain the time used by the calling process. For programs which cannot deal with this, a special version of `clock()` is provided in '`extra.lib`' (See Section 7.3.9 [extra.lib], page 57).

7.6.6 libamiga.a

To write programs using AmigaOS compatible functions, a replacement for the original (copyrighted) '`amiga.lib`' is provided with `vbcc`. This replacement ('`libamiga.a`') will invoke the MorphOS 68k emulator to execute the system function. Furthermore, it is adapted to vbcc and does not cause collisions with some functions (e.g. `sprintf`) and is available in small data.

Specify '`-lamiga`' to link with '`libamiga.a`'.

7.6.7 libauto.a

This library corresponds to the AmigaOS/68k version (See Section 7.3.8 [auto.lib], page 56).

7.6.8 libextra.a

This library corresponds to the AmigaOS/68k version (See Section 7.3.9 [extra.lib], page 57).

8 List of Errors

0. "declaration expected" (Fatal, Error, ANSI-violation)
   Something is pretty wrong with the source.
1. "only one input file allowed" (Fatal)
   vbcc accepts only a single filename to compile. You can use a frontend to compile multiple files or perhaps you mistyped an option.
2. "Flag <%s> specified more than once" ()
   You specified a command line option that should be specified only once more than once. Maybe you have this option in your config-file and used it in the command line, too? The first occurrence will override the latter ones.
3. "Flag <%s> needs string" (Fatal)
   This option hast to be specified with a string parameter, e.g. -flag=foobar
4. "Flag <%s> needs value" (Fatal)
   This option hast to be specified with an integer parameter, e.g. -flag=1234
5. "Unknown Flag <%s>" (Fatal)
   This option is not recognized by vbcc. Perhaps you mistyped it, used the wrong case or specified an option of the frontend to vbcc?
6. "No input file" (Fatal)
   You did not specify an input file. Your source file should not start with a '-' and if you use a frontend make sure it has the proper suffix.
7. "Could not open <%s> for input" (Fatal)
   A file could not be opened.
8. "need a struct or union to get a member" (Error, ANSI-violation)
   The source contains something like a.b where a is not a structure or union.
9. "too many (%d) nested blocks" (Fatal, Error)
   vbcc only allows a maximum number of nested blocks (compund-statements). You can increase this number by changing the line #define MAXN <something> in vbc.h and recompiling vbcc.
10. "left block 0" (Error, ANSI-violation)
    This error should not occur.
11. "incomplete struct <%s>" (Error, ANSI-violation)
    You tried to get a member of an incomplete structure/union. You defined struct x y; somewhere without defining struct x{...}.
12. "out of memory" (Fatal, Error)
    Guess what.
13. "redeclaration of struct <%s>" (Error, ANSI-violation)
    You may not redeclare a struct/union in the same block.
14. "incomplete type (%s) in struct" (Error, ANSI-violation)
    Every member in a struct/union declaration must be complete. Perhaps you only wanted a pointer to that type and forgot the '*'?

15. "function (%s) in struct/union" (Error, ANSI-violation)
    Functions cannot be members of structs/unions.
16. "redeclaration of struct/union member <%s>" (Error, ANSI-violation)
    Two members of a struct/union have the same name.
17. "redeclaration of <%s>" (Error, ANSI-violation)
    You used a name already in use in an enumeration.
18. "invalid constant expression" (Error, ANSI-violation)
    ??? Nowhere to find...
19. "array dimension must be constant integer" (Error, ANSI-violation)
    The dimensions of an array must be constants (real constants, const int x=100; int y[x]; is not allowed) and integers (int y[100.0]; is not allowed either).
20. "no declarator and no identifier in prototype" (Error, ANSI-violation)
21. "invalid storage-class in prototype" (Error, ANSI-violation)
    Function parameters may only be auto or register.
22. "void not the only function argument" (Error, ANSI-violation)
    You tried to declare a function that has an argument of type void as well as other arguments.
23. "<%s> no member of struct/union" (Error, ANSI-violation)
    The struct/union does not contain a member called like that.
24. "increment/decrement is only allowed for aithmetic and pointer types" (Error, ANSI-violation)
25. "functions may not return arrays or functions" (Error, ANSI-violation)
26. "only pointers to functions can be called" (Error, ANSI-violation)
    You tried to call something that did not decay into a pointer to a function.
27. "redefinition of var <%s>" (Error, ANSI-violation)
28. "redeclaration of var <%s> with new storage-class" (Error, ANSI-violation)
29. "first operand of conditional-expression must be arithmetic or pointer type" (Error, ANSI-violation)
30. "multiple definitions of var <%s>" (Error, ANSI-violation)
    There have been multiple definitions of a global variable with initialization.
31. "operands of : do not match" (Error, ANSI-violation)
    In an expression of the form a ? b : c - a and b must have the same type or - a and b both must have arithmetic types or - one of them must be a pointer and the other must be void * or 0
32. "function definition in inner block" (Error, ANSI-violation)
    C does not allow nested functions.
33. "redefinition of function <%s>" (Error, ANSI-violation)
    Defining two functions with the same name in one translation-unit is no good idea.
34. "invalid storage-class for function" (Error, ANSI-violation)
    Functions must not have storage-classes register or auto.

35. "declaration-specifiers expected" (Error, ANSI-violation)
36. "declarator expected" (Error, ANSI-violation)
37. "<%s> is no parameter" (Error, ANSI-violation)

    In an old-style function definition you tried to declare a name as parameter which was not in the identifier-list.
38. "assignment of different structs/unions" (Error, ANSI-violation)
39. "invalid types for assignment" (Error, ANSI-violation)

    In an assignment-context (this includes passing arguments to prototyped functions) the source and target must be one of the following types:

    - both are arithmetic types - both are the same struct/union - one of them is a pointer to void and the other one is any pointer - the target is any pointer and the source is an integral constant-expression with the value 0 - both are pointer to the same type (here the target may have additional const/volatile qualifiers - not recursively, however)

    Any other combinations should be illegal.
40. "only 0 can be compared against pointer" (Warning, ANSI-violation)

    You may not compare a pointer against any other constant but a 0 (null pointer).
41. "pointers do not point to the same type" (Warning, ANSI-violation)

    You tried to compare or assign pointers that point to different types. E.g. the types they point to may have different attributes.
42. "function initialized" (Error, Fatal, ANSI-violation)

    There was a '=' after a function declaration.
43. "initialization of incomplete struct" (Error, Fatal, ANSI-violation)

    A structure is incomplete if the only its name, but not the content is known. You cannot do much with such structures.
44. "initialization of incomplete union" (Error, Fatal, ANSI-violation)

    A union is incomplete if the only its name, but not the content is known. You cannot do much with such unions.
45. "empty initialization" (Error, ANSI-violation)

    There was no valid expression after the '=' in a variable definition.
46. "initializer not a constant" (Error, ANSI-violation)

    Static variables and compound types may only be initialized with constants. Variables with const qualifier are no valid constant-expressions here.

    Addresses of static variables are ok, but casting them may turn them into non-constant-expressions.
47. "double type-specifier" (Warning, ANSI-violation)
48. "illegal type-specifier" (Warning, ANSI-violation)
49. "multiple storage-classes" (Warning, ANSI-violation)
50. "storage-class specifier should be first" (Warning, ANSI-violation)
51. "bitfields must be ints" (Warning, ANSI-violation)
52. "bitfield width must be constant integer" (Warning, ANSI-violation)

53. "struct/union member needs identifier" (Warning, ANSI-violation)
54. "; expected" (Warning, ANSI-violation)
    Probably you forgot a ';' or there is a syntactic error in an expression.
55. "struct/union has no members" (Warning, ANSI-violation)
    You defined an empty struct or union.
56. "} expected" (Warning, ANSI-violation)
57. ", expected" (Warning, ANSI-violation)
58. "invalid unsigned" (Warning, ANSI-violation)
59. ") expected" (Warning, ANSI-violation)
60. "array dimension has sidefx (will be ignored)" (Warning, ANSI-violation)
61. "array of size 0 (set to 1)" (Warning, ANSI-violation)
    ANSI C does not allow arrays or any objects to have a size of 0.
62. "] expected" (Warning, ANSI-violation)
63. "mixed identifier- and parameter-type-list" (Warning, ANSI-violation)
64. "var <%s> was never assigned a value" (Warning)
65. "var <%s> was never used" (Warning)
66. "invalid storage-class" (Warning, ANSI-violation)
67. "type defaults to int" (Warning)
68. "redeclaration of var <%s> with new type" (Warning, ANSI-violation)
69. "redeclaration of parameter <%s>" (Warning, ANSI-violation)
70. ": expected" (Warning, ANSI-violation)
71. "illegal escape-sequence in string" (Warning, ANSI-violation)
72. "character constant contains multiple chars" (Warning)
73. "could not evaluate sizeof-expression" (Error, ANSI-violation)
74. "" expected (unterminated string)" (Error, ANSI-violation)
75. "something wrong with numeric constant" (Error, ANSI-violation)
76. "identifier expected" (Fatal, Error, ANSI-violation)
77. "definition does not match previous declaration" (Warning, ANSI-violation)
78. "integer added to illegal pointer" (Warning, ANSI-violation)
79. "offset equals size of object" (Warning)
80. "offset out of object" (Warning, ANSI-violation)
81. "only 0 should be cast to pointer" (Warning)
82. "unknown identifier <%s>" (Error, ANSI-violation)
83. "too few function arguments" (Warning, ANSI-violation)
84. "division by zero (result set to 0)" (Warning, ANSI-violation)
85. "assignment of different pointers" (Warning, ANSI-violation)
86. "lvalue required for assignment" (Error, ANSI-violation)
87. "assignment to constant type" (Error, ANSI-violation)
88. "assignment to incomplete type" (Error, ANSI-violation)

89. "operands for || and && have to be arithmetic or pointer" (Error, ANSI-violation)
90. "bitwise operations need integer operands" (Error, ANSI-violation)
91. "assignment discards const" (Warning, ANSI-violation)
    You assigned something like (const type *) to (type *).
92. "relational expression needs arithmetic or pointer type" (Error, ANSI-violation)
93. "both operands of comparison must be pointers" (Error, ANSI-violation)
    You wrote an expression like a == b where one operand was a pointer while the other was not. Perhaps a function is not declared correctly or you used NULL instead of 0?
94. "operand needs arithmetic type" (Error, ANSI-violation)
95. "pointer arithmetic with void * is not possible" (Error, ANSI-violation)
    Adding/subtracting from a pointer to void is not possible.
96. "pointers can only be subtracted" (Error, ANSI-violation)
    You cannot add, multiply etc. two pointers.
97. "invalid types for operation <%s>" (Error, ANSI-violation)
98. "invalid operand type" (Error, ANSI-violation)
99. "integer-pointer is not allowed" (Error, ANSI-violation)
    You may not subtract a pointer from an integer. Adding an integer or subtracting it from a pointer is ok.
100. "assignment discards volatile" (Warning, ANSI-violation)
    You assigned something like (volatile type *) to (type *).
101. "<<, >> and % need integer operands" (Error, ANSI-violation)
102. "casting from void is not allowed" (Error, ANSI-violation)
    Casting something of type void to anything makes no sense.
103. "integer too large to fit into pointer" (Error, ANSI-violation)
    You tried to assign an integer to a pointer that is too small to hold the integer. Note that assignment of pointers<->integers is never portable.
104. "only integers can be cast to pointers" (Error, ANSI-violation)
105. "invalid cast" (Error, ANSI-violation)
106. "pointer too large to fit into integer" (Error, ANSI-violation)
    You tried to assign a pointer to an integer that is too small to hold the pointer. Note that assignment of pointers<->integers is never portable.
107. "unary operator needs arithmetic type" (Error, ANSI-violation)
108. "negation type must be arithmetic or pointer" (Error, ANSI-violation)
109. "complement operator needs integer type" (Error, ANSI-violation)
110. "pointer assignment with different qualifiers" (Warning, ANSI-violation)
    You tried to assign a pointer to a pointer that points to a type with different qualifiers (e.g. signed<->unsigned).
111. "dereferenced object is no pointer" (Error, ANSI-violation)
112. "dereferenced object is incomplete" (Error, ANSI-violation)
    You tried to dereference a pointer to an incomplete object. Either you had a pointer to an array of unknown size or a pointer to a struct or union that was not (yet) defined.

113. "only 0 should be assigned to pointer" (Warning, ANSI-violation)
     You may not assign constants other than a null pointer to any pointer.
114. "typedef <%s> is initialized" (Warning, ANSI-violation)
115. "lvalue required to take address" (Error, ANSI-violation)
     You can only get the address of an object, but not of expressions etc.
116. "unknown var <%s>" (Error, ANSI-violation)
117. "address of register variables not available" (Error, ANSI-violation)
     If a variable is declared as 'register' its address may not be taken (no matter if the variable actually gets assigned to a register).
118. "var <%s> initialized after 'extern'" (Warning)
119. "const var <%s> not initialized" (Warning)
     A constant variable was not initialized in its definition. As there is no other legal way to assign a value to a constant variable this is probable an error.
120. "function definition after 'extern'" (Warning, ANSI-violation)
121. "return type of main is not int" (Warning, ANSI-violation)
     main() should be defined as
     int main(int argc, char **argv)
     Especially the return type of main must be 'int' - 'void' is not allowed by ANSI C.
122. "invalid storage-class for function parameter" (Warning, ANSI-violation)
     Function parameters may only have 'auto' or 'register' as storage-class. 'static' or 'extern' are not allowed.
123. "formal parameters conflict with parameter-type-list" (Warning, ANSI-violation)
124. "parameter type defaults to int" (Warning)
     A function definition contains no explicit type specifier. 'int' will be assumed.
125. "no declaration-specifier, used int" (Warning, ANSI-violation)
     A variable was declared/defined without a type specified. This is not allowed in ANSI C (apart from functions).
126. "no declarator in prototype" (Warning, ANSI-violation)
127. "static var <%s> never defined" (Warning)
128. "} expected" (Warning)
129. "left operand of comma operator has no side-effects" (Warning)
     In an expression of the form a,b a has no side-effects and is therefore superfluous.
130. "label empty" (Error, ANSI-violation)
     There was a ':' without an identifier before it.
131. "redefinition of label <%s>" (Error, ANSI-violation)
     The label was defined more than once in the same function. Consider that labels can not be hidden in inner blocks.
132. "case without switch" (Error, ANSI-violation)
     A case label was found outside of any switch-statements.

133. "case-expression must be constant" (Error, ANSI-violation)
The expression after 'case' must be constant.
134. "case-expression must be integer" (Error, ANSI-violation)
The expression after 'case' must be integer.
135. "empty if-expression" (Error, ANSI-violation)
There was no valid expression after 'if'.
136. "if-expression must be arithmetic or pointer" (Error, ANSI-violation)
The expression after 'if' must be arithmetic (i.e. an integer or floating point type) or a pointer.
137. "empty switch-expression" (Error, ANSI-violation)
There was no valid expression after 'switch'.
138. "switch-expression must be integer" (Error, ANSI-violation)
The expression after 'switch' must be an integer.
139. "multiple default labels" (Error, ANSI-violation)
There was more than one default label in a switch-statement.
140. "while-expression must be arithmetic or pointer" (Error, ANSI-violation)
The expression after the 'while' must be arithmetic (i.e. an integer or floating point type) or a pointer.
141. "empty while-expression" (Error, ANSI-violation)
There was no valid expression after 'while'.
142. "for-expression must be arithmetic or pointer" (Error, ANSI-violation)
The expression inside the 'for' must be arithmetic (i.e. an integer or floating point type) or a pointer.
143. "do-while–expression must be arithmetic or pointer" (Error, ANSI-violation)
The expression after the 'while' must be arithmetic (i.e. an integer or floating point type) or a pointer.
144. "goto without label" (Error, ANSI-violation)
'goto' must be followed by a label.
145. "continue not within loop" (Error, ANSI-violation)
'continue' is only allowed inside of loops. Perhaps there are unbalanced '{' '}'.
146. "break not in matching construct" (Error, ANSI-violation)
'break' is only allowed inside of loops or switch-statements. Perhaps there are unbalanced '{' '}'.
147. "label <%s> was never defined" (Error, ANSI-violation)
There is a goto to a label that was never defined.
148. "label <%s> was never used" (Warning)
You defined a label, but there is no goto that jumps to it.
149. "register %s not ok" (Warning)
There was an internal error (i.e. a bug in the compiler)! Please report the error to vb@compilers.de. Thanks!

150. "default not in switch" (Warning, ANSI-violation)

     A default label that is not in any switch-statement was found. Perhaps there are unbalanced '{' '}'.

151. "(expected" (Warning, ANSI-violation)

152. "loop eliminated" (Warning)

     There was a loop that will never be executed (e.g. while(0)...) and therefore the entire loop was eliminated. I do not know any reason for such loops, so there is probably an error.

153. "statement has no effect" (Warning)

     There is a statement that does not cause any side-effects (e.g. assignments, function calls etc.) and is therefore superfluous. E.g. you might have typed a==b; instead of a=b;

154. "'while' expected" (Warning, ANSI-violation)

     The 'while' in a do-while loop is missing.

155. "function should not return a value" (Warning)

     You specified an argument to return although the function is void. Declare the function as non-void.

156. "function should return a value" (Warning)

     You did not specify an argument to return although the function is not void. Declare the function as void or specify a return value.

157. "{ expected" (Warning, ANSI-violation)

158. "internal error %d in line %d of file %s !!" (Fatal, Error)

     There was an internal error (i.e. a bug in the compiler)! Please report the error to vb@compilers.de. Thanks!

159. "there is no message number %d" (Fatal)

     You tried to activate or suppress a message that does not exist.

160. "message number %d cannot be suppressed" (Fatal)

     You cannot suppress a message that displays a real error, ANSI-violation or another real problem. Only 'harmless' warnings can be suppressed.

161. "implicit declaration of function <%s>" (Warning)

     A function was called before it was declared and therefore implicitly declared as

     int function();

     This should be avoided in new programs.

162. "function call without prototype in scope" (Warning)

     When writing new programs it is probably sensible to use prototypes for every function. If a function is called without a prototype in scope this may cause incorrect type conversions and is usually an error.

163. "#pragma used" (Warning)

     Usage of #pragma should be avoided in portable programs.

164. "assignment in comparison context" (Warning)

     The expression in an if-, for-, while- or do-while-statement is an assignment, e.g.

if(i=0)...

This could an error, if you wanted if(i==0). If you turned on this warning and want it to shut up for a certain expression you can cast it to its type, e.g.

if((int)(i=0))...

Note that only assignments with '=' will be warned, not '+=' etc.

165. "comparison redundant because operand is unsigned" (Warning)

A comparison with an unsigned variable is redundant, because the result will always be constant, e.g.

unsigned int i; if(i<0)...

This usually is a programming error and can be avoided in all cases.

166. "cast to narrow type may cause loss of precision" (Warning)

A variable is cast to a type smaller than its original type, so that some information may get lost. However this warning will be displayed in lots of cases where no problem can arise, e.g. (short)(a==b).

167. "pointer cast may cause alignment problems" (Warning)

A pointer is cast to a pointer to a type with stricter alignment requirements, i.e. the new pointer might be invalid if you do not know what you are doing. Those casts should be avoidable in all 'usual' cases.

168. "no declaration of global variable <%s> before definition" (Warning)

It is usually good to declare all global variables (including functions) in header files.

169. "'extern' inside function" (Warning)

Declaration of external variables in inner blocks is usually not a good idea.

170. "dead assignment to <%s> eliminated" (Warning)

A variable is assigned a value that is never used or gets overwritten before it is used. If this occurs in real code then there is either an errror or an unnecessary assignment.

This is detected only in optimizing compilation.

171. "var <%s> is used before defined" (Warning)

The variable is used before it was assigned a value and therefore is undefined. It cannot be detected if the code where it is used can be reached, but if it is reached it will cause undefined behaviour. So it is most probably an error either way (see 170).

However not all unitialized usages can be found.

Also note that the compiler may choose convenient values for uninitialized variables. Example:

int f(int a) { int x; if(a) x=0; return(x); }

Here the optimizer may choose that x==0 if it is uninitialized and then only generate a return(0); It can also happen that you get different values if you read an uninitialized variable twice although it was not assigned a value inbetween.

This is only detected in optimizing compilation.

172. "would need more than %ld optimizer passes for best results" (Warning)

The optimizer would probably be able to do some further optimizations if you increased the number of allowed passes with the -optpasses=n option.

173. "function <%s> has no return statement" (Warning)

     A non-void function has no return statement. Either this function never returns (then better declare it as void) or it reaches end of control which would be an error.

     As main() cannot be declared as void you will not be warned if main has no return statement. If you want a warning for main, too, you can turn on warning 174.

174. "function <main> has no return statement" (Warning)

     The same like 173 for main, so you can turn it on/off separately.

175. "this code is weird" (Warning)

     The code has a very strange control flow. There is probably a jump inside a loop or something similar and the optimizer will not make any loop optimization and perhaps worse register allocation on this construct. There must be goto statements in the source.

     This warning is only detected in optimizing compilation.

176. "size of incomplete type not available" (Warning, ANSI-violation)

     An incomplete type must not be the argument for sizeof.

177. "line too long" (FATAL, Error, ANSI-violation, Preprocessor)
178. "identifier must begin with a letter or underscore" (FATAL, Error, ANSI-violation, Preprocessor)
179. "cannot redefine macro" (Error, ANSI-violation, Preprocessor)
180. "missing) after argumentlist" (Error, ANSI-violation, Preprocessor)
181. "identifier expected" (Error, ANSI-violation, Preprocessor)
182. "illegal character in identifier" (Error, ANSI-violation, Preprocessor)
183. "missing operand before/after ##" (Error, ANSI-violation, Preprocessor)
184. "no macro-argument after #-operator" (Error, ANSI-violation, Preprocessor)
185. "macro redefinition not allowed" (Error, ANSI-violation, Preprocessor)
186. "unexpected end of file (unterminated comment)" (FATAL, Error, Preprocessor)
187. "too many nested includes" (FATAL, Error, Preprocessor)
188. "#else without #if/#ifdef/#ifndef" (FATAL, Error, ANSI-violation, Preprocessor)
189. "#else after #else" (Error, ANSI-violation, Preprocessor)
190. "#endif without #if" (Error, ANSI-violation, Preprocessor)
191. "cannot include file" (FATAL, Error, Preprocessor)
192. "expected \" or < in #include-directive" (Error, ANSI-violation, Preprocessor)
193. "unknown #-directive" (Warning, Preprocessor)
194. "wrong number of macro arguments" (Error, ANSI-violation, Preprocessor)
195. "macro argument expected" (Error, ANSI-violation, Preprocessor)
196. "out of memory" (FATAL, Error, Preprocessor)
197. "macro redefinition" (Warning, Preprocessor)
198. "/* in comment" (Warning, Preprocessor)
199. "cannot undefine macro" (Error, ANSI-violation, Preprocessor)
200. "characters after #-directive ignored" (Warning, Preprocessor)

201. "duplicate case labels" (Warning, ANSI-violation)

Each case-label in a switch-statement must have a distinct constant value attached (after converting it to the type of the switch-expression).

202. "var <%s> is incomplete" (Warning, ANSI-violation)

An incomplete var was defined. probably you wrote something like:

int a[];

203. "long float is no longer valid" (Warning, ANSI-violation)

'long float' was a synonym for double in K&R C, but this is no longer allowed in ANSI C.

204. "long double is not really supported by vbcc" (Warning)

vbcc does not know about long double yet and therefore will use it simply as a synonym for double. This should not break any legal code, but you will not get error messages if you e.g. assign a pointer to double to a pointer to long double.

205. "empty struct-declarations are not yet handled correct" (Warning)

obsolete

206. "identifier too long (only %d characters are significant)" (Warning)

207. "illegal initialization of var <%s>" (Warning, ANSI-violation)

Perhaps you tried to initialize a variable with external linkage in an inner block.

208. "suspicious loop" (Warning)

vbcc thinks a loop-condition looks suspicious. A possible example could be for(i=0;i!=7;i+=2)

209. "ansi/iso-mode turned on" (Warning)

You turned on the ANSI/ISO-conforming mode. This warning is always displayed unless it is suppressed. So vbcc cannot be blamed to miss a diagnostic for any constraint violation. :-)

210. "division by zero (result set to 0)" (Warning, ANSI-violation)

Similar to warning 84.

211. "constant out of range" (Warning, ANSI-violation)

An integral constant is too large to fit into an unsigned long.

212. "constant is unsigned due to size" (Warning)

If an integral constant is so large that it cannot be represented as long its type is promoted to unsigned long.

213. "varargs function called without prototype in scope" (Warning)

A function which takes a variable number of arguments must not be called without a prototype in scope. E.g. calling printf() without #include <stdio.h> may cause this warning.

214. "suspicious format string" (Warning)

The format-string of a printf-/scanflike function seems to be corrupt or not matching the type of the arguments.

215. "format string contains \'\\0\'" (Warning)

The format string for a printf-/scanflike function contains an embedded '\0' character.

216. "illegal use of keyword <%s>" (Warning, ANSI-violation)
     The reserved keywords of C may not be used as identifier.
217. "register <%s> used with wrong type" (Error)
218. "register <%s> is not free" (Error)
219. "'__reg' used in old-style function definition" (Warning)
220. "unknown register \"%s\"" (Warning)
221. "'...' only allowed with prototypes" (Warning, ANSI-violation)
222. "Hey, do you really know the priority of '&&' vs. '||'?" (Warning)
223. "be careful with priorities of <</>> vs. +/-" (Warning)
224. "adress of auto variable returned" (Warning)
225. "void function returns a void expression" (Warning)
226. "redeclaration of typedef <%s>" (Warning, ANSI-violation)
227. "multiple specification of attribute \"%s\"" (Warning)
228. "redeclaration of var \"%s\" with differing setting of attribute \"%s\"" (Warning)
229. "string-constant expected" (Error)
230. "tag \"%s\" used for wrong type" (Warning, ANSI-violation)
231. "member after flexible array member" (Error, ANSI-violation)
232. "illegal number" (Error, ANSI-violation)
233. "void character constant" (Preprocessor, Error, ANSI-violation)
234. "spurious tail in octal character constant" (Preprocessor, Error, ANSI-violation)
235. "spurious tail in hexadecimal character constant" (Preprocessor, Error, ANSI-violation)
236. "illegal escape sequence in character constant" (Preprocessor, Error, ANSI-violation)
237. "invalid constant integer value" (Preprocessor, Error, ANSI-violation)
238. "a right parenthesis was expected" (Preprocessor, Error, ANSI-violation)
239. "a colon was expected" (Preprocessor, Error, ANSI-violation)
240. "truncated constant integral expression" (Preprocessor, Error, ANSI-violation)
241. "rogue operator '%s' in constant integral expression" (Preprocessor, Error, ANSI-violation)
242. "invalid token in constant integral expression" (Preprocessor, Error, ANSI-violation)
243. "trailing garbage in constant integral expression" (Preprocessor, Error, ANSI-violation)
244. "void condition for a #if/#elif" (Preprocessor, Error, ANSI-violation)
245. "void condition (after expansion) for a #if/#elif" (Preprocessor, Error, ANSI-violation)
246. "invalid '#include'" (Preprocessor, Error, ANSI-violation)
247. "macro expansion did not produce a valid filename for #include" (Preprocessor, Error, ANSI-violation)
248. "file '%s' not found" (Preprocessor, Error, ANSI-violation)
249. "not a valid number for #line" (Preprocessor, Error, ANSI-violation)
250. "not a valid filename for #line" (Preprocessor, Error, ANSI-violation)

251. "rogue '#'" (Preprocessor, Error, ANSI-violation)
252. "rogue #else" (Preprocessor, Error, ANSI-violation)
253. "rogue #elif" (Preprocessor, Error, ANSI-violation)
254. "unmatched #endif" (Preprocessor, Error, ANSI-violation)
255. "unknown cpp directive '#%s'" (Preprocessor, Error, ANSI-violation)
256. "unterminated #if construction" (Preprocessor, Error, ANSI-violation)
257. "could not flush output (disk full ?)" (Preprocessor, Error, ANSI-violation)
258. "truncated token" (Preprocessor, Error, ANSI-violation)
259. "illegal character '%c'" (Preprocessor, Error, ANSI-violation)
260. "unfinished string at end of line" (Preprocessor, Error, ANSI-violation)
261. "missing macro name" (Preprocessor, Error, ANSI-violation)
262. "trying to redefine the special macro %s" (Preprocessor, Error, ANSI-violation)
263. "truncated macro definition" (Preprocessor, Error, ANSI-violation)
264. "'...' must end the macro argument list" (Preprocessor, Error, ANSI-violation)
265. "void macro argument" (Preprocessor, Error, ANSI-violation)
266. "missing comma in macro argument list" (Preprocessor, Error, ANSI-violation)
267. "invalid macro argument" (Preprocessor, Error, ANSI-violation)
268. "duplicate macro argument" (Preprocessor, Error, ANSI-violation)
269. "'__VA_ARGS__' is forbidden in macros with a fixed number of arguments" (Preprocessor, Error, ANSI-violation)
270. "operator '##' may neither begin nor end a macro" (Preprocessor, Error, ANSI-violation)
271. "operator '#' not followed by a macro argument" (Preprocessor, Error, ANSI-violation)
272. "macro '%s' redefined unidentically" (Preprocessor, Error, ANSI-violation)
273. "not enough arguments to macro" (Preprocessor, Error, ANSI-violation)
274. "unfinished macro call" (Preprocessor, Error, ANSI-violation)
275. "too many argument to macro" (Preprocessor, Error, ANSI-violation)
276. "operator '##' produced the invalid token '%s%s'" (Preprocessor, Error, ANSI-violation)
277. "quad sharp" (Preprocessor, Error, ANSI-violation)
278. "void macro name" (Preprocessor, Error, ANSI-violation)
279. "macro %s already defined" (Preprocessor, Error, ANSI-violation)
280. "trying to undef special macro %s" (Preprocessor, Error, ANSI-violation)
281. "illegal macro name for #ifdef" (Preprocessor, Error, ANSI-violation)
282. "unfinished #ifdef" (Preprocessor, Error, ANSI-violation)
283. "illegal macro name for #undef" (Preprocessor, Error, ANSI-violation)
284. "unfinished #undef" (Preprocessor, Error, ANSI-violation)
285. "illegal macro name for #ifndef" (Preprocessor, Error, ANSI-violation)
286. "unfinished #ifndef" (Preprocessor, Error, ANSI-violation)

287. "reconstruction of <foo> in #include" (Preprocessor, Warning)
288. "comment in the middle of a cpp directive" (Preprocessor, Warning)
289. "null cpp directive" (Preprocessor, Warning)
290. "rogue '#' in code compiled out" (Preprocessor, Warning)
291. "rogue '#' dumped" (Preprocessor, Warning)
292. "#error%s" (Preprocessor, ANSI-violation, Error)
293. "trigraph ?""?%c encountered" (Preprocessor, Warning)
294. "unterminated #if construction (depth %ld)" (Preprocessor, Error, ANSI-violation)
295. "malformed identifier with UCN: '%s'" (Preprocessor, Warning, ANSI-violation)
296. "truncated UTF-8 character" (Preprocessor, Warning, ANSI-violation)
297. "identifier not followed by whitespace in #define" (Preprocessor, Warning, ANSI-violation)
298. "assignment discards restrict" (Warning, ANSI-violation)
299. "storage-class in declaration within for() converted to auto" (Warning, ANSI-violation)
300. "corrupted special object" (ANSI-violation, Fatal)
301. "<inline> only allowed in function declarations" (Error, ANSI-violation)
302. "reference to static variable <%s> in inline function with external linkage" (Error, ANSI-violation)
303. "underflow of pragma popwarn" (Error, ANSI-violation)
304. "invalid argument to _Pragma" (Preprocessor, Error, ANSI-violation)
305. "missing comma before '...'" (Preprocessor, Error, ANSI-violation)
306. "padding bytes behind member <%s>" (Warning)
307. "member <%s> does not have natural alignment" (Warning)
308. "function <%s> exceeds %s limit" (Warning)
309. "%s could not be calculated for function <%s>" (Warning)
310. "offsetof applied to non-struct" (Error, ANSI-violation)