vbcc compiler system

Volker Barthelmann

February 2022

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 17) 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

An exception for commercial usage is granted, provided that the target CPU is 6502 with MEGA65 extensions and the target HW is MEGA65. Resulting binaries may be distributed commercially without further licensing.

In all other cases you need my written consent.

This copyright applies to vc, vbcc, vsc and vcpr.

This archive may contain other tools (e.g. assemblers or linkers) which do not fall under this license. Please consult the corresponding documentation of these tools.

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

There is an Amiga Installer, which lets you install the binary and target archives with a simple mouse click. `vbcc` for AmigaOS/MorphOS is divided into the following packages:

`vbcc_bin_amigaos68k`
: Binaries for AmigaOS 2.x/3.x (68020+).

`vbcc_bin_amigaosppc`
: Binaries for AmigaOS 4.x (PowerPC).

`vbcc_bin_morphos`
: Binaries for MorphOS (PowerPC).

`vbcc_bin_powerup`
: Additional PPC-native binaries using the PowerUp kernel for AmigaOS 3.x.

`vbcc_bin_warpos`
: Additional PPC-native binaries using the WarpOS kernel for AmigaOS 3.x.

`vbcc_target_m68k-kick13`
: Header files and libraries for AmigaOS 1.x/M68k.

`vbcc_target_m68k-amigaos`
: Header files and libraries for AmigaOS 2.x/3.x/M68k.

`vbcc_target_ppc-amigaos`
: Header files and libraries for AmigaOS 4.x.

`vbcc_target_ppc-morphos`
: Header files and libraries for MorphOS.

`vbcc_target_ppc-powerup`
: Header files and libraries for PowerUp.

`vbcc_target_ppc-warpos`
: Header files and libraries for WarpOS.

Usually you will install the binary archive for your host architecture of choice, then add as many target archives you need.

When installing manually it is recommended to add the following assigns to your `s:User-Startup` file (only do the assignments required for the installed targets):

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

assign >NIL: vincludeos1: vbcc:targets/m68k-kick13/include
assign >NIL: vincludeos1: <path to your Kickstart 1.x header files> ADD
assign >NIL: vlibos1: vbcc:targets/m68k-kick13/lib

assign >NIL: vincludeos3: vbcc:targets/m68k-amigaos/include
assign >NIL: vincludeos3: <path to your AmigaOS3 header files> ADD
assign >NIL: vlibos3: vbcc:targets/m68k-amigaos/lib

assign >NIL: vincludeos4: vbcc:targets/ppc-amigaos/include
assign >NIL: vincludeos4: <path to your AmigaOS4 header files> ADD
assign >NIL: vlibos4: vbcc:targets/ppc-amigaos/lib

assign >NIL: vincludemos: vbcc:targets/ppc-morphos/include
assign >NIL: vincludemos: <path to your MorphOS header files> ADD
assign >NIL: vlibmos: vbcc:targets/ppc-morphos/lib

assign >NIL: vincludepup: vbcc:targets/ppc-powerup/include
assign >NIL: vincludepup: <path to your PowerUp header files> ADD
assign >NIL: vincludepup: <path to your AmigaOS3 header files> ADD
assign >NIL: vlibpup: vbcc:targets/ppc-powerup/lib

assign >NIL: vincludewos: vbcc:target/ppc-warpos/include
```

```
assign >NIL: vincludewos: <path to your WarpOS header files> ADD
assign >NIL: vincludewos: <path to your AmigaOS3 header files> ADD
assign >NIL: vlibwos: vbcc:target/ppc-warpos/lib
```

Also, the stack-size has to be increased from the default, for those binaries which don't do that automatically (e.g. AmigaOS4). 64KB is a sensible value, for very large projects higher values might be necessary.

For writing AmigaOS/MorphOS programs you will need the appropriate system header files. Please use NDK 3.9 or later for AmigaOS 2.x/3.x, as the proto/inline headers have been created with it.

For AmigaOS 1.x you should use the old Kickstart 1.x headers from that time. It makes sure that you don't use a feature of later OS releases.

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 is <target>_<host>.

Available config files, when all targets are installed, are

`aos68k` AmigaOS 2.x/3.x.

`aos68km` AmigaOS 2.x/3.x with minimal startup code.

`aos68kr` AmigaOS 2.x/3.x for resident programs.

`kick13` AmigaOS 1.x.

`kick13m` AmigaOS 1.x with minimal startup code.

`kick13r` AmigaOS 1.x for resident programs.

`aosppc` AmigaOS 4.x on PPC using vclib.

`newlib` AmigaOS 4.x on PPC using newlib.

`morphos` PPC systems running MorphOS.

`powerup` PPC boards using the PowerUp system.

`warpos` PPC boards using the WarpOS system.

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

```
vc +aos68k_powerup ...
```

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 `aos68k_warpos` to `aos68k`, `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 68020-native tools on AmigaOS 2.x/3.x - e.g. `+warpos` will create code for WarpOS, but the compiler and tools will run on the 68k. The

default `vc.config` will then create code for 68k using tools running on 68k. Having installed the MorphOS-native binary archive instead, the default `vc.config` will create PPC code for MorphOS using tools running on MorphOS.

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, using the default configuration from `vc.config`, 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`, `-lm040` or `-lm060`).

```
vc calc.c -o calc -lm
```

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. An empty `path` adds the current directory.

`-lulib` Link with library `ulib`.

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

`-static` Instruct the linker to link against static libraries. This may override the default to link against dynamic libraries first.

`-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.

`-rmcfg-<opt>`
Ignore all lines from the config file starting with `-<opt>`.

`+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. Note that there is an internal preprocessor, called ucpp, since V0.8, you usually don't need this option any more.

-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=vbccm68k -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=vasmm68k_mot -quiet -Fhunk -phxass -opt-pea -opt-clr %s -o %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=vlink -x -Bstatic -Cvbcc -nostdlib -Lvlibos3: vlibos3:startup.o %s %s -lvc -o %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.

`-ldnodb=string`
: This option string is inserted in the linker command before specifying the libraries, whenever an executable without debugging information and symbols should be created (AKA as a 'stripped' executable).

`-ldstatic=string`
: This option string is inserted in the linker command before specifying the libraries when static linking was requested with option `-static`.

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. Usually the predefined optimization options by the compiler driver should be used. See Section 3.4 [Optimizations], page 17.

`-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 28.

`-wpo` Create a high-level pseudo object for cross-module optimization (see Section 3.4.16 [Cross-Module Optimizations], page 31).

`-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 32).

`-cmd=<file>`
: A file containing additional command line options can be specified using this command. This may be useful for very long command lines.

`-c89`

`-c99`
: Set the C standard to be used. The default is the 1999 ISO C standard (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
  - designated initializers
  - non-constant initializers for automatic aggregates
  - compound literals
  - variable-length arrays (incomplete)

`-unsigned-char`
: Make the unqualified type of `char` unsigned.

`-maxoptpasses=n`
: Set maximum number of optimizer passes to n. See Section 3.4 [Optimizations], page 17.

`-inline-size=n`
: Set the maximum 'size' of functions to be inlined. See Section 3.4.11 [Function Inlining], page 26.

`-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 26.

`-unroll-size=n`
: Set the maximum 'size' of unrolled loops. See Section 3.4.10 [Loop Unrolling], page 24.

`-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 24.

`-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 34.

`-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 29.

`-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.

`-short-push`
: In the presence of a prototype, no promotion will be done on function arguments. For example, <char> will be passed as <char> rather than <int> and <float> will not be promoted to <double>. This may be more efficient on small targets.

  However, please note that this feature may not be supported by all backends and that using this option breaks ANSI/ISO conformance. For example, a function with a <char> parameter must never be called without a prototype in scope.

`-soft-float`
: On targets supporting this flag, software floating point emulation will be used rather than a hardware FPU. Please consult the corresponding backend documentation when using this flag.

`-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[,n...]`
: Suppress warning number n; suppress all warnings if n<0. Multiple warnings may be separated by commas. See Section 3.2 [Errors and Warnings], page 16,

`-warn=n`
: Turn on warning number n; turn on all warnings if n<0. See Section 3.2 [Errors and Warnings], page 16,

`-no-cpp-warn`
: Turn off all preprocessor warnings.

`-warnings-as-errors`
: Treat all enabled warnings as errors.

`-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.

`-no-include-stack`
: Do not display the include stack in error messages.

`-+`
`-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.

`-deps`
: Write a make-style dependency-line to <file>.dep.

`-deps-for-libs`
: By default, `-deps` will not include files that are included using the syntax `#include <...>`. Specify this option to add those files as well.

`-depobj=<file>`
: Use the specified filename as target in the generated dependency file instead of basing it on the input file name.

`-reserve-reg=<register>`
: Reserve that register not to be used by the backend. This option is dangerous and must only be used for registers otherwise available for the register allocator. If it used for special registers or registers used internally by the backend, it may be ignored, lead to corrupt code or even cause internal errors from the compiler.

  Only use if you know what you are doing!

`-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.

`-prefer-statics`
: Assign auto variables to static memory rather than the stack if it can be deduced that the function is not called recursively, i.e. the behaviour is still C compliant. This may be more efficient on targets that can access static data faster than stack. While stack-usage is reduced, total memory consumption is usually increased.

  Functions will not be re-entrant any more.

  This option only has effect on higher optimization levels (`-O3`).

`-force-statics`
: Like `-prefer-statics`, but assume all functions as non-recursive. This will break C compliance.

  This option only has effect on higher optimization levels (`-O`).

`-range-opt`
: Perform additional optimizations based on value range analysis. This option is under development and considered experimental. The following optimizations are currently implemented:
  - Induction variables of some loops are transformed to smaller types if it can be determined that they will only get assigned values that fit into a smaller type.

`-merge-strings`
: Overlay identical string-constants to save memory. Currently only strings identical to string-constants on top-level are recognized.

`-sec-per-obj`
: Tells the backend to put every function/object into its own separate section. This allows more fine-grained elimination of unused functions/objects by the linker. On the other hand, it may prevent some optimizations by the assembler.

  This option only has effect if it is supported by the backend.

`-mask-opt`
: Perform mask optimizations on suitable library function. This will create calls to optimized versions of e.g. the printf/scanf family of functions.

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 15 [List of Errors], page 131, 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 19.

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 19.

Bit 2 (4) Perform common subexpression elimination (see Section 3.4.3 [Common Subexpression Elimination], page 20) and copy propagation (see Section 3.4.4 [Copy Propagation], page 21). 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 21). 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 22).

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 23, and Section 3.4.7 [Loop-Invariant Code Motion], page 22. 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 28).

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 29). 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 19).

Bit 11 (2048)
More aggressive loop optimizations are performed (see Section 3.4.10 [Loop Unrolling], page 24, and Section 3.4.9 [Induction Variable Elimination], page 23). 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 26).

Bit 13 (8192)
: Reserved.

Bit 14 (16384)
: Perform inter-procedural analysis (see Section 3.4.15 [Inter-Procedural Analysis], page 30) and cross-module optimizations (see Section 3.4.16 [Cross-Module Optimizations], page 31).

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 29, Section 3.4.15 [Inter-Procedural Analysis], page 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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 31.

See also Section 3.5.3 [Inline-Assembly Functions], page 34.

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 34, and Section 3.4.11 [Function Inlining], page 26.

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 31.

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 31.

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 30, and Section 3.4.16 [Cross-Module Optimizations], page 31.

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[1] = x[2] * y[0] - x[0] * y[2];
  res[2] = 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[1] = x2 * y0 - x0 * y2;
  res[2] = 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 31.

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 34) 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 26), 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!

`#pragma pack(n)`
: Set alignment of structure members to a multiple of `n` bytes.

`#pragma pack()`
: Restores structure alignment to the target's default alignment, which was in effect when the compilation started.

`#pragma pack(push[,n])`
: Pushes the current structure alignment onto an internal stack and optionally sets a new alignment to a multiple of `n` bytes.

`#pragma pack(pop)`
: Restores the topmost structure alignment, saved by `pack(push)`, from an internal stack. Restores the default alignment, when the stack is empty.

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 36).

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 28):

```
__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 36).

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 35) 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 35) 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 30, and see Section 3.4.16 [Cross-Module Optimizations], page 31)) `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 34), 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 35).

The `__regsused(<register-list>)` attribute specifies the volatile 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 is 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.

3.5.10 Automatic constructor/destructor functions

The linker `vlink` provides a feature to collect pointers to all functions starting with the names `_INIT` or `_EXIT` in a prioritized array, labeled by `__CTOR_LIST__` and `__DTOR_LIST__`. The C-library (vclib) calls the constructor functions before entering `main()` and the destructor functions on program exit.

The format of these special function names is:

```
void _INIT[_<pri>][_<name>](void)
void _EXIT[_<pri>][_<name>](void)
```

The optional priority `<pri>` may be a digit between 1 and 9, where a constructor with a priority of 1 is executed first while a destructor with a priority of 1 is executed last. `<name>` is an optional name, used to differentiate functions of the same level.

3.5.11 `__noinline`

`__noinline` will prevent inlining of a given function. The heuristic used for deciding whether a function should be inlined generally makes a good trade-off between code size and performance, but sometimes it can be useful to override this behaviour. Use-cases include keeping "cold" functions out-of line to reduce code size, or to allow safe use of inline assembly with labels.

3.5.12 Predefined macros

The following macros are defined by the compiler.

```
#define __VBCC__
#define __entry __vattr("entry")
#define __str(x) #x
#define __asm(x) dostatic void inline_assembly()=x;inline_assembly();while(0)
#define __regsused(x) __vattr("regused("x")")
#define __varsused(x) __vattr("varused("x")")
#define __varsmodified(x) __vattr("varchanged("x")")
#define __noreturn __vattr("noreturn()")
#define __alwaysreturn __vattr("alwaysreturn()")
#define __nosidefx __vattr("nosidefx()")
#define __stack(x) __vattr(__str(stack1(x)))
#define __stack2(x) __vattr(__str(stack2(x)))
```

```
#define __noinline __vattr("noinline()")
#define __STDC_VERSION__ 199901L
```

`__STDC_VERSION__` is defined in C99-mode only.

3.5.13 Masked symbols

Together with vlink, this feature allows to provide a family of specially tailored functions in a library.

A symbol can be attached a mask using the `__mask` attribute, e.g.

```
__mask(15) void myfunc(void);
```

The symbol of this function will get the suffix `.15` attached.

A reference to a masked symbol can either be created by vbcc itself when using the option `-mask-opt`, or manually by referencing a symbol prefixed with `__maskm_<mask>`.

```
extern __mask(15) void myfunc(void);
...
  __maskm_15_myfunc();
```

If there are several masked references to a symbol, the linker will pull the first symbol containing a mask that has at least all bits set that are set in any masked reference. If no masked symbols matching that requirement are available, the symbol without a mask is used. Also, a non-masked reference will only pull a non-masked symbol from a library.

Masked objects may only be defined in libraries and zero masks are not allowed.

3.6 Known Problems

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

- Some exotic scope-rules are not handled correctly.
- Debugging-infos may have problems on higher optimization-levels.
- String-constants are not merged (partially done).

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
- Eero Tamminen
- Gabriele Svelto
- Dirk Stoecker
- Ralph Schmidt
- Markus Schmidinger
- Thorsten Schaaps

- Anton Rolls
- Michaela Pruess
- Thomas Pornin
- Joerg Plate
- Gilles Pirio
- Bartlomiej Pater
- Elena Novaretti
- Gunther Nikl
- Constantinos Nicolakakis
- Timm S. Mueller
- Robert Claus Mueller
- Joern Maass
- Aki M Laukkanen
- Kai Kohlmorgen
- Uwe Klinger
- Andreas Kleinert
- Julian Kinraid
- Acereda Macia Jorge
- Dirk Holtwick
- Matthew Hey
- Tim Hanson
- Kasper Graversen
- Jens Granseuer
- Volker Graf
- Marcus Geelnard
- Franta Fulin
- 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:

`-a2scratch`
: Allow using `A2` as scratch register.

`-amiga-softfloat`
: Call AmigaOS MathIEEE library functions via direct inline code, instead of callling stub routines from `mieee.lib`. It still requires that you either link with `mieee.lib` or define `MathIeeeSingBasBase`, `MathIeeeDoubBasBase` and `MathIeeeDoubTransBase` yourself.

`-conservative-sr`
: Restrict strength-reduction. Experimental.

`-const-in-data`
: By default constant data will be placed in the code section (and therefore is accessible 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.

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

`-d2scratch`
: Allow using `D2` as scratch register.

`-fastcall`
: Pass function arguments in volatile registers, when possible.

`-fp2scratch`
: Allow using `FP2` as scratch register.

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

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

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

`-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.

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

`-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.

`-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.

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

`-no-reserve-regs`
: Do not reserve temporary registers for the backend. Can lead to worse code generation.

`-old-softfloat`
: Use old libcall mechanism for software floating point. Should not be used, will usually generate worse code.

`-old-libcalls`
: Use old libcall mechanism for (some) integer support routines. Should not be used, will usually generate worse code.

`-phxass`
: Generate assembly output for the PhxAss assembler.

`-prof`
: Insert code for profiling.

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

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

`-use-commons`
: Use real common symbols instead of bss symbols for non-initialized external variables.

`-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.

4.2 ABI

The current version generates assembler output for use with the `vasmm68k_mot`. 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 `vasm`-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:

`__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.

`__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.

`__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`.

`__near`
: Currently ignored.

`__regargs`
: Declare function to use the `-fastcall` ABI. The first arguments are passed in volatile registers.

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

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

`__stdargs`
: Declare function to use the standard ABI (default), which passes all arguments on the stack.

4.9 Target-specific pragmas

This backend offers the following #pragmas:

`#pragma stdargs-on`
: Automatically declare the following functions with the `__stdargs` attribute.

`#pragma stdargs-off`
: Stop automatically declaring the following functions with the `__stdargs` attribute.

4.10 Predefined Macros

This backend defines the following macros:

`__AMIGADATE__`
: This is set to current date as `"(DD.MM.YYYY)"`, useful with version strings.

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

`__INTSIZE`
: Is set to the size of the `int` type. Either 16 (vbccm68ks) or 32 (vbccm68k).

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

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

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

`__M68K__`

`__SMALL_DATA__`
: (If `-sd` is selected to enable small data.)

4.11 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).

4.12 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)-1
```

```
#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
```

4.13 Known problems

– 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:

`-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.

`-baserel32mos`
: Use 32bit base-relative addressing as used by MorphOS.

`-baserel32os4`
: Use 32bit base-relative addressing as used by AmigaOS 4.

`-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.

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

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

`-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.

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

`-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.

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

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

`-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

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

`-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 (aka WarpOS ABI).

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

`-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.

`-use-commons`
: Use real common symbols instead of bss symbols for non-initialized external variables.

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

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 `vasmppc` 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.

`__saveall`
Make sure all registers are saved by this function. On lower optimization levels, all volatile registers will be saved additionally. On higher levels, only the ones that may be destroyed, are saved.

`__interrupt`
Return with en `rfi`-instruction 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__`

`__AMIGADATE__`
: This is set to current date as `"(DD.MM.YYYY)"`, useful with version strings.

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.

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),((type*

#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 PowerOpen 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)-1

#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

6 DEC Alpha Backend

This chapter documents the Backend for the DEC Alpha processor family.

6.1 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 and increase compilation time.

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

`-no-builtins`
: Do not replace certain builtin functions by inline code. This may be useful if you use this code generator with another frontend than vbcc. stdarg.h will not work with -no-builtins.

`-stabs`
: Generate stabs debug infos (if -g is specified) rather than DWARF2 which is the default. Consider this obsolete.

6.2 ABI

This backend supports the following registers:

- `$0` through `$31` for the general purpose registers and
- `$f0` through `$f31` for the floating point registers.

The current version generates assembly output for use with the GNU assembler. The generated code should work on systems with 21064, 21066, 21164 and higher Alpha CPUs.

The registers `$0-$8, $16-$28, $f0, $f1` and `$f10-$f30` are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved. Of course `$31` and `$f31` cannot be used.

The first 6 function arguments which have arithmetic or pointer types are passed in registers `$16/$f16` through `$21/$f21`.

Integers and pointers are returned in `$0`, floats and doubles in `$f0`. 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
char	8	1
short	16	2
int	32	4
long	64	8

```
long long          64                    8
all pointers       64                    8
float              32                    4
double             64                    8
```

6.3 Predefined Macros

This backend defines the following macros:

`__ALPHA__`

6.4 Stdarg

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

```
typedef struct {
  char *regbase;
  char *membase;
  int arg;
} va_list;

char *__va_start(void);
int __va_fixargs(void);

#define va_start(vl,dummy) \
(vl.arg=__va_fixargs(),vl.regbase=__va_start(),vl.membase=vl.regbase+(6-vl.arg)*16

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

#define __va_size(type) ((sizeof(type)+7)/8*8)
#define va_arg(vl,type) \
 ( \
  ((__typeof(type)&15)<=10&&++vl.arg<=6) ? \
   ( \
    ((__typeof(type)&15)>=6&&(__typeof(type)&15)<=8) ? \
      (vl.regbase+=16,*(type *)(vl.regbase-8)) \
    : \
      (vl.regbase+=16,*(type *)(vl.regbase-16)) \
   ) \
  : \
   (vl.membase+=__va_size(type),*(type *)(vl.membase-__va_size(type))) \
 )
```

6.5 Known problems

– generated code is rather poor

– several other problems

7 i386 Backend

This chapter documents the Backend for the Intel i386 processor family.

7.1 Additional options for this version

This backend provides the following additional options:

`-longalign`
: Align multibyte-values on 4-byte-boundaries. Needed by some operating systems.

`-elf`
: Do not use a '_'-prefix in front of external identifiers. Use a '.'-prefix for label names.

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

`-const-in-data`
: By default constant data will be placed in a read-only 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.

`-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 very slightly reduce the stack size needed by the compiled program.

`-safe-fp`
: Do not use the floating-point-stack for register variables. At the moment this is necessary as vbcci386 still has problems in some cases otherwise.

7.2 ABI

This backend supports the following registers:

- `%eax, %ebx, %ecx, %edx`
- `%esi, %edi, %ebp, %esp`

(And `%st(0)-%st(7)` for the floating point stack but these must not bes used for register variables because they cannot be handled like normal registers.)

The current version generates assembly output for use with the GNU assembler. The generated code should work on systems with Intel 80386 or higher CPUs with FPU and compatible chips.

The registers `%eax, %ecx` and `%edx` (as well as the floating point stack) are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved.

All elementary types up to 4 bytes are returned in register `%eax` Floating point values are returned in %st(0). 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.

vbcc uses %eax, %ebx, %ecx, %edx, %esi, %edi, %ebp and the floating point stack for temporary results and register variables. Local variables are created on the stack and addressed via %esp.

The elementary data types are represented like:

```
type         size in bits        alignment in bytes (-longalign)

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

7.3 Predefined Macros

This backend defines the following macros:

__I386__

__X86__

7.4 Stdarg

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

```
typedef unsigned char *va_list;

#define va_start(ap, lastarg) ((ap) = (va_list)(&lastarg + 1))
#define va_arg(ap, type) ((ap) += \
  (sizeof(type)<sizeof(int)?sizeof(int):sizeof(type)), ((type *)(ap))[-1])
#define va_end(ap) ((ap) = 0L)
```

7.5 Known Problems

- generated code is rather poor
- functions which return floating-point values sometimes are broken(?)
- in some cases (scare registers) non-reentrant code is generated

8 c16x Backend

This chapter documents the Backend for the c16x/st10 microcontroller family.

8.1 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 and increase compilation time.

`-const-in-data`
: By default constant data will be placed in a read-only section. Using this option it will be placed in the data section.

`-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 very slightly reduce the stack size needed by the compiled program.

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

`-tasking` Produce output for the Tasking assembler.

`-mtiny` Assume all functions are within one code-segment. Shorter instructions for calling functions are used and function-pointers will be only 2 bytes long. This results in shorter and faster code.

`-mlarge` All objects which are not explicitly qualified are assumed to be far (i.e. they may be in different segments but must not cross one segment-boundary). The default pointer size will be 4.

`-mhuge` All objects which are not explicitly qualified are assumed to be huge (i.e. they may be in different segments and may cross segment-boundaries). The default pointer size will be 4.

`-int32` Do not use.

8.2 ABI

This backend supports the following registers:

– `R0` through `R15` for the general purpose registers

Additionally, the register pairs `R2/R3, R3/R4, R4/R5, R6/R7, R7/R8, R8/R9, R12/R13, R13/R14,` and `R15/R15` are available.

`R1, R11` and `R12` are reserved by the backend.

The current version generates assembly output for use with the vasm assembler. Optionally, assembly code for the Tasking assembler can be generated. The default memory

model corresponds to the Tasking small-memory model with 16bit data-pointers and 32bit function-pointers. However, the DPPx registers have to be set up in a way to create a linear 16bit address space (i.e. DPPx=x). The generated code should work on systems with c161, c163, c164, c165 and c167 microcontrollers as well as ST10 derivates. Old versions like the c166 are not supported

The registers R1-R5 and R10-R15 are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved.

R0 is used as user stack pointer. Automatic variables and temporaries are put on the user stack. Return addresses are pushed on the system stack.

The first 4 function arguments which have integer or pointer types are passed in registers R12 through R15.

Integers and pointers are returned in R4/R5. 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
char	8	1
short	16	2
int	16	2
long	32	2
long long	n/a	n/a
near pointers	16	2
far pointers	32	2
huge pointers	32	2
float	n/a	n/a
double	n/a	n/a

8.3 Target-specific variable-attributes

The c16x-backend offers the following variable attributes:

__interrupt
: Return with rfi rather than blr. MDL/MDH will be saved, however it is recommended to switch to a new register bank as the gprs are not saved. Also, DPP0-DPP3 are not saved (the compiler does not use them).

__interrupt(<vector>)
: Like __interrupt, but also places a jump-instruction to the interrupt service at the corresponding vector table address (needs support from library and linker file).

__section(<name>)
: Place this object/function in section <name>.

__rbank(<bank>)
: Switch to another register-bank for this function.

8.4 Target-specific type-attributes

The c16x-backend offers the following type attributes:

`__near` Object resides within the same segment.

`__far` Object may reside in a different segment, but does not cross a segment-boundary.

`__huge` Object may reside in a different segment and may cross a segment-boundary

`__section(<name>)`
Place this function or object in section <name>.

`__sfr(<addr>)`
Used to declare a special function register at <addr>.

Example:

```
__sfr(0xff10) volatile unsigned int PSW;
```

`__esfr(<addr>)`
The same for extended special function registers.

`__sfrbit(<addr>,<bit>)`
Declare a single bit in the bit-addressable area.

Example:

```
__sfr(0xff10,11) volatile __bit IEN;
```

`__esfrbit(<addr>,<bit>)`
The same for the extended bit-addressable area.

8.5 Target-specific types

The c16x-backend offers the following additional types:

`__bit` A single bit in the bit-addressable internal RAM-area. Only static and external variables may use this type. It is not allowed for auto or register storage-class. Also, arrays of bits, pointers to bits or bits within aggregates are not allowed. Conversion of a bit to an integral type yields 0 if the bit is cleared and 1 if it is set.. Conversion of an integral type to a bit clears the bit if the integer is equal to 0 and sets it otherwise.

8.6 Predefined Macros

This backend defines the following macros:

`__C16X__`

`__C167__`

`__ST10__`

8.7 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 `R1`. This function has to consider its own stacksize and must restore all registers.

Only stack-checking of the user-stack is supported. Checking the system-stack is supported by hardware.

8.8 Stdarg

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

```
typedef char *va_list;

va_list __va_start(void);

#define __va_rounded_size(__TYPE)  \
  (((sizeof (__TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))

#define va_start(__AP,__LA) (__AP=__va_start())

#define va_arg(__AP, __TYPE) \
 (__AP = ((char *) (__AP) + __va_rounded_size (__TYPE)),     \
  *((__TYPE *)((__AP) - __va_rounded_size (__TYPE))))

#define va_end(__AP) ((__AP) = 0)
```

8.9 Known Problems

- no floating-point
- huge-pointers are sometimes derferenced as far-pointers
- addressing-modes sometimes ignore huge
- initialized data is placed in RAM, bits are not initialized
- struct-copy only works with near-pointers
- near/far-conversion assumes DPP0-DPP3 linear

9 68hc12 Backend

This chapter documents the Backend for the 68hc12 and 6809/6309 microcontroller families.

9.1 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 and increase compilation time. (No fp support yet.)

`-const-in-data`
: By default constant data will be placed in a read-only section. Using this option it will be placed in the data section.

`-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 very slightly reduce the stack size needed by the compiled program.

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

`-mem-cse` Try to hold values loaded from memory in registers and reuse them. Due to the small register-set of the hc12 this is disabled by default as it increases register-pressure and tends to spill to the stack.

`-cpu=<n>` Specify the processor family. Currently supported values are:

- 6812: Generate code for the Motorola 68hc12 series (default).
- 6809: Generate code for the Motorola 6809 series.
- 6309: Generate code for the Hitachi 6309 series (currently identical to 6809).

`-acc-glob`
: Make the accumulator available for global register allocation. By default, the accumulator will only be used for register allocation within basic blocks. In many cases, global allocation will result in worse code, because the accumulator has to be pushed/popped many times.

`-pcrel` Generate PC-relative code. May not fully work with 68hc12.

`-drel` Generate code that accesses data relative to register `u`. Does not work with 68hc12.

9.2 ABI

The current version generates assembly output for use with vasm6809_std or the GNU assembler using the non-banked model.

This backend supports the following registers:

– `d` for the accumulator (also used for byte, i.e. `b`)

- `x` for index register x
- `y` for index register y
- `u` for index register u (not on 68hc12)
- `sp` for the stack-pointer

The accumulator and `x` are caller-save. The `y` register is callee-save. The `u` register is used as data-page pointer with `-drel`.

The first function argument which has integer or pointer type up to 16 bits is passed in the accumulator `d`. The remaining arguments are passed on the stack.

Integers and pointers are returned in `d` or `d/x` (`x` contains the higher bits). 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
char	8	1
short	16	1
int	16	1
long	32	1
near pointers	16	1
far pointers	24	1 (not yet)
huge pointers	24	1 (not yet)
float	32	1 (not yet)
double	64	1 (not yet)
long double	64	1 (not yet)

9.3 Target-specific variable-attributes

The 6809/hc12-backend offers the following variable attributes:

`__interrupt`
: Return with `rti` rather than `rts`.

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

`__dpage` Place the variable in section `.dpage` and use direct addressing.

9.4 Predefined Macros

This backend defines the following macros:

`__HC12__` If code for 68hc12 is generated.

`__6809__` If code for 6809 is generated.

`__6309__` If code for 6309 is generated.

9.5 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 `y`. This function has to consider its own stacksize and must restore all registers.

9.6 Stdarg

Stdarg is supported by the provided include.

9.7 Known Problems

- Support for floating point and long long is still missing.
- U register is not really used yet.
- No support for 6309 extensions yet.
- Many optimizations still missing.
- Some code generation bugs to be fixed.

10 VideoCore IV Backend

This chapter documents the Backend for the VideoCore IV processor family.

The backend is in a very early stage, it is not complete, and it can not yet be considered useful!

Also note that it is based on freely available, unofficial, and possibly incorrect information on the target processor.

10.1 Additional options for this version

This backend provides the following additional options:

`-short-double`
: Use native 32bit floating point for double and long double. This is much more efficient, but not ISO C conforming.

`-one-section`
: Put all code and data in the same section (.text).

`-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.

`-no-peephole`
: Disable most backend peephole optimizations. Just for testing.

`-noext-regs`
: Do not use registers r16-r23. Just for testing.

`-cond-limit=<n>`
: Set the limit (in number of intermediate code instructions) for the length of code-sequences considered for conditional execution (default: 2).

10.2 ABI

This backend supports the following registers:

- `r0` through `r31` for the general purpose registers

Additionally, the register pairs `r0/r1` `r2/r3`, `r4/r5`, `r6/r7`, `r8/r9`, `r10/r11`, `r12/r13`, `r14/r15`, `r16/r17`, `r18/r19`, `r20/r21`, `r22/r23` are available.

`r14`, `r15`, `r24-r31` are currently reserved by the backend.

The current version generates assembly output for use with `vasm`.

The registers r0-r5 and r14-r15 are used as scratch registers (i.e. they can be destroyed in function calls), all other registers are preserved. r25 is the stack-pointer.

The first 6 function arguments which have integer, float32 or pointer types are passed in registers r0 through r5. All other arguments are passed on the stack.

Integers, float32 and pointers are returned in r0. 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

char                 8                      1
short               16                      2
int                 32                      4
long                32                      4
long long           64                      8    not yet supported
all pointers        32                      4
float               32                      4
double              64 (32)                 4
long double         64 (32)                 4
```

10.3 Target-specific variable-attributes

The vidcore-backend offers the following variable-attributes:

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

10.4 Target-specific pragmas

The vidcore-backend offers the following #pragmas:

`none at the moment...`

10.5 Predefined Macros

This backend defines the following macros:

`__VIDEOCORE__`
`__SHORT_DOUBLE__ (if -short-double is used)`

10.6 Stdarg

stdarg-implementation is not yet fully working. One restriction is that when calling a varargs function, the prototype must be in scope (this is ISO C conforming). Another one is that the stdarg-macros only work as long as all fixed arguments are passed in registers.

This will be fixed in the future.

10.7 Known problems

- no support for long long
- no support for 64bit floating point
- stdarg problems mentioned above

- suboptimal code quality
- ...

11 6502 Backend

This chapter documents the backend for the 6502 processor family.

11.1 Additional options

This backend provides the following additional options:

`-c02`

`-65c02` Generate code using 65C02 extensions.

`-ce02` Generate code using CSG 65CE02/4510 extensions.

`-mega65` Generate code using the MEGA65 extensions.

`-m65io` When using the MEGA65 multiplier/divider, assume that the IO area is directly accessible to generate smaller and faster code.

`-div-bug` Do not generate code using the MEGA65 divider as this is buggy in early versions.

`-std-syntax`
Generate code for the std syntax module of vasm rather than the default oldstyle syntax module.

`-cbmascii`
Convert string-constants and character-constants to CBM ASCII.

`-atascii` Convert string-constants and character-constants to Atari ASCII.

`-ieee` Use 32/64bit IEEE format for floating point rather than wozfp format.

`-softstack`
Try to use the hardware stack as little as possible. Use this if your hardware stack is overflowing.

`-mainargs`
Some targets may need special initializations when using the command line arguments to main(). vbcc will emit a call to a library function when those are used and `-mainargs` is specified.

`-no-bank-vars`
Do not automatically handle accesses to banked variables, but only function calls. See chapter on banking.

`-manual-banking`
Do not automatically handle banking. Useful if you want to use the section mapping of the `__bank`-attribute or `#pragma bank` but you prefer handle bank switching yourself.

`-avoid-bank-switch`
Prefer calling `__bankload/__bankstore` instead of `__bankswitch`. Useful for banking mechanisms that do not provide quick switching of an entire bank (e.g. the C64 REU).

`-common-banknr=<n>`
: Specify the bank number of the common bank. See chapter on banking. The default number is 0.

`-large`
: Use large memory model. All pointers will be far pointers. A corresponding library is required. This feature is experimental.

`-glob-acc`
: By default, the register allocator will only assign temporary variables to the accumulator register or `a/x` register pair. Usually this reduces necessary storing/loading of the accumulator as it is needed during most operations. This option allows the register allocator to assign variables with bigger live ranges to the accumulator. This option is likely to create worse code in most cases. Use only for experimentation.

11.2 ABI

The current version generates assembler output for use with `vasm6502_oldstyle` or `vasm6502_std`. The option `-opt-branch` is needed.

The register names provided by this backend are:

```
a, sp, r0..r31, btmp0..btmp3
```

`a` is the accumulator. It can be used for type `char`.

`r0` ... `r31` are 8bit variables that can be used for type `char`. They have to be mapped to zero page during linking. The compiler expects registers that can be used as register pairs (see below) to be mapped to contiguous memory locations. Some library routines may have additional requirements.

`sp` is a 16bit variable that has to be mapped to zero page. It is used by the compiler and not available for use. `sp` must be initialized to a suitable memory area. This stack is used for local variables, saved registers etc. The hardware stack is used mainly for return addresses and sometimes saved registers.

`btmp0`..`btmp3` are 32bit variables. The code generated by vbcc does not require them in zero page, but the current library implementation expects them to be located to contiguous zero page locations.

The following register pairs can be used for 16bit values.

```
a/x, r0/r1, r2/r3 ... r30/r31
```

`a/x` can be used for types `short` and `int`, the other register pairs can also be used for pointer types.

The following register pairs can be used for 64bit values (IEEE doubles or long long).

```
btmp0/btmp1, btmp2/btmp3
```

Registers `a, r0..r15, r28..r31, btmp0..btmp3` are volatile (i.e. they can be destroyed in function calls). `r16..r27` are preserved across function calls.

`r0..r7` are used for passing function arguments of type `char, short, int` and pointer types. `btmp0..btmp3` are used for passing arguments of type `long, long long, float, double, long double` and far pointers. All other types are passed on the stack.

For functions with a variable-argument-list, arguments that are part of the variable-argument-list are always passed on the stack. It is therefore required that a prototype is in scope when calling such functions (as required by C).

Scalar types are returned as follows:

```
type        registers

char        a
short       a/x
int         a/x
long        btmp0
long long   n/a
pointers    a/x
far-pointer btmp0
float       btmp0
double      btmp0 or btmp0/btmp1 (IEEE)
long double btmp0 or btmp0/btmp1 (IEEE)
```

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.

The basic data types are represented like this:

```
type        size in bits        alignment in bytes

char                 8                      1
short               16                      1
int                 16                      1
long                32                      1
long long           64                      1       (currently not supported)
near pointers       16                      1
far pointers        24                      1
float               32                      1       see below
double              32/64                   1       see below
long double         32/64                   1       see below
```

11.3 Math

For certain operations `vbcc` will emit calls to routines that have to be provided by a library. For integer code, the following operations are handled by library routines (some special cases involving constants may be handled by inline code).

```
__mulint16              16x16=>16 multiplication
__mulint32              32x32=>32 multiplication
__divint16              16x16=>16 signed division
__divint32              32x32=>32 signed division
__divuint16             16x16=>16 unsigned division
__divuint32             32x32=>32 unsigned division
__modint16              16x16=>16 signed modulo
__modint32              32x32=>32 signed modulo
__moduint16             16x16=>16 unsigned modulo
```

```
__moduint32                 32x32=>32 unsigned modulo
```

11.3.1 Floating Point

By default, all floating point types are implemented as 32bit values. The format used by the floating point routines published by Roy Rankin and Steve Wozniak is used. While this does work for many use cases, it is not fully C compliant by any means. Calculation of constants in the compiler is not done in that format. Therefore, the results of calculations done at compile-time may be different from those at run-time. The corresponding math library must be linked using `-lm`.

As an alternative, `vbcc` can use IEEE format by specifying `-ieee`. This will solve the problems mentioned above and provide accurate results using the SANE floating point environment. However, these routines are slower and much bigger. Also, you may have to clarify their legal status before using them. The IEEE routines will likely not work when running from ROM.

In addition to `-ieee`, the SANE library has to be linker using `-lmieee`.

When using floating point, the following library routines are needed (without `-ieee`):

```
        __addflt32                  floating point addition
        __subflt32                  floating point subtraction
        __mulflt32                  floating point multiplication
        __divflt32                  floating point division
        __negflt32                  floating point negation (-x)
        __cmpsflt32                 floating point comparison
                                    (sets a to pos., neg. or zero, depending on
                                    the comparison result)
        __sint16toflt32             convert signed 16bit integer to floating point
        __uint16toflt32             convert unsigned 16bit integer to float-
    ing point
        __sint32toflt32             convert signed 32bit integer to floating point
        __uint32toflt32             convert unsigned 32bit integer to float-
    ing point
        __flt32tosint16             convert floating point value to signed 16bit integer
        __flt32touint16             convert floating point value to unsigned 16bit integer
        __flt32tosint32             convert floating point value to signed 32bit integer
        __flt32touint32             convert floating point value to unsigned 32bit integer
```

Further math library functions may be needed by user code or the C library.

11.4 Target-Specific Variable Attributes

This backend offers the following variable attributes:

`__interrupt`
: Used for writing interrupt handlers. All used registers (including volatiles and accumulator) are saved/restored and `rti` is used to leave the function. Also, the user stack pointer is set to `__isrstack`. This value for the interrupt stack has to be provided e.g. by the linker script.

`__zpage`
: Place variable in section `zpage` and instruct `vbcc` to use it with zero-page addressing.

`__nocpr` Turn off code compression for this function even if `-size` is used. Useful for time-critical functions.

`__bank(<n>)`
Place the variable/function in bank n. See chapter on banking for details.

11.5 Target-Specific #pragmas

This backend offers the following #pragmas

`#pragma section <sec>`
The following functions and variables are placed in section `<sec>`.

`#pragma section default`
The following functions and variables are placed in default sections.

`#pragma bank <n>`
The following functions and variables are placed in bank `<n>`. See the chapter on banking for details.

`#pragma bank -1`
The following functions and variables are placed in the default bank. See the chapter on banking for details.

11.6 Predefined Macros

This backend defines the following macros:

`__6502__`

`__SIZE_T_INT`

11.7 Stack

Local variables and function arguments are put on the user stack. It can be up to 64KB, but accessing variables outside the lower 256 bytes is significantly slower. `vbcc` will put small variables on lower offsets to increase the number of variables that can be addressed fast. However, accessing the stack is always rather slow on the 6502.

The stack pointer is adjusted once during function entry/exit to avoid multiple costly adjustments to the stack pointer.

Variable-length-arrays as specified in c99 should be fully supported.

11.8 Banking

Many 6502 systems offer/need different banking mechanisms. vbcc offers different levels of support for those schemes. Depending on the target integration, more or less support is offered.

11.8.1 Manual Banking

Banking can usually be implemented manually if there is no suitable target integration or if manual optimization is preferred.

For manual banking, functions and/or data can be mapped to sections using the `section` attribute or the `#pragma section` directive. Note that string constants will be mapped to banked memory only when using the `#pragma` approach.

After placing the objects in suitable sections, they have to be located using a linker command file. See the documentation on vlink. Switching between banks has to done manually in a system-specific way.

If initialized variables are mapped into banked sections, be aware that you may have to provide means to initialize them on startup. It is probably not handled by startup code for non-banked systems. This is not relevant for systems that only provide banked ROM.

11.8.2 Automated Banking

If a suitable target integration is available, vbcc is able to automatically handle bank switching.

11.8.2.1 Memory Model

vbcc assumes a memory model which provides non-banked memory for code and data that is always visible. Additionally there can be a number of up to 255 memory banks, one of which can be visible at a time.

If several banks can be visible at the same time, this will also work, but vbcc does not make use of this feature.

11.8.2.2 Mapping

Code/data can be put into banks using the `bank`-attribute or `#pragma bank`. Note that string constants will be mapped to banked memory only when using the `#pragma` approach. Code/data not mapped to a bank will be mapped to the always visible non-banked area.

Banked objects will be mapped to sections suffixed with the bank number, e.g. in

```
__bank(0) int i = 1;
__bank(1) void f()
```

`i` will be put into section `data0` and `f` will be put into section `text1`.

11.8.2.3 Bank Switching

vbcc will try to determine when a bank switch has to be made and it will call library functions to map in the required data. This requires that when accessing an object, vbcc has to know which bank the object is assigned to. Therefore a declaration specifying the correct bank number (either using the `__bank()`-attribute or `#pragma bank`). If an object is accessed which has not been declared with a bank number, it is assumed that it can be accessed without bank switching.

11.8.2.4 Far Pointers

When accessing an object through a normal pointer, the bank number is not known. All accesses through normal pointers assume that the target is always visible. For all other cases, far-ointers have to be used. Far-pointers contain the bank number as additional information, resulting in a size of 3 bytes.

They are declared using the `__far`-qualifier. `__far` is used like a type qualifier, similar to `const`.

```
__bank(0) i0;
__bank(1) i1;
__far int *fp;
...

if(..)
  fp = &i0;
else
  fp = &i1;
```

Converting a far-pointer to a normal pointer will loose the bank information. Converting a normal pointer to a far-pointer will insert the bank number of the current function. Care must be taken not to loose bank information when working with pointers.

11.8.2.5 Performance Considerations

Accessing objects through bank switching generates much slower and larger code than direct accesses. Therefore it is crucial to organize your objects in a way that reduces task switches as much as possible. Following is a list of hints and explanations.

- Using far pointers will always incur overhead. Try to use them only when necessary.
- Accessing objects from the non-banked area is always fast (unless done through far-pointers).
- Accessing objects from the same bank the function is mapped to is usually fast (unless done through far-pointers).
- Accessing banked objects from non-banked code is usually faster than accessing them from banked code in another bank.
- Calling functions in another bank is a reasonably small overhead on systems with a fast bank switch. It can be much more overhead on RAM expansions that have to copy code.
- Be careful when using function inlining. Inlined code will be executed in the bank of the caller.

In general, try to reduce cross-bank accesses and far-pointer usage as much as possible. For best performance you can always only use the `section`-features to map everything and handle all bank-switching yourself.

11.8.2.6 Library

To use the automated bank switching, a series of library functions must be available (TODO: add more detailed information, may change):

`___bankswitch`
: Make the bank in `y` accessible.

`___bankjsr`
: Call the function pointer `r30/r31` in bank `y`. Return to the caller in bank `a`.

`___bankload<n> (n=1,2,3,4,8)`
: Copy `<n>` bytes from `r30/r31` in bank `y` to non-banked variable `___bankv` with offset `x`. Caller bank in `a`.

`___bankstore<n> (n=1,2,3,4,8)`
: Copy `<n>` bytes from non-banked variable `___bankv` with offset `x` to `r30/r31` in bank `y`. Caller bank in `a`.

`___bankcopy`
: Copy `___bankcopy_len/___bankcopy_len+1` bytes from `r28/r29` in bank `y` to `r30/r31` in bank `x`. Caller bank in `a`.

11.9 Debugging

The 6502 backend has some limited support for debugging.

When using vlink, the `-vicelabels` options can be used to output symbol values in a format that can be read by the vice emulator/debugger.

With the `-g` option, line numbers and file names of source code will be added to the assembly output. Using some tools, it should be possible to create a mixed C/assembly file for inspection. Depending on the optimization level, the results may be more or less usable, see section Debugging Optimized Code. Note that the added comments will affect the assembly peephole optimizer, resulting in worse code than without `-g`.

11.10 Code compressor

The `vcpr6502` code compressor supports the 6502 target. It should be called automatically on higher optimization levels when using the frontend `vc`. As the code will be slowed down when using compression, the 6502 backend will only enable compression when using the `-size` option.

TODO: manual overriding

11.11 Known problems

- `long long` not yet supported
- return value of cross-bank function calls may not work
- banking not tested very much

12 Instruction Scheduler

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

12.1 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.

12.2 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.

12.3 Known problems

– works only on basic-blocks

13 Code Compressor

vcpr - code compressor for vbcc (c) in 2020 by Volker Barthelmann

13.1 Introduction

vcpr is a code compressor which scans the assembly output of vbcc and tries to reduce code size of the generated code by moving common code sequences into separate subroutines. As a trade-off, the code will usually run slower.

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 compressor.

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

13.2 Usage

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

```
vcpr [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. Assembly code that was not generated by `vbcc` may not work correctly after running it through `vcpr`.

13.3 Known problems

– still in early development

13.4 Backend Interface

13.4.1 Building `vcpr`

To write a backend for `vcpr`, the file `compress.c` has to be created in the machine subdirectory. The executable `vcpr<target>` can be built using:

```
make TARGET=<target> bin/vcpr<target>
```

13.4.2 Basic Function

`vcpr` first reads the assembly source into a linked list of lines. It will call a backend function for each line to parse the line and fill in necessary information.

The frontend looks for identical code sequences and calculates the savings that can be obtained by outlining the code. If code sequences are found that provide enough savings, they will be moved to subroutines using functions provided by the backend.

Lines have to be textually identical to be considered for outlining. One exception are labels. The following sequences are considered equal as long as the labels are not used anywhere else:

```
  a
  bne l1
  b
l1:
  c

  a
  bne l2
  b
l2:
  c
```

Currently, `vcpr` only supports code sequences with one label.

13.4.3 Data Types

The main data type relevant for the backend represents the attributed source lines:

```
typedef struct line {
  ...
  char *code;
  int flags;
  int size;
  ...
  int l1,l2;
} line;
```

The following members are relevant to the backend:

`code` points to the assembly text of the line.

`size` contains the (estimated) size of the instruction.

`l1, l2` specify up to two labels referenced by this instruction.

`flags` specifies the type of instruction and can be a combination of:

- `LABDEF`: This line defines a label.
- `LABUSE`: This line references a label.
- `BARRIER`: This line must not be moved into a subroutine.
- `LBARRIER`: Used by the frontend.

13.4.4 Backend Variables

The following variables have to be defined and initialized by the backend:

- `const char tg_copyright[]` A copyright string.
- `int minsave;` The minimum size units saved by outlining a function. Should at least be larger than the size of the code for a subroutine call.

13.4.5 Backend Functions

The following functions have to be provided by the backend:

- `void parse_line(char *s,line *p);`

  This function parses assembly line `s` and has to fill the members `flags`, `l1`, `l2` and `size` of the line structure `p`.

- `add_header(line *after);`

  This function creates a line for the header of outlined code and inserts it into the line list.

- `add_ret(line *after);`

  This function adds a return line and inserts it into the line list.

- `add_jsr(line *after);`

  This function adds a subroutine call to the subroutine corresponding to the last call of `add_header()`.

13.4.6 Frontend Functions

The following functions are provided by the frontend and can be used by the backend:

- `void *mymalloc(size_t sz);`

  Allocate memory.

- `int new_label(void);`

  Create a new unused label number.

- `line *new_line(void);`

  Create a new line.

- `void insert_line(line *after, line *new);`

  Insert a line into the line list at a specified position.

14 C Library

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

14.1 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 part of the new functions from ISO9899:1999.

14.2 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.

Parts of the math library (e.g. transcendental functions) are derived from Sun's free math library:

```
* ====================================================
* Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
*
* Developed at SunPro, a Sun Microsystems, Inc. business.
* Permission to use, copy, modify, and distribute this
* software is freely granted, provided that this notice
* is preserved.
* ====================================================
```

The softfloat functions, used by some targets, are derived from John Hauser's IEC/IEEE Floating-point Artithmetic Package:

```
This C source file is part of the SoftFloat IEC/IEEE Floating-point
```

```
Arithmetic Package, Release 2.

Written by John R. Hauser.  This work was made possible in part by the
International Computer Science Institute, located at Suite 600, 1947 Center
Street, Berkeley, California 94704.  Funding was partially provided by the
National Science Foundation under grant MIP-9311980.  The original version
of this code was written as part of a project to build a fixed-point vector
processor in collaboration with the University of California at Berkeley,
overseen by Profs. Nelson Morgan and John Wawrzynek.  More information
is available through the web page `http://HTTP.CS.Berkeley.EDU/~jhauser/
arithmetic/softfloat.html'.

THIS SOFTWARE IS DISTRIBUTED AS IS, FOR FREE.  Although reasonable effort
has been made to avoid it, THIS SOFTWARE MAY CONTAIN FAULTS THAT WILL AT
TIMES RESULT IN INCORRECT BEHAVIOR.  USE OF THIS SOFTWARE IS RESTRICTED TO
PERSONS AND ORGANIZATIONS WHO CAN AND WILL TAKE FULL RESPONSIBILITY FOR ANY
AND ALL LOSSES, COSTS, OR OTHER PROBLEMS ARISING FROM ITS USE.

Derivative works are acceptable, even for commercial purposes, so long as
(1) they include prominent notice that the work is derivative, and (2) they
include prominent notice akin to these three paragraphs for those parts of
this code that are retained.
```

14.3 Global Variables

14.3.1 timezone

On some host operating systems vclib might be unable to determine the current time zone, which is required for functions like `mktime()` or `localtime()` to work. Here you can overwrite the following variables:

`long __gmtoffset`
: Offset in minutes, west of Greenwich Mean Time (GMT).

`int __dstflag`
: Set to non-zero, when Daylight Saving Time is active.

Targets which can determine their current time zone, will do so by initializing these variables on startup.

14.4 Embedded Systems

This section describes specifics of the C library for embedded systems.

14.4.1 Startup

The startup is usually split into two parts. The first part is done by assembly code that produces the object file `lib/startup.o`. This assembly code is usually provided with vbcc and may have to be adapted to the hardware you are using. The key actions that have to be performed by this code are:

−hardware initialization
: It may be necessary to perform some hardware initialization right at the beginning, e.g. to configure the memory system. This has to be modified by the user.

−variable initializations
: When running code from ROM, some memory sections have to be initialized. Usually, the init-values of initialized variables have to be copied from ROM to the data segment and the values of un-initialized variables have to be cleared in the bss segment. This code is usually provided in the startup code.

−stack pointer
: The stack pointer has to be set to a suitable memory area. The startup code will set the stack pointer to the value of the pointer `__stack`. There is a default stack provided in the C library which will be used unless the application defines its own stack using, for example, the following code (assuming that the stack grows downwards):

```
#define STACKSIZE <whatever>

static long mystack[STACKSIZE/sizeof(long)];
char *__stack=((char*)mystack)+STACKSIZE;
```

−calling `__main`
: After all the above initializations have been performed, the function `__main()` has to be called. This function is provided by the library and performs high-level initializations, if necessary (mainly it calls constructors created by the linker) and will then call the user `main()` function. Note that the library may not work correctly if the user `main()` function is called directly from the startup code.

14.4.2 Heap

When dynamic memory management is used (e.g. by using the `malloc()` function), a heap memory area is needed to allocate memory from. The `malloc()` function assumes that `__heapptr` is a variable pointing to the beginning of the heap memory and that `__heapsize` specifies the size of the heap area in bytes. The library will provide a default heap memory area that can be replaced by adding, for example, the following file to the application:

```
#define HEAPSIZE <whatever>

char __heap[HEAPSIZE],*__heapptr=__heap;
size_t __heapsize=HEAPSIZE;
```

14.4.3 Input/Output

The standard C input/output functions are provided also for embedded systems. Reading/writing to a stream will be directed to void unless the following low-level I/O-functions are provided by the application:

```
int __open(const char *name,const char *mode);
void __close(int h);
size_t __read(int h,char *p,size_t l);
```

```
size_t __write(int h,const char *p,size_t l);
off_t __seek(int h,off_t offset,int origin);
```

The `__open()` function receives a name and a mode string (as in the C `fopen()` function) as arguments and has to return a file-descriptor if it is possible to open this file. The other functions are equivalent to the corresponding POSIX functions. `__seek` can be implemented to return `-1` if the functionality is not needed.

Also, `stdin, stdout` and `stderr` can be used with the standard descriptors.

14.4.4 CTRL-C Handling

Some targets implement handlers to terminate the program on ctrl-c or a similar signal. This usually has the same effect as calling `exit(20)`.

If you want to change or disable the ctrl-c handling, you can overwrite the function `void _chkabort(void)`.

14.4.5 Floating Point

Whether floating point is supported, depends on the target architecture and chip. If it is supported, there will usually be a math-library that has to be linked (using option `-lm`) when floating point is used.

14.4.6 Useless Functions

Of course, some of the C library functions can not be implemented reasonably on embedded systems. These functions are contained in the library but will always return an error value. Mainly affected are:

−locale

−time

−signal

−filesystem functions

Depending on the hardware provided by a system it is possible to implement these functions and add them to the application. In this case, the new functions will be used rather than the default ones returning only error values.

14.4.7 Linking/Locating

To produce ROM images (e.g. in the form of absolute ELF executables, Intel Hex files or Motorola S-Records), the linker is called with a linker script. This script can be used to join together different sections of the input files and locate them to suitable absolute memory areas. Also, this linker script can be used to set symbols that may be used by the application or the startup code, e.g. addresses of data sections, initialization values or small data pointers.

Code or data that has to reside at special locations can be put into a special section using the `__section` attribute. This section can then be placed at the desired location using the linker script.

Usually, an example linker script will be provided. While this is often not suitable for different chips, it may serve as a starting point.

14.5 AmigaOS/68k

This section describes specifics of the C library for AmigaOS/68k provided by the target `m68k-amigaos`. The relevant files are `startup.o`, `minstart.o`, `minres.o`, `vc.lib`, `vcs.lib`, `mieee.lib`, `mieees.lib`, `m881.lib`, `m881s.lib`, `m040.lib`, `m040s.lib`, `m060.lib`, `m060s.lib`, `msoft.lib`, `msofts.lib`, `amiga.lib`, `amigas.lib`, `auto.lib` and `autos.lib`, `reaction.lib`, `reactions.lib`.

Note that `extra.lib` is no longer part of the vbcc distribution. It was replaced by 'PosixLib', available on Aminet `dev/c/vbcc_PosixLib.lha`, which has a much more comprehensive support for POSIX and Unix functions.

The following config files are available:

`aos68k` Standard startup code (`startup.o`) with command line parsing and optional Workbench startup (See Section 14.5.1 [Standard Startup], page 91).

`aos68km` Minimal startup code (`minstart.o`) without command line parsing. You have to open all libraries yourself (See Section 14.5.6 [Minimal Startup], page 93).

`aos68kr` Minimal startup code (`minres.o`) for resident programs. Always compiles in small data mode and links with `vcs.lib` (See Section 14.5.7 [Minimal Resident], page 94).

14.5.1 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.

14.5.2 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. At the moment there are five 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. By default all floating point routines are provided via stub functions in `mieee.lib`. With the option `-amiga-softfloat` you can tell `vbccm68k` to generate inline code for calling the MathIeee libraries directly.

`msoft.lib`

This one is based on John Hauser's IEC/IEEE Floating-point Arithmetic Package (See Section 14.3 [SoftfloatHauser], page 88) and doesn't need any system libraries for FP emulation. May be slower than the ROM libraries, though. Otherwise everything mentioned for `mieee.lib` applies here as well. Note that you have to call the `vc` frontend with the `-rmcfg-amiga-softfloat` option,

when your config file contains `-amiga-softfloat` (which is the case for `aos68k` since vbcc V0.9h).

`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` or `-fpu=68882`. 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 68040. Other functions use the Motorola emulation routines modified by Aki M Laukkanen and Matthew Hey. It should be used for programs compiled for 68040 with FPU. Return values are passed in fp0.

`m060.lib` This one uses only direct FPU instructions that do not have to be emulated on a 68060. Other functions use the Motorola emulation routines modified by Aki M Laukkanen and Matthew Hey. It should be used for programs compiled for 68060 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.

14.5.3 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.

14.5.4 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.

14.5.5 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.

14.5.6 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. You can also achieve that by simply using the `aos68km` config file instead.

This startup code does not set up everything needed by vc.lib, so you must not use most of these functions (string and ctype functions 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` is 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 or `-amiga-softfloat` 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 <proto/exec.h>
#include <proto/dos.h>

struct DosLibrary *DOSBase;

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

```
}
```

This can yield an executable of under 256 bytes when compiled with `-sc -sd` and linked with `minstart.o` and `amigas.lib` (using `vlink` - may not work with other linkers).

14.5.7 Minimal Startup for resident programs

AmigaOS can keep special "pure" programs resident in RAM, and restart them from there without having to load them again from disk. To make it easy to create such reentrant programs, even with static data, you can link with the special startup code `minres.o`, which is a minimal startup code for resident programs. Or simply use the config file `aos68kr` instead. Everything mentioned for `minstart.o` in the previous section is also valid for `minres.o`.

To create real resident programs you have to follow the following rules:

- Compile all your code for the small data model (`-sd` option).
- Avoid absolute references to small data symbols. Usually these are constant pointers to static data. The following example creates such an illegal relocation:

  ```
  int x;
  int const *p = &x;
  ```

  `vlink` warns about all potential problems.
- Link with the `minres.o` startup code, and use the small data versions of linker libraries (`vcs.lib`, `amigas.lib`, etc.).
- Set the Pure flag in the file attributes. Load the program into RAM with the AmigaDOS `resident` command.

14.5.8 amiga.lib

To write programs using 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 for the small data mode as well. 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.

Besides some support functions `amiga.lib` contains stub routines to call functions from all common AmigaOS libraries with stack arguments. By including the library's proto header file you make sure that AmigaOS functions are called directly by inline code, unless `_NO_INLINE` is defined.

Preprocessor defines to control the behaviour of vbcc's proto headers:

`__NOLIBBASE__`
: Do not declare the library base symbol.

`_NO_INLINE`
: Do not use optimized inline code for library function calls.

Note that the OS-call inlines have been generated using the NDK3.2 clib header files, while trying to keep compatibility to NDK3.9, so it is advised to use one of the two NDKs for development. Otherwise you will get warnings about missing `CONST` typedefs and similar.

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

14.5.9 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 without having 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 `auto.lib` depends on constructor/destructor handling in vclib, which means it cannot work when linking without vclib, without standard startup code, or only with a minimal startup code, like `minstart.o`.

14.5.10 reaction.lib

The `reaction.lib` in `vbcc` is a port of Stephan Rupprecht's rewrite of the copyrighted linker library, extended and fixed by Olaf Barthel for the NDK 3.2 release. This version should work in combination with NDK 3.9 as well.

To link with `reaction.lib` (or the small data version `reactions.lib`) specify the `-lreaction` or `-lreactions` option to `vc`.

The library contains ReAction GUI class support functions and their autoinitialization code. Refer to `reaction_lib.doc` from your NDK Autodocs for more information. As documented there, the version used to automatically open the classes can be defined by the variable `__reactionversion` with external linkage. Otherwise a default of version 0 is used.

14.6 Kickstart1.x/68k

This section describes specifics of the C library for Amiga Kickstart 1.2 and 1.3 provided by the target `m68k-kick13`. The relevant files are `startup.o`, `minstart.o`, `minres.o`, `startup16.o`, `minstart16.o`, `minres16.o`, `vc.lib`, `vcs.lib`, `m13.lib`, `m13s.lib`, `msoft.lib`, `msofts.lib`, `m881.lib`, `m881s.lib`, `amiga.lib`, `amigas.lib`, `auto.lib` and `autos.lib`, `vc16.lib`, `vc16s.lib`, `m1316.lib`, `m1316s.lib`, `msoft16.lib`, `msoft16s.lib`, `m88116.lib`, `m88116s.lib`, `amiga16.lib`, `amiga16s.lib`, `auto16.lib` and `auto16s.lib`.

This target makes it possible to develop programs targeted for these older versions of the Amiga operating system, using the original Commodore Kickstart 1.3 header files. Note that there are also libraries and config files for using a 16-bit int ABI, which was common at that time, and may have some advantages on 16-bit CPUs, like the 68000 or 68010.

The following config files are available:

`kick13` Standard startup code (`startup.o`) with command line parsing and optional Workbench startup (See Section 14.6.1 [Startup13], page 96) using 32-bit int.

`kick13m` Minimal startup code (`minstart.o`) without command line parsing. You have to open all libraries yourself (See Section 14.5.6 [Minimal Startup], page 93) using 32-bit int.

`kick13r` Minimal startup code (`minres.o`) for resident programs. Always compiles in small data mode and links with `vcs.lib` (See Section 14.5.7 [Minimal Resident], page 94) using 32-bit int.

`kick13s` Standard startup code (`startup.o`) with command line parsing and optional Workbench startup (See Section 14.6.1 [Startup13], page 96) using 16-bit int.

`kick13sm` Minimal startup code (`minstart.o`) without command line parsing. You have to open all libraries yourself (See Section 14.5.6 [Minimal Startup], page 93) using 16-bit int.

`kick13sr` Minimal startup code (`minres.o`) for resident programs. Always compiles in small data mode and links with `vcs.lib` (See Section 14.5.7 [Minimal Resident], page 94) using 16-bit int.

14.6.1 Startup

The startup code currently consists of a slightly modified standard AmigaOS 1.3 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.

14.6.2 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. At the moment there are two math libraries:

`m13.lib` This one uses the C= math libraries present under Kickstart 1.2 and 1.3. The startup code will always open mathffp.library, MathIeeeDoubBas.library and MathIeeeDoubTrans.library. Note that all single precision floating point calculations take place in FFP format and have to be converted between FFP and IEEE by the 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.

`msoft.lib`
This one is based on John Hauser's IEC/IEEE Floating-point Arithmetic Package (See Section 14.3 [SoftfloatHauser], page 88) and doesn't need any system libraries for FP emulation. May be slower than the ROM libraries, though. Otherwise everything mentioned for `m13.lib` applies here as well.

`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.

14.6.3 Stack

Stack-checking is available similar to AmigaOS/68k (See Section 14.5.3 [amiga-stack], page 92). But there is no automatic stack-extension under Kickstart 1.3 and a `__stack` variable will be ignored.

14.6.4 Small data model

Small data is supported as described for AmigaOS/68k (See Section 14.5.4 [amigasmalldata], page 93). The startup code takes care of clearing the unititalized part of a small data section (which Kickstart 1.x fails to do).

14.6.5 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 AmigaOS it is not possible to obtain the time used by the calling process.

14.6.6 amiga.lib

See Section 14.5.8 [amigalib], page 94.

This version of `amiga.lib` only supports the functionality present in Kickstart 1.2/1.3.

14.6.7 auto.lib

This library corresponds to the AmigaOS/68k version (See Section 14.5.9 [autolib], page 95), but only supports libraries of Kickstart 1.3.

14.6.8 Minimal Startup

You can use `minstart.o` similar to AmigaOS/68k (See Section 14.5.6 [Minimal Startup], page 93).

14.6.9 Minimal Startup for Resident Programs

You can use `minres.o` similar to AmigaOS/68k (See Section 14.5.7 [Minimal Resident], page 94).

14.7 PowerUp/PPC

This section describes specifics of the C library for PowerUp/PPC provided by the target `ppc-powerup`. The relevant files are `startup.o`, `minstart.o`, `libvc.a`, `libvcs.a`, `libm.a`, `libms.a` `libamiga.a`, `libamigas.a`, `libauto.a` and `libautos.a`.

Note that `libextra.a` is no longer part of the vbcc distribution. It was replaced by 'PosixLib', available on Aminet `dev/c/vbcc_PosixLib.lha`, which has a much more comprehensive support for POSIX and Unix functions.

14.7.1 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.

14.7.2 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.

14.7.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 14.5.3 [amiga-stack], page 92). 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.

14.7.4 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.

14.7.5 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.

14.7.6 Minimal Startup

The provided minimal startup code (`minstart.o`) is used similarly like the one for 68k (See Section 14.5.6 [Minimal Startup], page 93). Only use it if you know what you are doing.

14.7.7 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`.

14.7.8 libauto.a

This library corresponds to the AmigaOS/68k version (See Section 14.5.9 [autolib], page 95).

14.8 WarpOS/PPC

This section describes specifics of the C library for WarpOS/PPC provided by the target `ppc-warpos`. The relevant files are `startup.o`, `vc.lib`, `m.lib`, `amiga.lib` and `auto.lib`.

Note that `extra.lib` is no longer part of the vbcc distribution. It was replaced by 'PosixLib', available on Aminet `dev/c/vbcc_PosixLib.lha`, which has a much more comprehensive support for POSIX and Unix functions.

14.8.1 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.

14.8.2 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.

14.8.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 14.5.3 [amiga-stack], page 92).

14.8.4 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.

14.8.5 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`.

14.8.6 auto.lib

This library corresponds to the AmigaOS/68k version (See Section 14.5.9 [autolib], page 95).

14.9 MorphOS/PPC

This section describes specifics of the C library for MorphOS/PPC provided by the target `ppc-morphos`. The relevant files are `startup.o`, `minstart.o`, `libvc.a`, `libvcs.a`, `libm.a`, `libms.a` `libamiga.a`, `libamigas.a`, `libauto.a` and `libautos.a`.

Note that `libextra.a` is no longer part of the vbcc distribution. It was replaced by 'PosixLib', available on Aminet `dev/c/vbcc_PosixLib.lha`, which has a much more comprehensive support for POSIX and Unix functions.

14.9.1 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.

14.9.2 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.

14.9.3 Stack

Stack-handling is similar to AmigaOS/68k (See Section 14.5.3 [amiga-stack], page 92).

14.9.4 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.

14.9.5 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.

14.9.6 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`.

14.9.7 libauto.a

This library corresponds to the AmigaOS/68k version (See Section 14.5.9 [autolib], page 95).

14.10 AmigaOS4/PPC

This section describes specifics of the C library for AmigaOS4/PPC provided by the target `ppc-amigaos`. The relevant files are `startup.o`, `minstart.o`, `libvc.a`, `libvcs.a`, `libm.a`, `libms.a` `libamiga.a`, `libamigas.a`, `libauto.a` and `libautos.a`.

Note that `libextra.a` is no longer part of the vbcc distribution. It was replaced by 'PosixLib', available on Aminet `dev/c/vbcc_PosixLib.lha`, which has a much more comprehensive support for POSIX and Unix functions.

14.10.1 Startup

The startup code `startup.o` sets up some global variables and initializes stdin, stdout and stderr. Then it runs all constructors of dynamically linked libraries, before entering the main program. The exit code runs all destructors of dynamically linked libraries, 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.

14.10.2 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.

14.10.3 Stack

There is no automatic stack extension for AmigaOS 4! This should be done automatically by the operating system.

14.10.4 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.

14.10.5 Dynamic linking

Since `elf.library V52.2` AmigaOS4 supports dynamic linking with shared object files (`.so` extension), similar to Unix. The default behaviour is to prefer linking against a shared object over a static library. To force static linking you might want to give the `-static` option to the `vc` frontend.

14.10.6 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.

`Small data in dynamically linked executables`
: There is a bug in `elf.library V52.4` (and earlier), which doesn't load `.sdata` and `.sbss` as a contiguous block into memory, when the executable requires dynamic linking. I decided against writing a workaround, as the bug should be fixed in OS4.

14.10.7 libamiga.a

In contrast to other amigalibs the OS4 `libamiga.a` doesn't contain any stubs for calling system functions. AmigaOS 4 system calls are done through special macros in the SDK's interface header files.

The library only includes some remaining amigalib functions, not already integrated into the OS, like `CreateIO()`, but its use is discouraged.

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

14.10.8 libauto.a

Auto-open -close functions for the following libraries are included:

`Asl, CyberGfx, DataTypes, Dos, GadTools, Graphics, Icon, IFFParse, Intuition, Locale, LowLevel, Picasso96, BSDSocket, Utility, Workbench`

Note that gcc's `libauto.a` doesn't include CyberGfx.

14.10.9 newlib

14.10.9.1 Introduction

newlib.library is a shared AmigaOS4 library, which is covered by several BSD like licenses, and includes standard ANSI and POSIX functions as well as some functions common in Unix, BSD and similar operating systems. It is part of the OS4 SDK.

The config file `newlib` will be created on installation to use the paths for header files and libraries pointing to the newlib from the SDK.

What are the main differences between vclib and newlib?

- vclib contains (almost) only standard ANSI functions. If you want to port Unix programs you will probably miss a lot of functions. Also newlib supports things like mapping Unix directory paths to Amiga paths or expanding wildcards in command lines automatically.
- Programs compiled for newlib will be shorter because the code for all functions is not contained in the executable itself.
- Programs compiled for newlib will need the shared object `libc.so` present when started.
- Programs compiled for newlib will probably need more memory because the entire (rather large) `libc.so` will be loaded into memory. With vclib only the functions your program uses will be in RAM. However if you have several programs using newlib at the same time only one copy of `libc.so` should be loaded.

Things you should note:

- With newlib you do not need extra math-libraries.
- You must link with a vbcc-specific `startup.o` from the newlib `lib/` directory as startup code. The config-file `newlib` will usually take care of this.
- You _must_ use the newlib-includes from the SDK rather than the ones which are for vc.lib. The config-file `newlib` will usually take care of this.
- There may be vbcc-related bugs in the SDK-newlib. Patches are automatically installed when using the Amiga Installer. When installing the target manually, you also have to fix the SDK manually. For a list of known SDK bugs at this point of time, See Section 14.10.9.2 [Known Newlib Bugs], page 103.

14.10.9.2 Known Newlib Bugs

- The `__asm_toupper()` and `__asm_tolower()` assembler inlines in `newlib/include/ctype.h` are wrong, which makes `toupper()` and `tolower()` fail when including `ctype.h`. Fix:

```
--- ctype.h.orig 2006-04-03 18:00:00.000000000 +0200
+++ ctype.h 2017-05-07 19:32:00.000000000 +0200
@@ -64,8 +64,8 @@
 #elif defined(__VBCC__)
 int __asm_toupper(__reg("r3") int) =
         "\t.extern\t__ctype_ptr\n"
-        "\tlis\t11,(__ctype_ptr)@ha\n"
-        "\taddi\t11,11,(__ctype_ptr)@l\n"
+        "\tlis\t11,__ctype_ptr@ha\n"
```

```
+         "\tlwz\t11,11,__ctype_ptr@l(11)\n"
          "\tlbzx\t12,11,3\n"
          "\tandi.\t12,12,2\n"
          "\tbeq\t$+8\n"
@@ -73,8 +73,8 @@
          "#barrier";
 int __asm_tolower(__reg("r3") int) =
          "\t.extern\t__ctype_ptr\n"
-         "\tlis\t11,(__ctype_ptr)@ha\n"
-         "\taddi\t11,11,(__ctype_ptr)@l\n"
+         "\tlis\t11,__ctype_ptr@ha\n"
+         "\tlwz\t11,11,__ctype_ptr@l(11)\n"
          "\tlbzx\t12,11,3\n"
          "\tandi.\t12,12,1\n"
          "\tbeq\t$+8\n"
```

14.10.9.3 Usage

To compile a program to use newlib for OS4 you must make sure the proper config-file (`newlib`) is used, e.g.

```
vc +newlib hello.c
```

With a new SDK this will usually generate a dynamically linked executable, which requires `libc.so`. To force a statically linked executable:

```
vc +newlib -static hello.c
```

14.11 Atari TOS/MiNT

This section describes specifics of the C library for Atari TOS and MiNT. M680x0 processors are supported by the target `m68k-atari`, while ColdFire processors are supported by the target `cf-atari`. Both share the same startup-code and are based on common library sources and header files. Executables linked with this C library run on plain TOS as well as on MiNT, without modifications.

The relevant files are `startup.o`, `minstart.o`, `libvc.a`, `libm.a`, `libgem.a`. For the M68k target there are also math libs with FPU support (`libm881.a`, `libm040.a` and `libm060.a`) and 16-bit integer versions of all libraries (`lib*16.a`).

The following config files are available:

`tos` M68k 32-bit `int` for classic TOS machines.

`tos16` M68k 16-bit `int` for classic TOS machines.

`mint` M68k 32-bit `int` for MiNT. Also works on classic machines, but uses an embedded a.out header for MiNT, includes a changeable `__stksize` and sets the FastLoad, FastRAM and FastAlloc flags in the header.

`mintcf` ColdFire 32-bit `int`. Otherwise same as `mint`.

14.11.1 Startup

The startup code `startup.o` sets up some global variables and initializes stdin, stdout and stderr and returns the unneeded memory to the system. The exit code closes all open files and frees all memory.

14.11.2 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.

On the M68k target you have the option to enable FPU support with the `-fpu` option and choose the appropriate math library (See Section 14.5.2 [Floating point], page 91). Otherwise, there is a soft-float library, which is compatible with all the Atari models without an FPU.

14.11.3 Stack

The default stack size is 64k. There is a MiNT tool called `stack` which can adjust the stack size of an executable to any value, by looking for a symbol named `__stksize` (defined by vclib's startup code).

Additionally the required stack size can be specified by defining a variable `__stack` (of type `size_t`) with external linkage, as in other vbcc targets.

14.11.4 16-bit integer model

The default libraries use 32-bit `int` types, but you may want to use 16-bit `int` types for compatibility reasons. In this case you have to specify the config file `tos16` and link with the appropriate 16-bit libraries (which have a '16' attached to their name).

To compile and link a program using 16-bit integers a command like

```
vc +tos16 test.c -o test -lm16 -lvc16
```

may be used. There are no 16-bit versions for ColdFire targets, because this is strictly a 32-bit CPU.

14.11.5 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 neither under TOS nor under MiNT it is possible to obtain the time used by the calling process.

14.12 VideoCore/Linux

This section describes specifics of the C library for VideoCore under Linux provided by the target `vidcore-linux`.

The relevant files are `vcload`, `startup.o`, `libvc.a`, `libm.a`, `libms.a`.

The config file `vc4-linux` is part of the library.

14.12.1 Startup

The startup code `startup.o` sets up stack and heap and provides a function `__armcall()` to transfer control to the loader on the ARM side. The startup process calls constructors to set up some global variables and initialize stdin, stdout and stderr if needed.

14.12.2 Floating point

Note that you have to link with a math library if you want to use floating point operations that are not natively implemented. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math libraries only.

14.12.3 Stack

The library contains a default stack of 32KB. If another size is needed, you can add the following to your project:

```
        .align  4
        .space  <desired-size, suitably aligned>
___stackend:
        .global ___stackend
```

14.12.4 Heap

Currently, a global variable of 16KB is used to get memory for malloc() etc. If another size is needed, you can add the following to your project:

```
#define HEAPSIZE <desired size>

char __heap[HEAPSIZE],*__heapptr=__heap;
size_t __heapsize=HEAPSIZE;
```

Note that this mechanism will likely be changed in the future!

14.12.5 System Calls

To access system functions from the VideoCore-side, the function `__armcall()` can be used. It will save the current context and return to the loader. Registers `r0-r5` (the function arguments) will be saved and are available to the loader. The loader can then execute the system call and resume execution, passing the return value of the system function.

Resuming is done by calling the image with offset 2.

This functionality can also be used for debugging purposes.

14.12.6 Loader

A loader is required to execute VideoCore code from the ARM side. For standalone VideoCore code, the provided loader can be used. Usually, it will be necessary to adapt the loader to communicate between ARM and VideoCore side during runtime.

14.12.6.1 Object Format

Currently, the loader loads an simple binary image that must be pc-relative and located to address 0x00000000. Additionally, if present, a file with extension `.reltext` will be loaded for some limited relocation. This file contains a 32bit word containing the number

of relocations followed by n 32bit words containing an offset. For each offset, the address will be relocated to the image load address.

14.12.6.2 Command line arguments

`vcload [-debug] [-cache] [-offset] <image-name>`

The loader currently has the following options:

`-debug`

The loader will enter debug mode (see below).

`-cache`

The loader will set the LSB in the start address when executing code. This is supposed to inhibit a cache flush.

Just for testing!

`-offset`

The loader will allocate 1 KB more memory than required and leaves this space unused at the beginning of the allocated memory.

Just for testing!

14.12.6.3 Debug Mode

In debug mode, the loader will wait for user input before starting the VideoCore code as well as after every `__armcall`.

The following commands are available:

`w <addr> [<num>]`

Display <num> 32bit words starting at <addr>. <addr> must be the offset into the image. If <num> is omitted, one unit is displayed.

If one word is displayed, it is additionally displayed translated as an offset into the image.

`h <addr> [<num>]`

Display <num> 16bit halfwords starting at <addr>. <addr> must be the offset into the image. If <num> is omitted, one unit is displayed.

`b <addr> [<num>]`

Display <num> 8bit bytes starting at <addr>. <addr> must be the offset into the image. If <num> is omitted, one unit is displayed.

`c` Start/continue execution.

`q` Quit.

`bp <addr>` Set a breakpoint at <addr>.

This is currently a very crude implementation. It will just write a branch to `__armcall()` to <addr>. If everything works well, you will end in the debugger if <addr> is reached. However, the arguments passed are random (and might be dangerous syscalls by accident). Also, the old code at this address is currently not restored.

As a result, you must not continue execution after hitting a breakpoint!

14.12.7 Restrictions

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

- no real floating point support yet
- lots, lots, lots...

14.13 ATARI Jaguar/68k

This section describes specifics of the C library for ATARI Jaguar provided by the target `m68k-jaguar`.

The relevant files are `startup.o`, `libvc.a`, `libm.a`, `libjag.a`.

The config files `jaguar_unix` and `jaguar_windows` are part of the library.

14.13.1 Startup

The startup code `startup.o` sets up stack and heap. The startup process calls constructors to set up some global variables and initialize stdin, stdout and stderr.

The ATARI Jaguar has no OS, so it is impossible to define how input, output and files can be handled. There are a few set of function you have to define if you want to use stdio.

Alternatively you can use the `libjag.a`. This library initializes a console window with stdout support and uses optionally a SkunkBoard to redirect stderr and file I/O.

14.13.2 Floating point

Note that you have to link with a math library if you want to use floating point operations that are not natively implemented. All math functions, special startup code and printf/scanf functions which support floating point are contained in the math library only. Consider the ATARI Jaguar does not own a FPU so this library is pretty slow.

14.13.3 Stack

The library contains a default stack of 32KB. If another size is needed, you can add a global variable named __stack to your code:

```
/* Set 64kB stack */
unsigned long __stack = 65536;
```

14.13.4 Heap

Currently the free RAM is used as global heapsize for malloc() etc.

It is necessary to place a symbol named _BSS_END at the end of the BSS segment. The heap allocates the free RAM between _BSS_END and the bottom of the stack.

If less size is needed feel free to manipulate the value of _BSS_END.

All allocated heap objects can be used as internal JAGUAR objects, because they are qhrase aligned.

14.13.5 stdio support

The ATARI Jaguar lacks stdio support. So the `libvc.a` has just empty stub functions for open, close, read and write, which you may overwrite if you need stdio. Alternatively you can use `libjag.a` which has simple stdio and file I/O functionality.

```
/**
 * param name: name mentioned in fopen
 * param mode: mode mentioned in fopen
 * returns: > 0 a valid file handle
 *          < 0 to indicate an error
 *          the values 0,1,2 are used by stdin, stdout and stderr
 *
 * No need to handle stdin, stdout and stderr here
 */
int jagopen(const char *name,const char *mode)

/**
 * param handle: handle from jagopen
 *
 * No need to handle stdin, stdout and stderr here
 */
void jagclose(int handle)

/**
 * param handle: handle from jagopen
 * param p: points to the char buffer to fill.
 * param l: buffer size of p
 * returns: >=0 number of read bytes
 *          <0 indicate an error
 *
 * Handle stdin, stdout and stderr here
 */
size_t jagread(int handle,char *p,size_t l)

/**
 * param handle: handle from jagopen
 * param p: points to the char buffer to write.
 * param l: number of bytes of p
 * returns: >=0 number of bytes written
 *          <0 indicate an error
 *
 * Handle stdin, stdout and stderr here
 */
size_t jagwrite(int handle,const char *p, size_t l)

/**
 * param handle: handle from jagopen
```

```
 * param offset: number of bytes to seek.
 * param direction: see fseek direction
 * returns: =0 successful seek
            <>0 indicate an error
            -1: seek not supported
 *
 * Handle stdin, stdout and stderr here
 */
long jagseek(int handle,long offset,int direction)
```

14.13.6 The jaglib

The jaglib `libjag.a` provides simple functions to support your first steps in ATARI Jaguar programming. It initializes a simple console output window and comes with an old ATARI character set. If a SkunkBoard is available I/O functionality can be redirected.

Your first Jaguar program can look like this:

```
#include <stdio.h>

int main()
{
    printf("Hello, world\n");
}
```

Keep in mind: Your JAGUAR will get a red background color to indicate `main()` has exited.

The jaglib API documentation is available in a separate document. There is more demo code available in the jaglib-demo (`https://github.com/toarnold/jaglib-demo`) gibhub repository.

14.14 6502/C64

This is a port of vclib to the C64.

14.14.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

14.14.1.1 Startup

The default linker file creates program files that are loaded to address 0x801. A BASIC line is included so that the program can be started using `RUN` from BASIC. The startup code will turn off the BASIC ROM to allow usage of RAM until 0xD000 and most of the zero page without need for any special handling. The BSS segment will be cleared during startup.

With the default configuration, after exiting the C program, an infinite loop will be entered. When using the `+c64r` config, the program will return to BASIC an can be started again. However, this needs additional memory as the init values for the data section have to be stored in RAM. Also, some register values and zero page contents have to be saved. The overhead depends on the amount of initialized variables.

14.14.1.2 Command line

Command line parameters are supported by using the convention/code submitted by Stefan Haubenthal.

Command-lines look like these lines:

```
run
run : rem
run:rem arg1 " arg 2 is quoted "  arg3 "" arg5
```

14.14.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x02..0x8d` is used, but this can be changed in the linker file.

14.14.1.4 Stack

By default, the user stack is mapped from `0xC800..0xD000`. The size can be changed at the top of `vlink.cmd`.

14.14.1.5 Heap

Code and data/BSS are mapped starting after the BASIC init line. The heap is placed in the remaining space to stack start.

14.14.1.6 Banking

The following banking models are supported:

`-reuflat` This library supports a REU using a flat 16MB address space. The memory has to be addressed through far-pointers. It is not possible to declare variables in the REU nor to place code in the REU. The memory is addressed as 0x000000 to 0xFFFFFF. All accesses through far-pointers are addressing the REU. It is not possible to address the C64 memory through a far-pointer.

Far-pointer arithmetic only works on the lower 16bits. It is not possible to cross a bank boundary using far-pointers. Huge-pointers will support this, but are not yet fully implemented. In the meantime it is possible to use long integers and cast them to far-pointers.

Use `-lreuflat` to link with this library. Note that the library does not check for the presence of a REU.

`-reubank` This configuration reserves a 16KB memory space within the C64 memory as window for banking. As the bank number is stored as a single byte (with bank 255 denoting the unbanked memory), it is only possible to address up to about 4MB of memory in the REU.

Variables and code can be mapped into the REU using the `__bank()` attribute or `#pragma bank`. When calling a banked function, the corresponding bank will be copied from the REU into the C64 memory window.

Use the `+c64reu` configuration to use this mechanism. Currently the linker file provides 8 banks resulting in a 128K REU image. More banks can be added for larger expansions.

The configuration will create a usually C64 prg file containing the unbanked code and data as well as a REU image with extension `.b0`. It must be loaded (e.g. with an emulator or the TurboChameleon) before the prg file can is executed.

14.14.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.14.3 `stdio`

stdio supports `stdout`, `stderr` (both using the screen) and `stdin` (keyboard). Both are unbuffered by default.

Furthermore, file IO with standard C functions is supported for 1541 and compatible disk drives. Other devices have not been tested. Only sequential reading and writing of files is supported. No seeking etc. There are hardcoded limits for the maximum number of open files and the maximum length of filenames.

The `remove()` and `rename()` functions are supported using 1541

By default, device ID 8 is used. Another device ID can be specified as prefix to the filename:

```
/* try to open file "test" on the second drive */
FILE *f;
f=fopen("9:test","r");
...
```

`printf/scanf` functions which support floating point are contained in the math library only.

14.14.4 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.14.5 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`
- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

14.15 6502/NES

This is a port of vclib to the NES console.

14.15.1 Startup and Memory

Startup and memory layout is dependent on the ROM used. Currently two example configurations are provided. They can be selected with `+nrom256v` and `+unrom512v`. Have a look at the corresponding linker scripts in `vbcc/targets/6502-nes` for further details.

The necessary library routines to support configurations with several ROM banks are included.

14.15.1.1 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. The entire zero page can be used, but this can be changed in the linker file.

14.15.1.2 Stack

By default, the user stack starts from `0x0800` growing downwards.

14.15.1.3 Heap

By default, code and data/BSS are mapped starting after the system stack at `0x0200`. The heap is placed in the remaining space to stack start.

14.15.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.15.3 `stdio`

At the moment, stdio only supports `stdout`, `stderr` (both using the screen) and `stdin` (simple input via joypad).

`printf/scanf` functions which support floating point are contained in the math library only.

For input, up/down changes the current character, left/right moves the cursor, the B button deletes from the cursor position, and the A button confirms the input. You do not want to use this in real code.

The library contains a default character set. To replace it, link an object that contains a character set mapped to section `chars` and defines the global symbol `___stdchr`.

To replace stdio, the function `__read()` and `__write()` have to be implemented.

14.15.4 Interrupts

The library contains a default NMI implementation that is used for stdio handling and the `clock()`-function. It can be replaced by linking with an own implementation that starts at the global symbol `___nmi`. In this case the stdio and timing functions from vclib can not be used.

`___irq` can be used to overwrite the other IRQ vector. The default implementation in the library immediately returns.

14.15.5 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.15.6 Floating Point / IEEE

IEEE floating point is currently not available for this target.

14.16 6502/Atari

This is a port of vclib to Atari 8bit computers.

14.16.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

14.16.1.1 Startup

The default linker file creates program files that are loaded to address 0x600. The memory area can be adapted by changing `MEMSTART` and `MEMEND` in `vlink.cmd`.

With the default configuration, after exiting the C program, it will wait for pressing the return key before returning to DOS.

14.16.1.2 Command line

Command line parameters are not yet supported.

14.16.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x82..0xFF` is used, but this can be changed in the linker file.

14.16.1.4 Stack

By default, the startup code maps the user stack from `MEMTOP-STACKLEN..MEMTOP`. The size can be changed at the top of `vlink.cmd`.

14.16.1.5 Heap

Code and data/BSS are mapped starting at `MEMSTART`. The heap is placed in the remaining space to stack start.

14.16.1.6 Banking

Banking support for this target has not yet been implemented.

14.16.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.16.3 `stdio`

At the moment, stdio only supports `stdout`, `stderr` (both using the screen) and `stdin` (keyboard). Both are line-buffered by default.

`printf/scanf` functions which support floating point are contained in the math library only.

14.16.4 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.16.5 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`
- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

14.17 6502/BBC Micro/Master

This is a port of vclib to BBC 8bit computers.

14.17.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

14.17.1.1 Startup

The default linker file creates program files that are loaded to address 0x1900 up to 0x7B00 with 256 bytes of software stack. The memory area can be adapted by changing `OSHWM`, `HIMEM` and `STACKSTART` in `vlink.cmd`.

With the default configuration (`+bbc`), after exiting the C program, the code will enter an endless loop. If the reentrant configs are used (`+bbcr` or `+bbcbr`), the program will return to the command prompt. As this requires saving the zero page, a bit more memory is used.

14.17.1.2 Command line

If `main()` uses arguments, the command line parameters will be passed accordingly. There are hardcoded limits to the number of arguments (currently 8) and the maximum total command length (currently 80).

Space is used to separate arguments. The quote character (") can be used to group arguments containing spaces.

14.17.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x00..0x90` is used, but this can be changed in the linker file.

14.17.1.4 Stack

By default, the startup code maps the user stack from `STACKSTART..HIMEM`. The size can be changed at the top of `vlink.cmd`.

14.17.1.5 Heap

Code and data/BSS are mapped starting at `OSHWM`. The heap is placed in the remaining space to stack start.

14.17.1.6 Banking

When using the `+bbcb` or `bbcbr` configurations, vbcc supports banked memory, including automated bank-switching. Up to 16 sections of 16K size are supported. Each section starts at 0x8000.

The corresponding linker files `vlinkb.cmd` and `vlinkbr.cmd` can be edited to choose the banks that are required. Unused banks can be removed by commenting out (using old-style C-comments) the corresponding entries in the `SECTIONS` part of the linker file. When using bank 1-3 the section in the linker file could look like this:

```
...
```

```
SECTIONS
{
  text : {*(text)} >ram
  .dtors : { *(.dtors) } > ram
  .ctors : { *(.ctors) } > ram
  rodata : {*(rodata)} >ram
  data: {*(data)} >ram
  init : {*(init)} >ram
  zpage (NOLOAD) : {*(zpage) *(zp1) *(zp2)} >zero
  bss (NOLOAD): {*(bss)} >ram

/*
  b0 : {.=PAGEADDR; *(text0) *(rodata0) *(data0) *(bss0)
        RESERVE(PAGEADDR+PAGESIZE-.);
       } >b0 AT>dummy0
*/

  b1 : {.=PAGEADDR; *(text1) *(rodata1) *(data1) *(bss1)
        RESERVE(PAGEADDR+PAGESIZE-.);
       } >b1 AT>dummy1

  b2 : {.=PAGEADDR; *(text2) *(rodata2) *(data2) *(bss2)
        RESERVE(PAGEADDR+PAGESIZE-.);
       } >b2 AT>dummy2

  b3 : {.=PAGEADDR; *(text3) *(rodata3) *(data3) *(bss3)
        RESERVE(PAGEADDR+PAGESIZE-.);
       } >b3 AT>dummy3
/*
  b4 : {.=PAGEADDR; *(text4) *(rodata4) *(data4) *(bss4)
        RESERVE(PAGEADDR+PAGESIZE-.);
       } >b4 AT>dummy4
*/

  ...
```

During the linking process, apart from the normal output file, a 16K large image for each bank and a loader script will be generated. E.g. when using banks 1-3 and using the output file name test, the following files will be generated:

`test` The unbanked code/data.

`test.inf` Info file with start address.

`testb1` Image for bank1.

`testb2` Image for bank2.

`testb2` Image for bank2.

`loadtest` Loader

The contents of the loader `loadtest` will look like this:

```
*srload testb1 8000 1
*srload testb2 8000 2
*srload testb3 8000 3
*run test
```

The program can be started with `*exec loadtest`.

14.17.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.17.3 `stdio`

`stdout`, `stderr` (both using the screen) and `stdin` (keyboard) are supported. Furthermore normal file operations are possible using the usual C functions. There are hardcoded limits on the maximum number of simultaneously open files as well as the length of filenames.

Sequential reading and writing is supported, but no seeking. Furthermore, the `remove()` call is supported.

When using stdio to emit VDU control sequences, the function `__vdu_sequence()` is available to ensure verbatim 1:1 transmission of all characters:

```
/* print diagonal line */
__vdu_sequence(1);
for(int i=0;i<20;i++)
  printf("\x1f%c%c0",i,i);
__vdu_sequence(0);
```

`printf/scanf` functions which support floating point are contained in the math library only.

14.17.4 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.17.5 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`
- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

14.18 6502/MEGA65

This is a port of vclib to the MEGA65. This port is intended for the C65 mode with a C65 or compatible ROM (although the ROM is not used after the program is started). The C64 configuration can be used to create programs for the C64 mode.

14.18.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

The following basic configurations are available. See below for more details:

`+m65s` Standard unbanked configuration.

`+m65sr` Standard unbanked reentrant configuration.

`+m65sb` Standard banked configuration.

`+m65l` Large unbanked configuration.

`+m65lr` Large unbanked reentrant configuration.

`+m65lb` Large banked configuration.

14.18.1.1 Startup

The default linker file creates program files that are loaded to address 0x2001. A BASIC line is included so that the program can be started using `RUN` from BASIC. The startup code will switch to VIC-IV mode, remove write protection of ROM banks, turn on full speed and change to a suitable mapping. The BSS segment will be cleared during startup.

There are two sets of configurations that affect the configuration of upper memory. The standard versions (`+m65s, +m65sr, +m65sb`) will keep the IO area mapped in at $D000. This will limit the contiguous memory block for unbanked configurations to 0xCFFF. For banked configurations (see below) it will make a 16K window from 0x8000..0xBFFF available for banking. The large configurations (`+m65l, +m65lr, +m65lb`) will move the upper bound for unbanked programs to 0xFFFF. With banking, 32K window will be available from 0x8000..0xFFFF. In both cases the total amount of memory available for banking is the same in both configurations.

While the large configurations provide larger contiguous memory areas, accesses to the IO area have to be made through extended 28bit instructions which are much larger and slower. For programs doing many IO accesses, the standard configurations are recommended.

With the default configurations, after exiting the C program, an infinite loop will be entered. When using the reentrant (`+m65sr, +m65lr`) configs, the program will return to BASIC an can be started again. However, this needs additional memory as the init values for the data section have to be stored in RAM. Also, some register values and zero page contents have to be saved. The overhead depends on the amount of initialized variables.

Caution: The current configuration assumes that the Z register always contains 0. To work correctly, the Z register has to be 0 when C code is executed. The startup code will set it correctly and the compiler generated code will not touch it. However, when calling other code you may have to take care to save/restore the Z register or to set the Z register to 0 again.

14.18.1.2 Command line

Command line parameters are supported by using the convention/code submitted by Stefan Haubenthal.

Command-lines look like these lines:

```
run
run : rem
run:rem arg1 " arg 2 is quoted "  arg3 "" arg5
```

14.18.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x02..0xFF` is used, but this can be changed in the linker file.

14.18.1.4 Stack

By default, the user stack is mapped from `0xB800..0xC000`. For the banked version, it is mapped from `0x7800..0x8000`. The size can be changed at the top of `vlink.cmd` and `vlinkbank.cmd`.

14.18.1.5 Heap

Code and data/BSS are mapped starting after the BASIC init line. The heap is placed in the remaining space depending on the configuration.

14.18.1.6 Banking

The following banking models are supported:

`+m65sb` 16K window at 0x8000 with IO area mapped in at all times.

`+m65lb` 32K window at 0x8000.

Automated bank switching is supported in both modes. The mapping of banks to real memory in the standard configuration is like this:

```
Unbanked:       0x000000..0x007FFF
Bank0:          0x008000..0x00BFFF
Bank1:          0x00C000..0x00FFFF
Bank2:          0x010000..0x013FFF
Bank3:          0x014000..0x017FFF
          ...
```

On the large configuration, it looks like this:

```
Unbanked:       0x000000..0x007FFF
Bank0:          0x008000..0x00FFFF
Bank1:          0x010000..0x017FFF
Bank2:          0x018000..0x01FFFF
Bank3:          0x020000..0x027FFF
          ...
```

In both cases, the program start is moved to 0x1000. When using the banked configurations, the code can not be simply loaded from BASIC. The linker will create on large image without any BASIC lines. The file can be executed from SD-card by using a special loader that can be loaded from BASIC off a disk or disk image. When specifying a name as command line argument (see above), the loader will try to load this image from SD-card. If no argument is given, the loader will look for a file of the same name. Therefore by renaming the loader it can be made to automatically run a specific file.

If the loader is on the current disk/image and `myimage` on the SD:

```
LOAD "LOADER"
RUN:REM MYIMAGE
```

After renaming `LOADER` to `MYIMAGE`, it can be done like this:

```
RUN "MYIMAGE"
```

The colour RAM will be relocated to 0xFF80800 before loading to avoid being overwritten through the window at 0x1F800.

14.18.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.18.3 stdio

At the moment, stdio only supports `stdout`, `stderr` (both using the screen) and `stdin` (keyboard). Both are unbuffered by default. Using those streams will directly access the screen buffer and keyboard hardware. No ROM functions are needed once the program runs.

Furthermore it is possible to read files on the SD-card using standard C functions after opening them using `fopen()`. Hyppo services are used to read those files. There are several limitations due to the restrictions of Hyppo:

- Files can only be read sequentially, no seeking etc.
- Files can not be written to.
- Only one file can be open at the same time.

`printf/scanf` functions which support floating point are contained in the math library only.

14.18.4 Multiplication/Division

When generating code for the MEGA65, `vbcc` will make use of hardware multiplier/divider. This can greatly improve performance of such operations. Please note the following issues:

- Some versions of the MEGA65 core contain a bug in the hardware divider which will calculate wrong results in certain cases. As workaround you can specify option `-divbug` to use (much slower) 6502 software routines instead. Multiplication is not affected by the bug and will still be using the hardware multiplier.
- The hardware multiplier registers are mapped in the IO area. When using the large configurations (`+m65l, +m65lr, +m65lb`), they can only be accessed using extended 28bit instructions. The code generator and library functions will handle this, but there is some overhead (still nowhere near using software multiplication). If your code is speed critical and uses many multiplications we strongly recomment to use the standard configurations (`+m65s, +m65sr, +m65sb`). Those will set the option `-m65io` that tells `vbcc` to use faster direct IO accesses.

14.18.5 Interrupts

The provided configurations will disable interrupts on the MEGA65. All the library functions are written to work with disabled interrupts and do not use any ROM routines. The interrupt handlers in existing C65 ROMs do not work well with assembly language code and deficiencies in the mapping hardware make it very hard to use the ROM in a non-BASIC environment.

If an application wants to use interrupts, interrupt vectors have to be installed at 0xFFFA..0xFFFF. Take care that there are always valid vectors visible at this address (especially in a banked configuration). Also take care that those always point to a valid handler that is visible (i.e. do not use an ISR in banked memory).

14.18.6 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.18.7 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`
- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

14.19 6502/X16

This is a port of vclib to the Commander X16.

14.19.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

14.19.1.1 Startup

The default linker file creates program files that are loaded to address 0x801. A BASIC line is included so that the program can be started using `RUN` from BASIC. The startup code will turn off the BASIC ROM to allow usage of RAM until 0x9F00 and most of the zero page without need for any special handling. The BSS segment will be cleared during startup.

With the default configuration, after exiting the C program, an infinite loop will be entered. When using the `+x16r` config, the program will return to BASIC an can be started again. However, this needs additional memory as the init values for the data section have to be stored in RAM. Also, some register values and zero page contents have to be saved. The overhead depends on the amount of initialized variables.

14.19.1.2 Command line

Command line parameters are supported by using the convention/code submitted by Stefan Haubenthal.

Command-lines look like these lines:

```
run
run : rem
run:rem arg1 " arg 2 is quoted "  arg3 "" arg5
```

14.19.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x02..0x7e` is used, but this can be changed in the linker file.

14.19.1.4 Stack

By default, the user stack is mapped from `0x9700..0x9F00`. The size can be changed at the top of `vlink.cmd`.

14.19.1.5 Heap

Code and data/BSS are mapped starting after the BASIC init line. The heap is placed in the remaining space to stack start.

14.19.1.6 Banking

Banking support for this target is not yet implemented.

14.19.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.19.3 `stdio`

At the moment, stdio only supports `stdout`, `stderr` (both using the screen) and `stdin` (keyboard). Both are unbuffered by default.

`printf/scanf` functions which support floating point are contained in the math library only.

14.19.4 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.19.5 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`
- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

14.20 6502/PET

This is a port of vclib to the CBM PET series of computers.

14.20.1 Startup and Memory

Startup and memory layout is described in the following paragraphs.

14.20.1.1 Startup

The default linker file creates program files that are loaded to address 0x401. A BASIC line is included so that the program can be started using `RUN` from BASIC. RAM is available until 0x7FFF and most of the zero page without need for any special handling. The BSS segment will be cleared during startup.

With the default configuration, after exiting the C program, an infinite loop will be entered. When using the `+petr` config, the program will return to BASIC an can be started again. However, this needs additional memory as the init values for the data section have to be stored in RAM. Also, some register values and zero page contents have to be saved. The overhead depends on the amount of initialized variables.

14.20.1.2 Command line

Command line parameters are supported by using the convention/code submitted by Stefan Haubenthal.

Command-lines look like these lines:

```
run
run : rem
run:rem arg1 " arg 2 is quoted "  arg3 "" arg5
```

14.20.1.3 Zero Page

`vbcc` uses a number of zero page locations for register variables, stack pointer etc. in section `zpage`. Also, variables can be mapped to zero page using the `__zpage` attribute. By default the area `0x02..0x8d` is used, but this can be changed in the linker file.

14.20.1.4 Stack

By default, the user stack is mapped from `0x7F00..0x7FFF`. The size can be changed at the top of `vlink.cmd`.

14.20.1.5 Heap

Code and data/BSS are mapped starting after the BASIC init line. The heap is placed in the remaining space to stack start.

14.20.1.6 Banking

Automated banking is currently not supported.

14.20.2 Runtime

Apart from standard C library functions, `libvc.a` also provides a few runtime support functions needed by the compiler. Apart from the math and floating point functions mentioned in the documentation of the 6502 backend, it includes functions for saving/restoring registers.

14.20.3 stdio

At the moment, stdio only supports `stdout`, `stderr` (both using the screen) and `stdin` (keyboard). Both are unbuffered by default.

`printf/scanf` functions which support floating point are contained in the math library only.

14.20.4 Floating Point / wozfp

When using floating point, the math library `libm.a` must be linked using the `-lm` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on Steve Wozniaks routines from the 70s, somewhat adapted to the ABI of `vbcc`. These functions are small and reasonably usable, but they do not fully satisfy the requirements of C99.

Only a part of the C library functions for floating point is implemented. The list currently includes:

- `exp()`
- `pow()`
- `log()`
- `log10()`

14.20.5 Floating Point / IEEE

When using IEEE floating point, `-ieee` must be specified and the math library `libmieee.a` must be linked using the `-lmieee` option. It contains the floating routines as well as versions of the `printf/scanf` family that support floating point.

The floating point routines are based on SANE, somewhat adapted to the ABI of `vbcc` using wrapper functions. These functions should be fully C and IEEE compliant and provide precise results for 32 and 64bit floating point numbers (the library actually internally calculates all operation using 80bits, but vbcc currently only uses up to 64 bits).

Currently, this library probably must be run from RAM.

Most parts of the C library functions for floating point are implemented. The list currently includes:

- `exp()`, `expf()`, `expl()`
- `exp2()`, `exp2f()`, `exp2l()`
- `exp1m()`, `exp1mf()`, `exp1ml()`
- `pow()`, `powf()`, `powl()`
- `log()`, `logf()`, `logl()`
- `log1p()`, `log1pf()`, `log1pl()`
- `log2()`, `log2f()`, `log2l()`
- `log10()`, `log10f()`, `log10l()`
- `sqrt()`, `sqrtf()`, `sqrtl()`
- `sin()`, `sinf()`, `sinl()`
- `cos()`, `cosf()`, `cosl()`

- `tan()`, `tanf()`, `tanl()`
- `atan()`, `atanf()`, `atanl()`

15 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 error 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 uninitialized 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 in between.

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. "address 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)
311. "trailing garbage in #ifdef" (Preprocessor, Warning, ANSI-violation)
312. "too many arguments to macro" (Preprocessor, Warning, ANSI-violation)
313. "truncated comment" (Preprocessor, Warning|ANSI-violation)
314. "trailing garbage in preprocessing directive" (Preprocessor, Warning, ANSI-violation)
315. "variable-length array must have auto storage-class" (Error, ANSI-violation)
316. "member <%s> has type with zero alignment/size (probably void)" (Error, ANSI-violation, Fatal)
317. "stack information for target <%s> unavailable" (Warning)
318. "used registers information unavailable for target <%s>" (Warning)
319. "computed %sstack usage %d but set to %d" (Warning)
320. "unable to compute call targets" (Warning)
321. "computed register usage differs from specified one" (Warning)

322. "trailing garbage in #include" (Preprocessor, Warning ,ANSI-violation)
323. "target-warning: %s" (Warning)
324. "target-error: %s" (Error)

16 Backend Interface

16.1 Introduction

This chapter is under construction!

This chapter describes some of the internals of `vbcc` and tries to explain what has to be done to write a code generator for `vbcc`. However if someone wants to write one, I suggest to contact me first, so that it can be integrated into the source tree.

You have to create a new directory for the new target named `machines/<target-name>` and write the files `machine.c`, `machine.h` and `machine.dt`. The compiler for this target will be called `vbcc<target-name>` and can be built doing a `make TARGET=<target-name> bin/vbcc<target-name>`.

From now on integer means any of `char, short, int, long, long long` or their unsigned couterparts. Arithmetic means integer or `float` or `double` or `long double`. Elementary type means arithmetic or pointer.

Note that this documentation may mention explicit values when introducing symbolic constants. This is due to copying and pasting from the source code. These values may not be up to date and in some cases can be overridden. Therefore do never use the absolute values but rather the symbolic representations.

16.2 Building vbcc

This section deals with the steps necessary to build the typical `vbcc` executables from the sources.

16.2.1 Directory Structure

The vbcc-directory contains the following important files and directories:

`vbcc/` The main directory containing the compiler sources.

`vbcc/Makefile`
: The Makefile used to build `vbcc`.

`vbcc/frontend/`
: Directory containing the source to `vc`, the compiler driver.

`vbcc/machines/<target>/`
: Directory for the `<target>` backend.

`vbcc/machines/ucpp/`
: Directory containing the builtin preprocessor.

`vbcc/vsc/`
: Directory containing source to `vsc`, the instruction scheduler.

`vbcc/bin/`
: Directory the executables will be placed in.

All compiling is done from the main directory. The frontend `vc` is not target-dependend and therefore only one version is created.

Every available target has at least one subdirectory with its name in `vbcc/machines` and contains at least the files `machine.h`, `machine.c` and `machine.dt`. Target-specific object-files will also be stored in that directory.

The executables will be placed in `vbcc/bin/`. The main compiler will be called `vbcc<target>`.

16.2.2 Adapting the Makefile

Before building anything you have to insert correct values for `CC, NCC, LDFLAGS` and `NLDFLAGS` in the `Makefile`.

`CC` Here you have to insert a command that invokes an ANSI C compiler you want to use to build vbcc. It must support `-D`, `-I`, `-c` and `-o` the same like e.g. `vc` or `gcc`. Additional options should also be inserted here. E.g. if you are compiling for the Amiga with `vbcc` you should add `-DAMIGA`.

`LDFLAGS` Here you have to add options which are necessary for linking. E.g. some compilers need special libraries for floating-point.

`NCC`
`NLDFLAGS` These are similar to `CC` and `LDFLAGS` but they must always describe a native compiler, i.e. Programs compiled with `NCC/NLDFLAGS` must be executable on the host system. This is needed because during the build programs may have to be executed on the host.

An example for the Amiga using `vbcc` would be:

```
CC = vc -DAMIGA -c99
LDFLAGS = -lmieee
NCC = $(CC)
NLDFLAGS = $(LDFLAGS)
```

An example for a typical Unix-installation would be:

```
CC = cc
LDFLAGS = -lm
NCC = $(CC)
NLDFLAGS = $(LDFLAGS)
```

The following settings are probably necessary for Open/Free/Any BSD i386 systems:

```
CC = gcc -D_ANSI_SOURCE
LDFLAGS = -lm
NCC = $(CC)
NLDFLAGS = $(LDFLAGS)
```

16.2.3 Building vc

Note to users of Open/Free/Any BSD i386 systems: You will probably have to use GNU make instead of BSD make, i.e. in the following examples replace "make" with "gmake".

Type:

```
make bin/vc
```

16.2.4 Building vsc

Type:

```
make TARGET=<target> bin/vsc<target>
```

For example:

```
make TARGET=alpha bin/vscalpha
```

Omit this step if there is no file `machines/<target>/schedule.c`.

16.2.5 Building vbcc

Type:

```
make TARGET=<target> bin/vbcc<target>
```

For example:

```
make TARGET=alpha bin/vbccalpha
```

During the build the program `dtgen` will be generated and executed on the host-system. First it will ask you whether you are building a cross-compiler.

Answer `y` only if you are building a cross-compiler (i.e. a compiler which does not produce code for the same machine it it running on).

Note that it does _not_ matter if you are cross-building a compiler, i.e. if you are running on system A and building a B->B compiler by using an A->B compiler then you can answer `n`.

If you answered `y` you will be asked if your system/compiler offers certain datatypes. This refers to the compiler you described with CC in the Makefile. E.g. if `CC` is an A->B cross-compiler you have to answer the questions according to B. To each question answer `y` or `n` depending if such a datatype is available on that compiler. If you answered `y` you have to type in the name of that type on the compiler (e.g. `signed int`, `unsigned char` etc.). If there are not enough datatypes available to build `vbcc` an error message will be printed and the build aborts.

16.2.6 Configuring

Consult the `vbcc`-documentation for information on how to create the necessary config-files.

16.2.7 Building Cross-Compilers

As there is often confusion when it comes to cross-building compilers or building cross-compilers, here is what has to be done to cross-build a B->C cross-compiler on system A with only a native A->A compiler available.

This is done by first building an A->B compiler and then cross-building the B->C compiler using the A->B compiler.

For the first step you use the A->A compiler for `CC` as well as `NCC`. Now you type:

```
make bin/vc
make TARGET=B bin/vscB   # omit if there is no machines/B/schedule.c
make TARGET=B bin/vbccB
```

The questions about datatypes are answered according to A. Then you should write a `vc.config` for the `vbccB` cross-compiler.

Now create a second directory containing all the sources to `vbcc` and set `CC/LDFLAGS` to `vc` using the config-file for vbccB and `NCC/NLDFLAGS` to the A->A compiler. Type:

```
make bin/vc
make TARGET=C bin/vscC   # omit if there is no machines/C/schedule.c
make TARGET=C bin/vbccC
```

16.3 The Intermediate Code

`vbcc` will generate intermediate code for every function and pass this code to the code generator which has to convert it into the desired output.

In the future there may be a code generator generator which reads a machine description file and generates a code generator from that, but it is not clear whether this could simplify much without taking penalties in the generated code. Anyway, this would be a layer on top of the current interface to the code generator, so that the interface described in this document would still be valid and accessible.

16.3.1 General Format

The intermediate code is represented as a doubly linked list of quadruples (I am calling them ICs from now on) consisting mainly of an operator, two source operands and a target. They are represented like this:

```
struct IC{
    struct IC *prev;
    struct IC *next;
    int code;
    int typf;
    int typf2;
    [...]
    struct obj q1;
    struct obj q2;
    struct obj z;
    [...]
    struct ext_ic ext;  /* optional */
};
```

The only members relevant to the code generator are `prev, next, code, typf, typf2, q1, q2, z` and (optionally) `ext_ic`.

`prev` and `next` are pointers to the previous and next IC. The first IC has `prev==0` and the last one has `next==0`.

`typf` and `typf2` are the type of the operands of this IC. In most ICs all operands have the same type and therefore only `typf` is used. However, some ICs have operands of different types (e.g. converting an operand to another type or adding an integer to a pointer). `typf2` is used in these cases.

Macros are provided which yield the type of an operand. `q1typ(), q2typ()` and `ztyp()` return the type of the first source operand, the second source operand and the destination, respectively. They have to be passed a pointer to a valid IC as argument. The results are

undefined if the IC does not contain the specified operand (e.g. `q2typ()` for an IC with only a single operand).

The standard types which are defined by default are:

```
#define CHAR
#define SHORT
#define INT
#define LONG
#define LLONG
#define FLOAT
#define DOUBLE
#define LDOUBLE
#define VOID
#define POINTER
#define ARRAY
#define STRUCT
#define UNION
#define ENUM          /*  not relevant for code generator     */
#define FUNKT
```

and can be additionally or'ed by

```
#define UNSIGNED
#define CONST
#define VOLATILE
#define UNCOMPLETE
```

However, only `UNSIGNED` is of real importance for the code generator. `typf&NQ` yields the type without any qualifiers, `typf&NU` yields the type without any qualifiers but `UNSIGNED`.

It is possible for backends to define additional types. See Section 16.9.8 [exttypes], page 185, for documentation on how to extend the type system.

16.3.2 Operands

`q1`, `q2` and `z` are the source1 (quelle1 in German), source2 and target (ziel) operands, respectively. If a result has to be computed, it always will be stored in the object `z` and the objects `q1` and `q2` usually may not be destroyed during this operation (unless they are aliased with the destination).

The objects are described by this structure:

```
struct obj{
    int flags;
    int reg;
    int dtyp;
    struct Var *v;
    struct AddressingMode *am;
    union atyps{
        zchar vchar;
        zchar vuchar;
        zshort vshort;
```

```
            zushort vushort;
            zint vint;
            zuint vuint;
            zlong vlong;
            zulong vulong;
            zllong vllong;
            zullong vullong;
            zmax vmax;
            zumax vumax;
            zfloat vfloat;
            zdouble vdouble;
            zldouble vldouble;
        } val;
    };
```

`flags` specifies the kind of object. It can be a combination of

`#define KONST 1`
: The object is a constant. Its value is in the corresponding (to `typf` or `typf2`) member of `val`.

`#define VAR 2`
: The object is a variable. The pointer to its `struct Var` is in `v`. `val.vlong` vontains an offset that has to be added to it. Fur further details, see Section 16.3.3 [Variables], page 153.

`#define DREFOBJ 32`
: The content of the location in memory the object points to is used. `dtyp` contains the type of the pointer. In systems with only one pointer type, this will always be `POINTER`.

`#define REG 64`
: The object is a register. `reg` contains its number.

`#define VARADR 128`
: The address of the object is to be used. Only together with static variables (i.e. `storage_class == STATIC` or `EXTERN`).

The possible combinations of these flags should be:

- `0` (no object)
- `KONST`
- `KONST|DREFOBJ`
- `REG`
- `VAR`
- `VAR|REG`
- `REG|DREFOBJ`
- `KONST|DREFOBJ`
- `VAR|DREFOBJ`

- `VAR|REG|DREFOBJ`
- `VAR|VARADR`

Also some other bits which are not relevant to the code generator may be set.

Constants will usually be in `q2` if possible. One of the sources always is not constant and the target is always an lvalue. The types of the operands can be queried using the macros `q1typ()`, `q2typ()` and `ztyp()`. In most cases (i.e. when not explicitly stated) the type is an elementary type (i.e. arithmetic or pointer).

`am` can be used to store information on special addressing modes. This has to be handled by the by the code generator. However `am` has to be 0 or has to point to a `struct AddressingMode` that was allocated using `malloc()` when the code generator returns. `struct AddressingMode` has to be defined in `machine.h`.

`val` stores either the value of the object if it is a constant or an offset if it is a variable.

`code` specifies the operation. Fur further details see Section 16.3.5 [operations], page 156.

16.3.3 Variables

A `struct Var` looks like:

```
struct Var{
    int storage_class;
    [...]
    char *identifier;
    [...]
    zmax offset;
    struct Typ *vtyp;
    [...]
    char *vattr;
    unsigned long tattr;   /* optional */
};
```

The relevant entries are:

`identifier`
: The name of the variable. Usually only of interest for variables with external-linkage.

`storage_class`
: One of:

```
#define AUTO 1
#define REGISTER 2
#define STATIC 3
#define EXTERN  4
#define TYPEDEF 5       /*  not relevant    */
```

  The backend should use the macros `isauto()`, `isstatic()` and `isextern()` to check which category a variable falls into.

`offset`
: Contains an offset relative to the beginning of the variable's storage. Used, for example, when accessing members of structures.

`vtyp` The type of the variable (see Section 16.3.4 [compositetypes], page 154).

`vattr` A string with attributes used in the declaration of the variable. See Section 16.9.6 [targetattributes], page 184, for further details.

`tattr` Flags used when declaring the variable. See Section 16.9.6 [targetattributes], page 184, for further details.

If the variable is not assigned to a register (i.e. bit `REG` is not set in the flags of the corresponding `struct obj`) then the variable can be addressed in the following ways (with examples of 68k-code):

`isauto(storage_class) != 0`
: `offset` contains the offset inside the local-variables section. The code generator must decide how it's going to handle the activation record. If `offset < 0` then the variable is a function argument on the stack. In this case the offset in the parameter-area is `- (offset + maxalign)`.

  The code generator may have to calculate the actual offset to a stack- or frame-pointer from the value in `offset`.

  ```
  offset + val.vlong(sp)
  ```

  Note that `storage_class == REGISTER` is equivalent to `AUTO` - whether the variable is actually assigned a register is specified by the bit `REG` in the `flags` of the `struct obj`.

`isextern(storage_class) != 0`
: The variable can be addressed through its name in `identifier`.

  ```
  val.vlong + '_'identifier
  ```

`isstatic(storage_class) != 0`
: The variable can be addressed through a numbered label. The label number is stored in `offset`.

  ```
  val.vlong+'l'offset
  ```

16.3.4 Composite Types

The C language offers types which are composed out of other types, e.g. structures or arrays. Therefore, a C type can be an arbitrarily complex structure. Usually the backend does not need to deal with those structures. The ICs contain only the simple type flags, e.g. `INT` or `STRUCT`, but not the members of a structure (instead the size is given).

Most backends do not have to deal with complex types at all, but there are some ways to access them, if needed (for example, they may be needed when generating debug information). Therefore, this chapter describes the representation of such full types.

Types are represented by the following structure:

```
struct Typ {
  int flags;
  struct Typ *next;
  struct struct_declaration *exact;
  zmax size;
  char *attr;
```

```
};
```

`flags` is the simple type as it is generally used in the backend. The meaning of the other members depends on `flags`. `attr` is an attribute that can be added to the type using the sytax `__attr("...")` (which is parsed like a type-qualifier, e.g. `const`). If several attributes are specified for a type, the strings will be concatenated, separated by semi-colons.

If the type is a pointer (`ISPOINTER(flags) != 0`), then `next` will point to the type the pointer points to.

If the type is an array (`ISARRAY(flags) != 0`), then `size` contains the number of elements and `next` points to a type structure representing the type of each array element.

If the type is a structure (`ISSTRUCT(flags) != 0`), a union (`ISUNION(flags) != 0`) or a function (`ISFUNC(flags) != 0`), then `exact` is a pointer to a `struct_declaration` (which is also used to represent unions and function prototypes) that looks like this:

```
struct struct_declaration {
  int count;
  int label;
  int typ;
  ...
  struct struct_list (*sl)[];
  char *identifier;
};
```

`count` is the number of members, `label` can be used to store a label when generating debug information. `typ` is either `STRUCT`, `UNION` or `FUNKT` to denote whether it applies to a structure, union or function-prototype.

`identifier` is only available for struct- and union-tags.

`sl` points to an array of `struct struct_list`s which contain information on each member/parameter:

```
struct struct_list {
  char *identifier;
  struct Typ *styp;
  zmax align;
  int bfoffset;
  int bfsize;
  int storage_class;
  int reg;
};
```

`identifier` is the identifier of the member/parameter, if available. `styp` denotes the full type, `align` the alignment in bytes (only for struct/union), `bfoffset` and `bfsize` the size and offset of bitfield-members, `storage_class` the storage class of function parameters (may be `AUTO` or `REGISTER`) and `reg` denotes the register a parameter is passed in.

Example: If `struct Typ *t` points to a structure-type, then the type of the second structure member can be accessed through `(*t->exact->sl)[1].styp`.

A prototyped function will have a last argument of type `VOID` unless it is a function accepting a variable number of arguments. If a function was declared without a prototype it will have

no parameters, a function declared with prototype accepting no arguments will have one parameter of type `VOID`.

Also, in the case of a function type, the `next`-member of a `struct Typ` points to the return type of the function.

16.3.5 Operations

This section lists all the different operations allowed in the intermediate code passed to the backend. It lists the symbolic name of the `code` value (the value should not be used), a template of the operands and a description. The description sometimes contains internals (e.g. which types are stored in `typf` and which in `typf2`), but they should not be used. Access them using the macros provided (e.g. `q1typ,q2typ,ztyp`) whenever possible.

`#define ASSIGN 2`

Copy `q1` to `z`. `q1->z`.

`q2.val.vmax` contains the size of the objects (this is necessary if it is an array or a struct). It should be accessed using the `opsize()`-macro. `typf` does not have to be an elementary type!

The only case where `typf == ARRAY` should be in automatic initializations.

It is also possible that `(typf&NQ) == CHAR` but the size is != 1. This is created for an inline memcpy/strcpy where the type is not known.

`#define OR 16`
`#define XOR 17`
`#define AND 18`

Bitwise boolean operations. `q1,q2->z`.

All operands are integers.

`#define LSHIFT 25`
`#define RSHIFT 26`

Bit shifting. `q1,q2->z`.

`'q2'` is the number of shifts. All operands are integers.

`#define ADD 27`
`#define SUB 28`
`#define MULT 29`
`#define DIV 30`

Standard arithmetic operations. `q1,q2->z`.

All operands are of arithmetic types (integers or floating point).

`#define MOD 31`

Modulo (%). `q1,q2->z`.

All operands are integers.

`#define KOMPLEMENT 33`

Bitwise complement. `q1->z`.

All operands are integers.

`#define MINUS 38`

Unary minus. `q1->z`.

All operands are of arithmetic types (integers or floating point).

`#define ADDRESS 40`

Get the address of an object. `q1->z`.

`z` is always a pointer and `q1` is always an auto variable.

`#define CALL 42`

Call the function `q1`. `q1`.

`q2.val.vmax` contains the number of bytes pushed on the stack as function arguments for this call (use the `pushedargsize()`-macro to access this size). Those may have to be popped from the stack after the function returns depending on the calling mechanism.

A `CALL` IC has a member `arg_cnt` which contains the number of arguments to this function call. `arg_list[i]` (with `i` in the range 0...`arg_cnt-1`) contains the pointer to the IC passing the `i`-th argument.

`#define CONVERT 50`

Convert one type to another. `q1->z`.

`z` is always of the type `typf`, `q1` of type `typf2`.

Conversions between floating point and pointers do not occur, neither do conversions to and from structs, unions, arrays or void.

`#define ALLOCREG 65`

Allocate a register. `q1`.

From now on the register `q1.reg` is in use. No code has to be generated for this, but it is probably useful to keep track of the registers in use to know which registers are available for the code generator at a certain time and which registers are trashed by the function.

`#define FREEREG 66`

Release a register. `q1`.

From now on the register `q1.reg` is free.

Also it means that the value currently stored in `q1.reg` is not used any more and provides a little bit of data flow information. Note however, if a `FREEREG` follows a branch, the value of the register may be used at the target of the branch.

`#define COMPARE 77`

Compare and set condition codes. `q1,q2(->z)`.

Compare the operands and set the condition code, so that `BEQ, BNE, BLT, BGE, BLE` or `BGT` works as desired. If `z.flags == 0` (this is always the case unless the backend sets `multiple_ccs` to 1 and `-no-multiple-ccs` is not used) the condition codes will be evaluated only by an IC immediately following the `COMPARE`, i.e. the next instruction (except possible `FREEREG`s) will be a conditional branch.

However, if a target supports several condition code registers and sets the global variable `multiple_ccs` to 1, `vbcc` might use those registers and perform certain optimizations. In this case `z` may be non-empty and the condition codes have to be stored in `z`.

Note that even if `multiple_ccs` is set, a backend must nevertheless be able to deal with `z == 0`.

`#define TEST 68`

Test `q1` against 0 and set condition codes. `q1(->z)`

This is equivalent to `COMPARE q1,#0` but only the condition code for `BEQ` and `BNE` has to be set.

`#define LABEL 69`

Generate a label. `typf` specifies the number of the label.

```
#define BEQ 70
#define BNE 71
#define BLT 72
#define BGE 73
#define BLE 74
#define BGT 75
```

Branch on condition codes. (`q1`).

`typf` specifies the label where program execution shall continue, if the condition code is true (otherwise continue with next statement). The condition codes mean equal, not equal, less than, greater or equal, less or equal and greater than. If `q1` is empty (`q1.flags == 0`), the codes set by the last `COMPARE` or `TEST` must be evaluated. Otherwise `q1` contains the condition codes.

On some machines the type of operands of a comparison (e.g unsigned or signed) is encoded in the branch instructions rather than in the comparison instructions. In this case the code generator has to keep track of the type of the last comparison.

Similarly, in some architectures, the compare and the branch can be combined.

`#define BRA 76`

Branch always. `typf` specifies the label where program execution continues.

`#define PUSH 78`

Push `q1` on the stack (for argument passing). `q1`.

`q2.val.vmax` contains the size of the object (should be accessed using the `opsize()`-macro), `z.val.vmax` contains the size that has to be pushed (access it using the `pushsize()`-macro). These sizes may differ due to alignment issues.

`q1` does not have to be an elementary type (see `ASSIGN`). Also, `q1` can be empty. This is used for ABIs which require stack-slots to be omitted.

Depending on `ORDERED_PUSH` the `PUSH` ICs are generated starting with the first or the last arguments. The direction of the stack-growth can be chosen by the backend. Note that this is only used for function-arguments, they can be pushed in opposite direction of the real stack.

`#define ADDI2P 81`

Add an integer to a pointer. `q1,q2->z`.

`q1` and `z` are always pointers (of type `typf2`) and `q2` is an integer of type `typf`. `z` has to be `q1` increased by `q2` bytes.

`#define SUBIFP 82`

Subtract an Integer from a pointer. `q1,q2->z`.

`q1` and `z` are always pointers (of type `typf2`) and `q2` is an integer of type `typf`. `z` has to be `q1` decreased by `q2` bytes.

`#define SUBPFP 83`

Subtract a pointer from a pointer. `q1,q2->z`.

`q1` and `q2` are pointers (of type `typf2`) and `z` is an integer of type `typf`. `z` has to be `q1 - q2` in bytes.

`#define GETRETURN 93`

Get the return value of the last function call. `->z`.

If the return value is in a register, its number will be `q1.reg`. Otherwise `q1.reg` will be 0. `GETRETURN` immediately follows a `CALL` IC (except possible `FREEREG`s).

`#define SETRETURN 94`

Set the return value of the current function. `q1`.

If the return value is passed in a register, the register number will be `z.reg`. Otherwise `z.reg` will be 0. `SETRETURN` is immediately followed by a function exit (i.e. it is the last IC or followed by an unconditional branch to a label which is the last IC - always ignoring possible `FREEREG`s).

`#define MOVEFROMREG 95`

Move a register to memory. `q1->z`.

`q1` is always a register and `z` an array of size `regsize[q1.reg]`.

`#define MOVETOREG 96`

Load a register from memory. `q1->z`.

`z` is always a register and `q1` an array of size `regsize[z.reg]`.

`#define NOP 97`

Do nothing.

16.4 Type System

16.4.1 Target Data Types

As the compiler should be portable, we must not assume anything about the data types of the host system which is not guaranteed by ANSI/ISO C. Especially do not assume that the data types of the host system correspond to the ones of the target system.

Therefore, `vbcc` will provide typedefs which can hold a data type of the target machine and (as there is no operator overloading in C) functions or macros to perform arithmetic on these types.

The typedefs for the basic target's data types (they can be extended by additional types) are:

`zchar` Type `char` on the target machine.

`zuchar` Type `unsigned char` on the target machine.

`zshort` Type `short` on the target machine.

`zushort`	Type `unsigned short` on the target machine.
`zint`	Type `int` on the target machine.
`zuint`	Type `unsigned int` on the target machine.
`zlong`	Type `long` on the target machine.
`zulong`	Type `unsigned long` on the target machine.
`zllong`	Type `long long` on the target machine.
`zullong`	Type `unsigned long long` on the target machine.
`zmax`	A type capable of storing (and correctly doing arithmetic on) every signed integer type. Defaults to `zllong`.
`zumax`	A type capable of storing (and correctly doing arithmetic on) every unsigned integer type. Defaults to `zullong`.
`zfloat`	Type `float` on the target machine.
`zdouble`	Type `double` on the target machine.
`zldouble`	Type `long double` on the target machine.
`zpointer`	A byte pointer on the target machine. Not really used.

These typedefs and arithmetic functions to work on them will be generated by the program `dtgen` when compiling `vbcc`. It will create the files `machines/$(TARGET)/dt.h` and `dt.c`.

These files are generated from `machines/$(TARGET)/machine.dt` which must describe what representations the code generator needs. `dtgen` will then ask for available types on the host system and choose appropriate ones and/or install emulation functions, if available.

In `machine.dt`, every data type representation gets a symbol (the ones which are already available can be looked up in `datatypes/datatypes.h` - new ones will be added when necessary). The first 14 lines must contain the representations for the following types:

1. `signed char`
2. `unsigned char`
3. `signed short`
4. `unsigned short`
5. `signed int`
6. `unsigned int`
7. `signed long`
8. `unsigned long`
9. `signed long long`
10. `unsigned long long`
11. `float`
12. `double`
13. `long double`
14. `void *`

If the code generator can use several representations, these can be added on the same line separated by spaces. E.g. the code generator for m68k does not care if the integers are stored big-endian or little-endian on the host system because it only accesses them through the provided arithmetic functions. It does, however, access floats and doubles through byte-pointers and therefore requires them to be stored in big-endian-format.

16.4.2 Target Arithmetic

Now you have a lot of functions/macros performing operations using the target machine's arithmetic. You can look them up in `dt.h/dt.c`. E.g. `zmadd()` takes two `zmax` and returns their sum as `zmadd`. `zumadd()` does the same with `zumax`, `zldadd()` with long doubles. No functions for smaller types are needed because you can calculate with the wider types and convert the results down if needed.

Therefore, there are also conversion functions which convert between types of the target machine. E.g. `zm2zc` takes a `zmax` and returns the value converted to a `zchar`. Again, look at `dt.h/dt.c` to see which ones are there.

A few functions for converting between target and host types are also there, e.g. `l2zm` takes a long and returns it converted to `zmax`.

At last there are functions for comparing target data types. E.g. `zmleq(a,b)` returns true if `zlong a <= zlong b` and false otherwise. `zleqto(a,b)` returns true if `zlong a == zlong b` and false otherwise.

16.5 `machine.h`

This section describes the contents of the file `machine.h`. Note that some optional macros/declaration may be described someplace else in this manual.

`#include "dt.h"`
: This should be the first statement in `machine.h`.

`struct AddressingMode { ... };`
: If machine-specific addressing modes (see Section 16.8.2 [addressingmodes], page 173) are used, an appropriate structure can be specified here. Otherwise, just enter the following code:

```
struct AddressingMode {
    int never_used;
};
```

`#define MAXR <n>`
: Insert the number of available registers.

`#define MAXGF <n>`
: Insert the number of command line flags that can be used to configure the behaviour of the code generator. This must be at least one even if you do not use any flags.

`#define USEQ2ASZ <0/1>`
: If this is set to zero, `vbcc` will not generate ICs with the target operand being the same as the 2nd source operand. This can sometimes simplify the code-generator, but usually the code is better if the code-generator allows it.

`#define MINADDI2P <type>`
: Insert the smallest integer type that can be added to a pointer. Smaller types will be automatically converted to type `MINADDI2P` when they are to be added to a pointer. This may be subsumed by `shortcut()` in the future.

`#define MAXADDI2P <type>`
: Insert the largest integer type that can be added to a pointer. Larger types will be automatically converted to type `MAXADDI2P` when they are to be added to a pointer. This may be subsumed by `shortcut()` in the future.

`#define BIGENDIAN <0/1>`
: Insert 1 if integers are represented in big endian, i.e. the most significant byte is at the lowest memory address, the least significant byte at the highest.

`#define LITTLEENDIAN <0/1>`
: Insert 1 if integers are represented in little endian, i.e. the least significant byte is at the lowest memory address, the most significant byte at the highest.

`#define SWITCHSUBS <0/1>`
: Insert 1 if switch-statements should be compiled into a series of `SUB/TEST/BEQ` instructions rather than `COMPARE/BEQ`. This may be useful if the target has a more efficient `SUB`-instruction which sets condition codes (e.g. 68k).

`#define INLINEMEMCPY <n>`
: Insert the largest size in bytes allowed for inline memcpy. In optimizing compilation, certain library memcpy/strcpy-calls with length known at compile-time will be inlined using an `ASSIGN` IC if the size is less or equal to `INLINEMEMCPY`. The type used for the `ASSIGN` IC will be `UNSIGNED|CHAR`.

  This may be replaced by a variable of type `zmax` in the future.

`#define ORDERED_PUSH <0/1>`
: Insert 1 if `PUSH` ICs for function arguments shall be generated from left to right instead right to left.

`#define HAVE_REGPARMS 1`
: Insert this line if the backend supports register parameters (see Section 16.8.4 [regparm], page 177).

`#define HAVE_REGPAIRS 1`
: Insert this line if the backend supports register pairs (see Section 16.8.5 [regpairs], page 177).

`#define HAVE_INT_SIZET 1`
: Insert this line if `size_t` shall be of type `unsigned int` rather than `unsigned long`.

`#define EMIT_BUF_LEN <n>`
: Insert the maximum length of a line of code output.

`#define EMIT_BUF_DEPTH <n>`
: Insert the number of output lines that should be buffered. This can be useful for peephole-optimizing the assembly output (see below).

`#define HAVE_TARGET_PEEPHOLE <0/1>`
: Insert 1 if the backend provides an `asm_peephole()` function (see Section 16.8.14 [asmpeephole], page 180).

`#define HAVE_TARGET_ATTRIBUTES 1`
: Insert this line if the backend provides old target-specific variable-attributes (see Section 16.9.6 [targetattributes], page 184).

`#define HAVE_TARGET_PRAGMAS 1`
: Insert this line if the backend provides target-specific `#pragma`-directives (see Section 16.9.7 [targetpragmas], page 185).

`#define HAVE_REGS_MODIFIED 1`
: Insert this line if the backend supports inter-procedural register-allocation (see Section 16.8.11 [regsmodified], page 180).

`#define HAVE_TARGET_RALLOC 1`
: Insert this line if the backend supports context-sensitive register-allocation (see Section 16.8.10 [targetralloc], page 179).

`#define HAVE_TARGET_EFF_IC 1`
: Insert this line if the backend provides a `mark_eff_ics()` function (see Section 16.8.15 [markeffics], page 181).

`#define HAVE_EXT_IC 1`
: Insert this line if the backend provides a `struct ext_ic` (see Section 16.8.13 [extic], page 180).

`#define HAVE_EXT_TYPES 1`
: Insert this line if the backend supports additional types (see Section 16.9.8 [exttypes], page 185).

`#define HAVE_TGT_PRINTVAL 1`
: Insert this line if the backend provides an own `printval` function see Section 16.9.9 [tgtprintval], page 185).

`#define JUMP_TABLE_DENSITY <float>`
`#define JUMP_TABLE_LENGTH <int>`
: These values controls the creation of jump-tables (see Section 16.8.9 [jumptables], page 178).

`#define ALLOCVLA_REG <reg>`
`#define ALLOCVLA_INLINEASM <inline-asm>`
`#define FREEVLA_REG <reg>`
`#define FREEVLA_INLINEASM <inline-asm>`
`#define OLDSPVLA_INLINEASM <inline-asm>`
`#define FPVLA_REG <reg>`
: Necessary defines for C99 variable-length-arrays (see Section 16.9.14 [vlas], page 188).

`#define HAVE_LIBCALLS 1`
: Insert this line if the backend wants certain ICs to be replaced with calls to library functions (see Section 16.9.15 [libcalls], page 188).

`#define AVOID_FLOAT_TO_UNSIGNED 1`
`#define AVOID_UNSIGNED_TO_FLOAT 1`

Insert these lines to tell the frontend not to generate `CONVERT` ICs that convert between unsigned integers and floating point. In those cases, additional intermediate code will be generated that implements the conversion using only signed integers.

16.6 `machine.c`

This is the main part of the code generator. The first statement should be `#include "supp.h"` which will include all necessary declarations.

The following variables and functions must be provided by machine.c.

16.6.1 Name and Copyright

The codegenerater must define a zero-terminated character array `char cg_copyright[];` containing name and copyright-notice of the code-generator.

16.6.2 Command Line Options

You can use code generator specific commandline options. The number of flags is specified as `MAXGF` in `machine.h`. Insert the names for the flags as `char *g_flags_name[MAXGF]`. If an option was specified `(g_flags[i]&USEDFLAG)` is not zero. In `int g_flags[MAXGF]` you can choose how the options are to be used:

`0` The option can only be specified. E.g. if `g_flags_name[2]=="myflag"`, the commandline may contain `-myflag` and `(g_flags[2]&USEDFLAG)!=0`.

`VALFLAG` The option must be specified with an integer constant, e.g. `-myflag=1234`. This value can be found in `g_flags_val[2].l`.

`STRINGFLAG`
The option must be specified with a string, e.g. `-myflag=Hallo`. The pointer to the string can be found in `g_flags_val[2].p`.

16.6.3 Data Types

The following variables have to be initialized to describe the representation of the data types.

`MAX_TYPE` This macro contains the number of different types. In case of target-specific extended types (see Section 16.9.8 [exttypes], page 185) this is set by the backend, otherwise the frontend will use a default.

`zmax char_bit;`
The number of bits in a `char` on the target (usually 8).

`zmax align[MAX_TYPE+1];`
This array must contain the necessary alignments for every type in bytes. Some of the entries in this array are not actually used, but `align[type&NQ]` must yield the correct alignment for every type. `align[CHAR]` must be 1.

The alignment of a structure depends not only on `sizetab[STRUCT]` but also on the alignment of the members. The maximum of the alignments of all

members and `sizetab[STRUCT]` is the alignment of any particular structure, i.e. `sizetab[STRUCT]` is only a minimum alignment.

The same applies to unions and arrays.

`zmax maxalign;`
: This variable must be set to an alignment in bytes that is used when pushing arguments on the stack. (FIXME: describe stackalign)

`zmax sizetab[MAX_TYPE+1];`
: This array must contain the sizes of every type in bytes.

`zmax t_min[MAX_TYPE+1];`
: This array must contain the smallest representable number for every signed integer type.

`zumax t_max[MAX_TYPE+1];`
: This array must contain the largest representable number for every signed integer type.

`zumax tu_max[MAX_TYPE+1];`
: This array must contain the largest representable number for every unsigned integer type.

As `zmax` and `zumax` may be no elementary types on the host machine, those arrays have to be initialized dynamically (in `init_cg()`). It is recommended to use explicit typenames, e.g. `sizetab[INT]=l2zm(4L);` to keep it portable and allow later extensions of the type system.

Also note that those values may not be representable as constants by the host architecture and have to be calculated using the functions for arithmetic on the target's data types. E.g. the smallest representable value of a 32bit twos-complement data type is not guaranteed to be valid on every ANSI C implementation.

You may not use simple operators on the target data types but you have to use the functions or convert them to an elementary type of the host machine before (if you know that it is representable as such).

16.6.4 Register Set

The following variables have to be initialized to describe the register set of the target.

`MAXR`
: The valid registers are numbered from 1..`MAXR`. `MAXR` must be `#defined` in `machine.h`.

`char *regnames[MAXR+1]`
: This array must contain the names for every register. They do not necessarily have to be used in the assembly output but are used for explicit register arguments.

`zmax regsize[MAXR+1]`
: This array must contain the size of each register in bytes. It is used to create storage if registers have to be saved.

`int regscratch[MAXR+1]`
: This array must contain information whether a register is a scratchregister, i.e. may be destroyed during a function call (1 or 0). vbcc will generate code

to save/restore all scratch-registers which are assigned a value when calling a function (unless it knows the register will not be modified). However, if the code generator uses additional scratch-registers it has to take care to save/restore them.

Also, the code generator must take care that used non-scratch-registers are saved/restored on function entry/exit.

`int regsa[MAXR+1]`

This array must contain information whether a register is in use or not at the beginning of a function (1 or 0). The compiler will not use any of those registers for register variables or temporaries, therefore this can be used to mark special registers like a stack- or frame-pointer and to reserve registers to the code-generator. The latter may be reasonable if for many ICs code cannot be generated without using additional registers.

You must set `regsratch[i] = 0` if `regsa[i] == 1`. If you want it to be saved across function calls the code generator has to take care of this.

`int reg_prio[MAXR+1];`

This array must contain a priority (>=0) for every register. When the register allocator has to choose between several registers which seem to be equal, it will choose the one with the highest priority (if several registers have the same priority it is undefined which one will be taken).

Note that this priority is only the last decision factor if everything else seems to be equal. If one register seems to give a higher cost saving (according to the estimation of the register allocator) but has a lower priority, it will nevertheless be used. The priority can be used to fine-tune the register selection. Some guidelines:

- Scratch registers might have a higher priority than non-scratch registers (although the register-allocator will usually handle this anyway).
- Registers which are more restricted should have a higher priority (if they seem to give the same saving it is usually better to use the restricted registers and try to keep the more versatile ones for situation in which they can give better savings).
- Registers which are used for argument-passing should have lower priority than registers not used for arguments. The priority within the argument-registers should decrease as the frequency of usage as argument increases (typically the register for the first argument is used most frequently, etc.).

Note that for the array `zmax regsize[]` the same comments mentioned in the section on data types regarding initialization apply.

16.6.5 Functions

The following functions have to be implemented by the code generator. There may be optional additional functions described in other sections.

`int init_cg(void);`

This function is called after the commandline arguments are parsed. It can set up certain internal data, etc. The arrays regarding the data types and

the register set can be set up at this point rather than with a static initialization, however the arrays regarding the commandline options have to be static initialized. The results of the commandline options are available at this point.

If something goes wrong, 0 has to be returned, otherwise 1.

`void cleanup_cg(FILE *f);`
This function is called before the compiler exits. `f` is the output file which must be checked against 0 before using.

`int freturn(struct Typ *t);`
This function has to return the number of the register return values of type `t` are passed in. If the type is not passed in a register, 0 must be returned. Usually the decision can be made only considering `t->flags`, ignoring the full type (see Section 16.3.4 [compositetypes], page 154).

`int regok(int r, int t, int mode);`
Check whether the type t can be stored in register `r` and whether the usual operations (for this type) can be generated. Return 0, if not.

If `t` is a pointer and `mode==0` the register only has to be able to store the pointer and do arithmetic, but if `mode!=0` it has to be able to dereference the pointer.

`mode==-1` is used with context-sensitive register-allocation (see Section 16.8.10 [targetralloc], page 179). If the backend does not support it, this case can be handled equivalent to `mode==0`.

If `t==0` return whether the register can be used to store condition codes. This is only relevant if `multiple_ccs` is set to 1.

`int dangerous_IC(struct IC *p);`
Check if this IC can raise exceptions or is otherwise dangerous. Movement of ICs which are dangerous is restricted to preserve the semantics of the program.

Typical dangerous ICs are divisions or pointer dereferencing. On certain targets floating point or even signed integer arithmetic can raise exceptions, too.

`int must_convert(int from,int to,int const_expr);`
Check if code must be generated to convert from type `from` to type `to`. E.g. on many machines certain types have identical representations (integers of the same size or pointers and integers of the same size).

If `const_expr != 0` return if a conversion was necessary in a constant expression.

For example, a machine may have identical pointers and integers, but different sets of registers (one set supports integer operations and the other pointer operations). Therefore, `must_convert()` would return 1 (we need a `CONVERT` IC to move the value from one register set to the other).

This would imply that `vbcc` would not allow a cast from a pointer to an integer or vice-versa in constant expressions (as it will not generate code for static initializations). However, in this case, a static initialization would be ok as the representation is identical and registers are not involved. Therefore, the backend can return 1 if `const_expr == 0` and 0 otherwise.

`int shortcut(int code, int t);`

In C no operations are done with chars and shorts because of integral promotion. However sometimes `vbcc` might see that an operation could be performed with the short types yielding the same result.

Before generating such an instruction with short types `vbcc` will ask the code generator by calling `shortcut()` to find out whether it should do so. Return true iff it is a win to perform the operation `code` with type `t` rather than promoting the operands and using e.g. `int`.

`void gen_code(FILE *f, struct IC *p, struct Var *v, zmax offset);`

This function has to emit code for a function to stream `f`. `v` is the function being generated, `p` is a pointer to the list of ICs, that has to be converted. `offset` is the space needed for local variables in bytes.

This function has to take care that only scratchregisters are destroyed by this function. The array `regused` contains information about the registers that have been used by `vbcc` in this function. However if the code generator uses additional registers it has to take care of them, too.

The `regs[]` and `regused[]` arrays may be overwritten by `gen_code()` as well as parts of the list of ICs. However the list of ICs must still be a valid list of ICs after `gen_code()` returns.

All assembly output should be generated using the available `emit` functions. These functions are able to keep several lines of assembly output buffered and allow peephole optimizations on assembly output (see Section 16.8.14 [asmpeephole], page 180).

`void gen_ds(FILE *f, zmax size, struct Typ *t);`

Has to emit output that generates `size` bytes of type `t` initialized with proper 0.

`t` is a pointer to a `struct Typ` which contains the precise type of the variable. On machines where every type can be initialized to 0 by setting all bits to zero, the type does not matter. Otherwise see Section 16.3.4 [compositetypes], page 154.

All assembly output should be generated using the available `emit` functions.

`void gen_align(FILE *f, zmax align);`

Has to emit output that ensures the following data to be aligned to `align` bytes.

All assembly output should be generated using the available `emit` functions.

`void gen_var_head(FILE *f, struct Var *v);`

Has to print the head of a static or external variable `v`. This includes the label and necessary information for external linkage etc.

Typically variables will be generated by a call to `gen_align()` followed by `gen_var_head()` and (a series of) calls to `gen_dc()` and/or `gen_ds()`. It may be necessary to keep track of the information passed to `gen_var_head()`.

All assembly output should be generated using the available `emit` functions.

`void gen_dc(FILE *f, int t, struct const_list *p);`

Emit initialized data. `t` is the basic type that has to be emitted. `p` points to a `struct const_list`.

If `p->tree != 0` then `p->tree->o` is a `struct obj` which has to be emitted. This will usually be the address of a variable of storage class static or unsigned, possibly with an offset added (see Section 16.3.2 [operands], page 151, for further details).

if `p->tree == 0` then `p->val` is a `union atyps` which contains (in the member corresponding to `t`) the constant value to be emitted.

All assembly output should be generated using the available `emit` functions.

`void init_db(FILE *f);`
: If debug-information is requested, this functions is called after `init_cg()`, but before any code is generated. See also Section 16.9.10 [debuginfo], page 185.

`void cleanup_db(FILE *f);`
: If debug-information is requested, this functions is called prior to `cleanup_cg()`. See also Section 16.9.10 [debuginfo], page 185.

16.7 Available Support Functions, Macros and Variables

This section lists a series of general variables, macros and functions which are available to the backend and may prove useful. Note that there may be additional support specific to certain features which will be mentioned at appropriate sections in this manual.

`MAXINT`
: A constant for the largest target integer type (`zmax`). It is outside the range of the other types and cannot be accessed by an application (although there will usually be an accessible type with the same representation).

`MAX_TYPE`
: The type number of the last type.

`NQ`
: A mask. `t & NQ` will delete all type-qualifiers of a type.

`NU`
: A mask. `t & NU` will delete all type-qualifiers but `UNSIGNED` of a type.

`q1typ(p)`
: Yields the type of the first source operand of IC `p`. Undefined if the operand is not used!

`q2typ(p)`
: Yields the type of the second source operand of IC `p`. Undefined if the operand is not used!

`ztyp(p)`
: Yields the type of the destination operand of IC `p`. Undefined if the operand is not used!

`iclabel(p)`
: Returns the label of an IC. Only defined if `p->code` is `LABEL`, `BEQ`, `BNE`, `BLT`, `BGT`, `BLE` or `BGE`.

`opsize(p)`
: Returns the size of the operand of an `ASSIGN` or `PUSH` IC as `zmax`.

`pushsize(p)`
: Returns the stack-adjustment value of a `PUSH` IC as `zmax`. It is always greater or equal than `opsize(p)`.

`pushedargsize(p)`
: Returns the space occupied by arguments passed on the stack as parameters for a function call. Only valid for `CALL` ICs.

`isstatic(sc)`
: Tests whether the storage-class `sc` denotes a variable with static storage and no external linkage.

`isextern(sc)`
: Tests whether the storage-class `sc` denotes a variable with static storage and external linkage.

`isauto(sc)`
: Tests whether the storage-class `sc` denotes a variable with automatic storage-duration.

`t_min(t)`
`t_max(t)`
: These macros yield the smallest and largest representable value of any target integer type, e.g. `t_min(INT)` or `t_max(UNSIGNED|LONG)`.

`ISPOINTER(t)`
`ISINT(t)`

`ISFLOAT(t)`
`ISFUNC(t)`
`ISSTRUCT(t)`
`ISUNION(t)`
`ISARRAY(t)`
`ISSCALAR(t)`
`ISARITH(t)`
: These macros test whether the simple type `t` is a pointer type, an integral type, a floating point type, a function, a structure type, a union type, an array type, a scalar (integer, floating point or pointer) and an arithmetic type (integer or floating point), respectively.

`int label;`
: The number of the last label used so far. For a new label number, use `++label`.

`zmax falign(struct Typ *t);`
: This function returns the alignment of a full type. Contrary to the `align[]` array provided by the backend (which is used by this function), it will yield correct values for composite types like structures and arrays.

`zmax szof(struct Typ *t);`
: This function returns the size in bytes of a full type. Contrary to the `sizetab[]` array provided by the backend (which is used by this function), it will yield correct values for composite types like structures and arrays.

`void *mymalloc(size_t size);`
`void *myrealloc(void *p,size_t size);`
`void myfree(void *p);`
: Memory allocation functions similar to `malloc()`, `realloc()` and `free`. They will automatically clean up the exit in the case an allocation fails. Also, some debug possibilities are available.

```
void emit(FILE *f,const char *fmt,...);
void emit_char(FILE *f,int c) ;
void emitval(FILE *f,union atyps *p,int t);
void emitzm(FILE *f,zmax x);
void emitzum(FILE *f,zumax x);
```

All output produced by the backend should be produced using these functions. `emit()` uses a format like `printf()`, `emitval(), emitzm()` and `emitzum()` are suitable to output target integers as decimal text. Currently emitting floating point constants has to be done by the backend.

```
int is_const(struct Typ *);
```

Tests whether a full type is constant (e.g. to decide whether it can be put into a ROM section).

```
int is_volatile_obj(struct obj *);
int is_volatile_ic(struct IC *);
```

Tests whether an object or IC is volatile. Only of interest to the backend in rare cases.

```
int switch_IC(struct IC *p);
```

This function checks whether `p->q2` and `p->z` use the same register (including register pairs). If they do, it will try to swap `p->q1` and `p->q2` (only possible if the IC is commutative). It is often possible to generate better code if `p->q2` and `p->z` do not collide. Note however, that it is not always possible to eliminate a conflict and the code generator still has to be able to handle such a case.

The function returns 0 if no modification took place and non-zero if the IC has been modified.

```
union atyps gval;
void eval_const(union atyps *p,int t);
void insert_const(union atyps *p,int t);
```

For every target data type there is a corresponding global variable of that type, e.g. `zchar vchar, zuchar vuchar, zmax vmax` etc. These two functions simplify handling of target data types by transferring between a `union atyps` and these variables.

`eval_const()` reads the member of the union corresponding to the type `t` and converts it into all the global variables while `insert_const()` takes the global variable according to `t` and puts it into the appropriate member of the `union atyps`.

The global variable `gval` may be used as a temporary `union atyps` by the backend.

```
void printzm(FILE *f,zmax x);
void printzum(FILE *f,zumax x) ;
void printval(FILE *f,union atyps *p,int t);
void printtype(FILE *o,struct Typ *p);
void printobj(FILE *f,struct obj *p,int t);
void printic(FILE *f,struct IC *p);
void printiclist(FILE *f,struct IC *first);
```

This is a series of functions which print a more or less human readable version of the corresponding type to a stream. These functions are to be used only for debugging purposes, not for generating code. Also, the arguments must contain valid values.

`bvtype`
`BVSIZE(n)`

`vbcc` provides macros and functions for handling bit-vectors which may also be used by the backend. `bvtype` is the basic type to create bit-vectors of. `BVSIZE(n)` yields the number of bytes needed to implement a bit-vector with `n` elements.

```
bvtype *mybv = mymalloc(BVSIZE(n));
```

`BSET(bv,n)`
`BCLR(bv,n)`
`BTST(bv,n)`

Macros which set, clear and test the `n`-th bit in bit-vector `bv`.

```
void bvunite(bvtype *dest,bvtype *src,size_t len);
void bvintersect(bvtype *dest,bvtype *src,size_t len);
void bvdiff(bvtype *dest,bvtype *src,size_t len);
```

These functions calculate the union, intersection and difference of two bit-vectors. `dest` is the first operand as well as the destination. `len` is the length of the bit-vectors in bytes, not in bits.

```
void bvcopy(bvtype *dest,bvtype *src,size_t len);
void bvclear(bvtype *dest,size_t len);
void bvsetall(bvtype *dest,size_t len);
```

These functions copy, clear and fill bit-vectors.

```
int bvcmp(bvtype *bv1,bvtype *bv2,size_t len);
int bvdointersect(bvtype *bv1,bvtype *bv2,size_t len);
```

These functions test whether two bit-vectors are equal or have a non-empty intersection, respectively. The do not modify the bit-vectors.

16.8 Hints for common Optimizations

While it is no easy job to produce a stable code generator for a new target architecture, there is a huge difference between a simple backend and a highly optimized code generator which produces small and efficient high quality code. Although `vbcc` is able to do a lot machine independent global optimizations for every target automatically, it is still common for an optimized backend to produce code up to twice as fast on average as a simple one.

Sometimes, a simple backend is sufficient and the work required to produce high-quality code is not worthwhile. However, this section lists a series of common backend optimizations which are often done in case that good code-quality is desired. Note that neither are all of these optimizations applicable (without modifications or at all) to all architectures nor is this an exhaustive list. It is just a list of recommendations to consider. You have to make sure that the optimization is safe and beneficial for the architecture you are targetting.

16.8.1 Instruction Combining

While ICs are often a bit more powerful than instructions of a typical microprocessor, sometimes several of them can be implemented by a single instruction or more efficient code can be generated when looking at a few of them rather than at each one separately.

In the simple case, this can be done by looking at the current IC, deciding whether it is a candidate for combining and then test whether the next IC (or ICs) are suitable for combining. This is relatively easy to perform, however some care has to be taken to verify that the combination is indeed legal (e.g. what happens if the first IC modifies a value which is used by the following IC).

A more sophisticated implementation might look at a larger sequence of instructions to find more possibilities for optimization. Detecting whether the combination is legal becomes much more difficult then.

Sometimes the IC migh compute a temporary result which would be eliminated by the complex machine instruction. Then it is necessary to verify that it was indeed a temporary result which is not used anywhere else. As long as the result is in a register, this can be done by checking for a `FREEREG` IC.

Examples for instruction combining are multiply-and-add or bit-test instructions which are available on many architectures. Special cases are complex addressing modes and instructions which can automatically set condition codes which are described in the following sections.

16.8.2 Addressing Modes

The intermediate code generated by vbcc does not use any addressing-modes a target might offer. Therefore the code generator must find a way to combine several statements if it wants to make use of these modes. E.g. on the m68k the intermediate code

```
ADDI2P  int     a0,#20->a1
ASSIG   int     #10->(a1)
FREEREG         a1
```

could be translated to

```
move.l  #10,20(a0)
```

(notice the `FREEREG` which is important).

To aid in this there is a pointer to a `struct AddressingMode` in every `struct obj`. A code generator could e.g. do a pass over the intermediate code, find possible uses for addressing-modes, allocate a `struct AddressingMode` and store a pointer in the `struct obj`, effectively replacing the obj.

If the code generator supports extended addressing-modes, you have to think of a way to represent them and define the `struct AddressingMode` so that all modes can be stored in

it. The machine independent part of `vbcc` will not use these modes, so your code generator has to find a way to combine several statements to make use of these modes.

A possible implementation of a structure to handle the addressing mode described above as well as a register-indirect mode could be:

```
#define IMM_IND 1
#define REG_IND 2

struct AddressingMode {
  int flags;   /* either IMM_IND or REG_IND */
  int base;    /* base register */
  zmax offset; /* offset in case of IMM_IND */
  int idx;     /* index register in case of REG_IND */
}
```

When the code generator is done that pointer in every `struct obj` must either be zero or point to a `mymalloc`ed `struct AddressingMode` which will be free'd by `vbcc`.

Following is an example of a function which traverses a list of ICs and inserts addressing modes with constant offsets where possible.

```
/* search for possible addressing-modes */
static void find_addr_modes(struct IC *p)
{
  int c,c2,r;
  struct IC *p2;
  struct AddressingMode *am;

  for(;p;p=p->next){
    c=p->code;

    if(IMM_IND&&(c==ADDI2P||c==SUBIFP)&&
       isreg(z)&&(p->q2.flags&(KONST|DREFOBJ))==KONST){
      /* we have found addi2p q1,#const->reg */
      int base;zmax of;struct obj *o;

      eval_const(&p->q2.val,p->typf);
      /* handle sub instead of add */
      if(c==SUBIFP)
        of=zmsub(l2zm(0L),vmax);
      else
        of=vmax;

      /* Is the offset suitable for an addressing mode? */
      if(ISVALID_OFFSET(vmax)){
        r=p->z.reg;
        /* If q1 is a register, we use it as base-register,
           otherwise q1 is loaded in the temporary register
           and this one used as base register. */
        if(isreg(q1))
```

```
  base=p->q1.reg;
else
  base=r;

o=0;
/* Now search the following instructions. */
for(p2=p->next;p2;p2=p2->next){
  c2=p2->code;

  /* End of a basic block. We have to abort. */
  if(c2==CALL||c2==LABEL||(c2>=BEQ&&c2<=BRA)) break;

  /* The temporary register is used. We have to abort. */
  if(c2!=FREEREG&&(p2->q1.flags&(REG|DREFOBJ))==REG&&
     p2->q1.reg==r)
      break;
 if(c2!=FREEREG&&(p2->q2.flags&(REG|DREFOBJ))==REG&&
     p2->q2.reg==r)
      break;

  if(c2!=CALL&&(c2<LABEL||c2>BRA)&&c2!=ADDRESS){
    /* See, if we find a valid use (dereference) of the
       temporary register. */
    if(!p->q1.am&&(p2->q1.flags&(REG|DREFOBJ))==(REG|DREFOBJ)&&
      p2->q1.reg==r){
      if(o) break;
      o=&p2->q1;
    }
    if(!p->q1.am&&(p2->q2.flags&(REG|DREFOBJ))==(REG|DREFOBJ)&&
       p2->q2.reg==r){
      if(o) break;
      o=&p2->q2;
    }
    if(!p->q1.am&&(p2->z.flags&(REG|DREFOBJ))==(REG|DREFOBJ)&&
       p2->z.reg==r){
      if(o) break;
      o=&p2->z;
    }
  }
  if(c2==FREEREG||(p2->z.flags&(REG|DREFOBJ))==REG){
    int m;
    if(c2==FREEREG)
      m=p2->q1.reg;
    else
      m=p2->z.reg;
    if(m==r){
    /* The value of the temporary register is not used any more
```

```
                    (either due to FREEREG or because it is overwritten).
                    If we have found exactly one dereference, we can use
                    a target addressing mode. */
                  if(o){
                    o->am=am=mymalloc(sizeof(*am));
                    am->flags=IMM_IND;
                    am->base=base;
                    am->offset=zm2l(of);
                    if(isreg(q1)){
                      /* The base already was in a register. We can
                         eliminate the ADDI2P IC. */
                      p->code=c=NOP;p->q1.flags=p->q2.flags=p->z.flags=0;
                    }else{
                      /* The base was not in a register.
                         We have to load it . */
                      p->code=c=ASSIGN;p->q2.flags=0;
                      p->typf=p->typf2;p->q2.val.vmax=sizetab[p->typf2&NQ];
                    }
                  }
                  break;
                }
                if(c2!=FREEREG&&m==base) break;
                continue;
              }
            }
          }
        }
      }
    }
```

16.8.3 Implicit setting of Condition Codes

Many architectures have instruction that automatically set the condition codes according to the computed result. For these architectures it is generally a good idea to keep track of the setting of condition codes (e.g. if they reflect the state of some object or register). A subsequent `TEST` or `COMPARE` instruction can then often be eliminated.

Care has to be taken to delete this information if either the condition codes may be modified or the object they represent is modified. Also, this optimization is usually hard to do across labels.

Some architectures provide versions of instructions which set condition codes as well as versions which do not. This obviously enable more optimizations, but it is more difficult to make use of this. One possibility is to search the list of ICs backwards starting from every suitable `TEST` or `COMPARE` instruction. If an IC is found which computes the tested object, the IC can be marked (extended ICs can be used for marking, see Section 16.8.13 [extic], page 180).

16.8.4 Register Parameters

While passing of arguments to functions can be done by pushing them on the stack, it is often more efficient to pass them in registers if the architecture has enough registers.

To use register parameters you have to add the line

```
#define HAVE_REGPARMS 1
```

to `machine.h` and define a

```
struct reg_handle {...}
```

This struct is used by the compiler to find out which register should be used to pass an argument. `machine.c` has to contain an initialized variable

```
struct reg_handle empty_reg_handle;
```

which represents the default state, and a function

```
int reg_parm(struct reg_handle *, struct Typ *, int vararg, struct Typ *);
```

which returns the number of the register the next argument will be passed in (or 0 if the argument is not passed in a register). Also, it has to update the reg_handle in a way that successive calls to `reg_parm()` yield the correct register for every argument.

`vararg` is different from zero, if the argument is part of the variable arguments of a function accepting a variable number of arguments.

It is also possible to return a negative number `x`. In this case, the argument will be passed in register number `-x`, but also a stack-slot will be reserved for this argument (i.e. a `PUSH` IC without an operand will be generated). If `-double-push` is specified, the argument will also be written to the stack-slot (i.e. it will be passed twice, in a register and on the stack).

16.8.5 Register Pairs

Often, there are types which cannot be stored in a single machine register, but it may be more efficient to store them in two registers rather than in memory. Typical examples are integers which are bigger than the register size or architectures which combine two floating point registers into one register of double precision.

To make use of register pairs, the line

```
#define HAVE_REGPAIRS 1
```

has to be added to `machine.h`. The register pairs are declared as normal registers (each register pair counts as an own register and `MAXR` has to be adjusted). Usually only adjacent registers are declared as register pairs. Note that `regscratch` must be identical for both registers of a pair.

Now the function

```
int reg_pair(int r,struct rpair *p);
```

must be implemented. If register `r` is a register pair, the function has to set `p->r1` and `p->r2` to the first and second register which comprise the pair and return 1. Otherwise, zero has to be returned.

16.8.6 Elimination of Frame-Pointer

Local variables on the stack can usually be addressed via a so-called frame-pointer which is set set to current stack-pointer at the entry of a function. However, in the code generated by `vbcc`, the difference between the stack-pointer and the frame-pointer is fixed at any instruction.

Therefore it is possible to keep track of this offset (by counting the bytes every time code for pushing or popping from the stack is generated). Using this offset, local variables can perhaps be addressed using the stack-pointer directly. Benefit would be smaller function entry/exit code as well as an additional free register which can be used for other purposes.

Note that only few debuggers can handle such a situation.

16.8.7 Delayed popping of Stack-Slots

In most ABIs arguments which are pushed on the stack are not popped by the called function but the caller pops them by adjusting the stack after the callee returns (otherwise variable arguments would be hard to implement).

If several functions are called in sequence, it is not necessary to adjust the stack after each call but the arguments for several calls can be popped at once. It can be implemented by keeping track of the size to be popped and deferring popping to a point where it has to be done (e.g. a label or a branch). Also, in the case of nested calls, care has to be taken to pop arguments at the right time.

Note that this usually saves code-size and execution time but will increase stack-usage. Therefore, it may not be advisable for small systems.

16.8.8 Optimized Return

Return instructions are not explicitly represented in ICs. Rather, they are branches to a label which is the last IC in the list (except possible `FREEREG`s).

It is possible to generate working code by translating these branches normally, but directly inserting the function exit code instead of a branch is often faster. It is most recommendable if the exit code is small (e.g. no registers have to be restored and no stack-frame removed).

Another common possibility for optimization is a function call as the last IC. If return addresses are pushed on the stack and no function exit code is needed, it is usually possible to generate a jump-instruction, i.e. replace

```
call  somefunc
ret
```

by

```
jmp   somefunc
```

16.8.9 Jump Tables

An important optimization is the creation of jump-tables for a series of comparisons with constants. Such series are usually created by a C `switch` construct, but vbcc can also recognize some of them if they are created through `if`-sequences.

`supp.c` provides the function `calc_case_table(<IC>,<density>)` to check for constructs that can be replaced by a jump table. The arguments are the start IC to look for (it has to be a `COMPARE`-IC with a constant as `q2`) and a minimal density. The density reflects the

number of cases that are used divided by the range of cases. If the density is high, vbcc will use jump-tables only for sequences that have few unused cases inside. If the case tables occupy multiple ranges, vbcc is able to split them up and create multiple jump-tables.

`calc_case_table` returns a pointer to a `struct case_table` with the following content:

`num` The number of cases.

`typf` The type of the case IDs.

`next_ic` The first IC after the list of ICs that can be replaced by the jump-table.

`density` The case density.

`vals` The values of the case IDs (array containing `num` entries).

`labels` The labels of the code corresponding to the case IDs (array containing `num` entries).

`min` The lowest case ID.

`max` The highest case ID.

`diff` `max-min`.

If the backend decides to emit a jump-table, it has to generate code that will check that the control expression lies between `min` and `max`. If not, the jump-table must not be executed. Code for the computed jump must then be generated. The actual table can be emitted using `emit_jump_table()`. Processing can then continue with `next_ic`.

16.8.10 Context-sensitive Register-Allocation

The `regok()` function is only a simple means of telling the register allocator which registers to use. It works fairly well with orthogonal register and instruction sets. However, it does not really care about the operations performed and it allocates variables to registers only according to their type, not according to the operations performed.

Some architectures provide different kinds of registers which are able to store a type, but not all of them are able to perform all operations or some operations are more expensive with some registers. To do good register allocation for these systems, the operations which are used on variables have to be considered.

If the backend wants to support this kind of register allocation, it has to define `HAVE_TARGET_RALLOC` and provide the following functions or macros:

`int cost_move_reg(int x,int y);`
The cost of copying register `x` to register `y`.

`int cost_load_reg(int r,struct Var *v);`
The cost of loading register `r` from variable `v`.

`int cost_save_reg(int r,struct Var *v);`
The cost of storing register `r` into variable `v`.

`int cost_pushpop_reg(int r);`
The cost of storing register `r` during function prologue and restoring it in the epilogue.

`int cost_savings(struct IC *p,int r,struct obj *o);`

Estimate the savings which would be obtained if the object `o` in IC `p` would be assigned to register `r` (in this IC). If the backend was not able to emit code in this case, `INT_MIN` must be returned.

If `(o->flags & VKONST) != 0`, the register allocator is thinking about putting a constant (or address of a static variable) in a register. In this case, the real object which would be put in a register is found in `o->v->cobj`.

The unit of the costs can be chosen by the backend, but should be some reasonable small values.

If `regok()` is called with a third parameter of -1, it is possible to return non-zero for a register which cannot perform all operations. The register allocator will call `cost_savings()` and returning `INT_MIN` can be used to prevent this register from being allocated, if the register is not suitable for a certain operation.

16.8.11 Inter-procedural Register-Allocation

To support inter-procedural register allocation, the backend must tell the optimizer which registers are used by a function. As the backend might use some registers internally, the frontend can not know this.

Apart from defining `HAVE_REGS_MODIFIED` in `machine.h`, the backend has to mark all registers that are modified in the bitfield `regs_modified`. A register can be marked with `BSET(regs_modified,<reg>)`. For a call IC, the function `calc_regs()` (from `supp.h`) can be called to mark the registers used by a call IC. It will return 1 if it was able to determine all registers used by this IC.

If the register usage could be determined for the entire function, the backend can set the bit `ALL_REGS` in the fi-member of the function variable (`v->fi->flags|=ALL_REGS;`).

16.8.12 Conditional Instructions

FIXME: To be written.

16.8.13 Extended ICs

If the backend defines `HACE_EXT_IC`, it has to define a `struct ext_ic` in `machine.h`. This structure will be added to each IC and can be used by the backend for private use.

16.8.14 Peephole Optimizations on Assembly Output

Some optimizations are easier to do on the generated assembly code rather than doing them before emitting code. Therefore it is possible to do peephole optimizations on the emitted code before it is really written to a file.

`EMIT_BUF_DEPTH` lines will be stored in a ring buffer and are available to examination and modification by a function `emit_peephole()`. The actual assembly output is stored in `emit_buffer`, the index of the first line to be output in `emit_f` and the index of the last one in `emit_l` (note that you have to calculate modulo `EMIT_BUF_DEPTH` - it is a ring buffer).

The output may be modified in memory and the first line may be removed using `remove_asm()`. If a modification took place, a non-zero value has to be returned (0 otherwise). The following example code would combine two consecutive additions to the same register:

```
int emit_peephole(void)
```

```
{
  int entries,i,r1,r2;
  long x,y;
  /* pointer to the lines in order of output */
  char *asmline[EMIT_BUF_DEPTH];
  i=emit_l;
  /* compute number of entries in ring buffer */
  if(emit_f==0)
    entries=i-emit_f+1;
  else
    entries=EMIT_BUF_DEPTH;
  /* the first line */
  asmline[0]=emit_buffer[i];
  if(entries>=2){
    /* we have at least two line sin the buffer */
    /* calculate the next line (modulo EMIT_BUF_DEPTH) */
    i--;
    if(i<0) i=EMIT_BUF_DEPTH-1;
    asmline[1]=emit_buffer[i];
    if(sscanf(asmline[0],"\tadd\tR%d,#%ld",&r1,&x)==2&&
       sscanf(asmline[1],"\tadd\tR%d,#%ld",&r2,&y)==2&&
       r1==r2){
      sprintf(asmline[1],"\tadd\tR%d,#%ld\n",r1,x+y);
      remove_asm();
      return 1;
    }
  }
  return 0;
}
```

Be very careful when doing such optimizations. Only perform optimizations which are really legal. Especially assembly code often has side effects like setting of flags.

Depending on command line flags inline assembly code may or may not be passed through this peephole optimizer. By default, it will be done, enabling optimizations between generated code and inline assembly.

16.8.15 Marking of efficient ICs

If the backend sets `HAVE_EFF_ICS` in `machine.h`, it has to provide a function `void mark_eff_ics(void)`. This function will be called (possibly multiple times) by the frontend. The function has to set or clear the bit `EFF_IC` in the member `flags` of every IC.

The flag should be set when the operation is in a context that suggests it will translate to efficient machine code. The optimizer will transform this IC less aggressively.

As this is all happens before register allocation, the decision is of a very heuristic nature.

16.8.16 Function entry/exit Code

At entry and exit of function, there is usually some code to set up the new environment for this function. For example, registers will have to be saved/restored, a frame pointer

may be set up and a stack frame will be created. It is generally worthwile to optimize this entry/exit code. For example, if no registers need to be saved and no local variables are used on the stack, it may not be necessary to create a stack frame.

The exact possibilities for optimization depend on the architecture and the ABI.

16.8.17 Multiplication/division with Constants

Many architectures do not provide instruction for multiplication, division or modulo calculation. And on most architectures providing such instructions they are rather slow. Therefore, it is recommended to emit cheaper instructions, if possible.

Usually, this can only be done if one operand of the operation is a constant. Multiplications may be replaced by a series of shift and add instructions, for example. The simplest and most important cases to replace are multiplication, division and modulo with a power of two. Multiplication by x can be replaced by a left shift of log2(x), unsigned division of x can be replaced by logical right shift of log2(x) and unsigned modulo by x can be replaced by anding with x-1.

Note that signed division and modulo can usually not be replaced that simple because most division instructions give different results for some negative values. An additional adjustment would be necessary to get correct results. Whether this is still an improvement, depends on the architecture details.

The following function can be used to test whether a value is a power of two:

```
static long pof2(zumax x)
/*  Yields log2(x)+1 or 0. */
{
  zumax p;int ln=1;
  p=ul2zum(1L);
  while(ln<=32&&zumleq(p,x)){
    if(zumeqto(x,p)) return ln;
    ln++;p=zumadd(p,p);
  }
  return 0;
}
```

16.8.18 Block copying

There are many cases of copying of larger data. For the backend, those will mostly be used in `PUSH` and `ASSIGN` ICs. It is very important to implement those as efficient as possible.

Some things to consider:

- When alignment is known, use word-copy instead of byte-copy.
- Copy small blocks by a series of copy instructions.
- For larger blocks, loading addresses in registers may help.
- For large blocks, use a loop. Implement it efficiently and try to unroll the loop a few times.
- For very large blocks, calling a library function may be useful. While this creates some overhead, the function can dynamically check the alignment or perhaps even use special hardware, if available.

– Set `INLINEMEMCOPY` to reasonable values. Set it to a very high value if you implement very good block copying.

16.8.19 Optimized Library Functions

FIXME: To be written.

16.8.20 Instruction Scheduler

FIXME: To be written.

16.9 Hints for common Extensions

This section lists some common extensions to the C language which are often very helpful when using a compiler in practice. Depending on the kind of target system they may range from nobody-really-cares to absolutely essential. For example, consider the ability to specify the section within an object file a variable or function should be placed in. This is rarely of any interest when targeting a Unix-like operating system. On a stand-alone embedded system, however, it may be absolutely necessary.

Therefore, consider this list as a recommendation of ideas that might be helpful.

16.9.1 Inline Assembly

The possibility to insert assembly code into C source is very handy in many cases. It can be used in headers to implement specially optimized versions of time-critical library routines or enable access to CPU features which are not otherwise accessible by normal C constructs.

In general, almost all work is done by the frontend and only a few lines have to be inserted in the backend to make it work. Therefore, it is recommended to always support this important feature.

Everything that has to be done is to check a certain condition when code for a `CALL` IC is generated. Instead of emitting a normal call instruction, call the `emit_inline_asm()` function:

```
if((p->q1.flags & (VAR|DREFOBJ)) == VAR &&
    p->q1.v->fi &&
    p->q1.v->fi->inline_asm){
        emit_inline_asm(f,p->q1.v->fi->inline_asm);
    }else{
        emit(f,"\tcall\t");
        emit_obj(f,&p->q1,t);
        emit(f,"\n");
    }
```

Note that argument-passing, adjusting the stack after a `CALL` IC etc. is not affected. Only the actual emitting of call code is changed in the case of inline assembly.

16.9.2 -speed/-size

Often it is desired to generate code which runs as fast as possible but sometimes small code is needed. The command line options `-speed` and `-size` are provided for the user to specify his intention.

These options already may change the intermediate code produced by the frontend, but the backend should also respect these switches, if possible. The variables `optspeed` and `optsize` can be queried to see if these options were specified.

If e.g. `optspeed` was specified, the backend should choose faster code-sequences, even if code-size is increased significantly. Vice-versa, if `optsize` is specified, it should always choose the shorter code if there is a trade-off between size and speed.

Typical cases for such tradeoffs are for example, block-copy (`ASSIGN` and `PUSH`) ICs. Often it is possible to call a library function or generate a simple short loop for small code, but an unrolled inlined loop for fast code.

16.9.3 Target-specific Macros

A backend is able to provide macro definitions which are automatically active. It is recommended to define macros which allow applications to query the target architecture and the selected chip (if possible). Also, it is recommended to provide internal macros for backend specific attributes using the `__attr()` and `__vattr()` attributes.

The definition of these macros can be done in `init_cg()` (the results of command line parsing are available at this point). There is a variable

```
char **target_macros;
```

which can be set to an array of pointers to strings which contain the macro definitions. The array has to be terminated by a null pointer and the syntax of the macro definitions is similar to the command line option `-D`:

```
static char *marray[] = {
  "__TARGET_ARCH__",
  "__section(x)=__vattr(\"section(\"#x\")\")",
  0
};
...
target_macros = marray;
```

16.9.4 stdarg.h

FIXME: To be written.

16.9.5 Section Specifiers

Especially for embedded systems it can be very important to be able to place variables and functions in specific section to override default placement. This can relatively easily be done using variable attributes (see Section 16.9.6 [targetattributes], page 184).

16.9.6 Target-specific Attributes

There are two ways of adding target-specific attributes to variables and functions. A general way is the use of `__vattr()` which will add the string argument to the `vattr` member of the corresponding `struct Var`, separating it by a semi-colon. The backend can use this information by parsing the string. The frontend will just build the string, it will not interpret it. If a backend offers attributes using the `__vattr()` mechanism, it is recommended to provide target-specific macros (see Section 16.9.3 [targetmacros], page 184) which expand to the appropriate `__vattr()`-syntax. Only these macros should be documented.

A second way to specify attributes is enabled by adding

```
#define HAVE_TARGET_PRAGMAS 1
```

to `machine.h` and adding an array

```
char *g_attr_name[];
```

to `machine.c`. This array should point to the strings used for the attributes, terminated by a null-pointer, e.g.:

```
char *g_attr_name[] = {
  "__far",
  "__near",
  "__interrupt",
  0
};
```

These attributes can be queried in the member

```
unsigned long tattr;
```

of a `struct Var`. The first attribute is represented by bit 1, the second by bit 2 and so on. Using this mechanism, the frontend will check for redeclarations with different setting of attributes or multiple specification of attributes. However, only boolean attributes are possible. If parameters have to be specified, the `__vattr()`-mechanism has to be used.

16.9.7 Target-specific `#pragmas`

FIXME: To be written.

16.9.8 Target-specific extended Types

FIXME: To be written.

16.9.9 Target-specific `printval`

FIXME: To be written.

16.9.10 Debug Information

Debug information which enables (source level) debugging of compiled programs is an important feature to improve the user-friendliness of a compiler. Depending on the object format and debugger used, the format and capabilities of debug information can vary widely. Therefore, it is the responsibility of each backend to generate debug information. However, for common debug standards there will be modules which can be used by the backends and will do most of the work. Currently there is one such module for the DWARF2 debug standard.

The compiler frontend provides a variable `debug_info` which can be queried to test whether debug information is desired. Also, the functions `init_db()` and `cleanup_db()` are helpful.

Each `struct Var` contains the members `char *dfilename` and `int dline` which specify the file and line number of the variable's definition. Also, every IC contains the members `char *file` and `int line` with the file name and line number this IC belongs to. Note however, that there may be ICs with `file == 0` - not all ICs can be assigned a certain code location. Also, ICs do not always have increasing line numbers and line numbers may repeat. Not all debuggers may be able to deal with this.

16.9.10.1 DWARF2

There is support for the DWARF2 debug standard which can be added to a backend rather easily. The following additions are necessary:

1. Add the line
   ```
   #include "dwarf2.c"
   ```
   to `machine.c`.
2. Add the following lines to `init_db()`:
   ```
   dwarf2_setup(sizetab[POINTER],
                ".byte",
                ".2byte",
                ".4byte",
                ".4byte",
                labprefix,
                idprefix,
                ".section");
   dwarf2_print_comp_unit_header(f);
   ```
   The arguments to `dwarf2_setup()` have the following meanings:
   1. The size of an address on the target.
   2. An assembler directive to create one byte of initialized storage.
   3. An assembler directive to create two bytes of initialized storage (without any padding for alignment).
   4. An assembler directive to create four bytes of initialized storage (without any padding for alignment).
   5. An assembler directive to create initialized storage representing a target address (without any padding for alignment).
   6. A prefix which is used for emitting numbered labels (or empty string).
   7. A prefix which is used for emitting external identifiers (or empty string).
   8. An assembler directive to switch to a new named section.
3. Add the line
   ```
   dwarf2_cleanup(f);
   ```
   to `cleanup_db()`.
4. Write the function
   ```
   static int dwarf2_regnumber(int r);
   ```
   which returns the DWARF2 regnumber for a `vbcc` register number.
5. Write the function
   ```
   static zmax dwarf2_fboffset(struct Var *v);
   ```
   which returns the offset of variable `v` from the DWARF2 frame-pointer.
6. Write the function
   ```
   static void dwarf2_print_frame_location(FILE *f,struct Var *v);
   ```
   which prints a DWARF2 location of the frame pointer. It can use the function
   ```
   void dwarf2_print_location(FILE *f,struct obj *o);
   ```

to output the location. For example, if the frame pointer is a simple register, it might look like this:

```
static void dwarf2_print_frame_location(FILE *f,struct Var *v)
{
  struct obj o;
  o.flags=REG;
  o.reg=frame_pointer_register;
  o.val.vmax=l2zm(0L);
  o.v=0;
  dwarf2_print_location(f,&o);
}
```

7. Before emitting code for an IC `p`, execute the code

   ```
   if(debug_info)
     dwarf2_line_info(f,p);
   ```

8. After emitting code for a function `v`, a new numbered label has to be emitted after the function code and the function

   ```
   void dwarf2_function(FILE *f,struct Var *v,int endlabel);
   ```

   must be called.

Note that the DWARF2 standard supports use of location lists which can be used to describe a variable whose location changes during the program (e.g. in a register for some time, then in memory and again in a register) as well as a moving frame pointer (very useful if no separate frame pointer is used but all local variables are accessed through a moving stack pointer). Unfortunately, none of the debuggers I have tried so far could handle these location lists. Therefore, the current DWARF2 module does not output location lists, but future version will probably offer them as an option.

Without location lists, accessing local variables will only work with a fixed frame pointer and no register variables. Even with these restrictions, function parameters which are passed in registers will not be correctly displayed during the function entry code.

16.9.11 Interrupt Handlers

Especially for embedded systems, support for writing interrupt handlers in C is a common feature. Variable attributes (see Section 16.9.6 [targetattributes], page 184) can be used to mark functions which are used as interrupt handlers.

Typical changes which might be necessary for interrupt handlers are:

- Using a different return instruction.
- Saving all modified registers, including scratch-registers.
- Creating an entry in the interrupt vector table.

16.9.12 Stack checking

Dynamic checking of the stack used (or possibly extending the stack size if possible) is another useful feature. If the variable `stack_check` is set, stack-checking code should be emitted, if possible. Every function should call a library function (usually called `__stack_check`) and pass the maximum size of stack used in this function as argument. This obviously has to be done before allocating the stack-frame.

The library function is responsible to take into account its own stack-frame.

16.9.13 Profiling

FIXME: To be written.

16.9.14 Variable-length Arrays

With the `-c99` option, vbcc supports variable-length arrays that are allocated on the stack. The backend has to take several steps to support this:

`vlas`
: When this variable is non-zero, the current function uses variable-length arrays. The backend may take necessary steps to support this. For example, if local variables are usually addressed via stackpointer, switching to a separate framepointer may be necessary.

`ALLOCVLA_INLINEASM`
: This define must contain inline code that is called when a vla is allocated. It has to create additional room on the stack and return a pointer to the beginning of the new space.

`ALLOCVLA_REG`
: The register in which to pass the size to be allocated on the stack. 0 will pass on the stack.

`FREEVLA_INLINEASM`
: This define must contain inline code that is called when a vla is freed. It has to restore the old stack pointer.

`FREEVLA_REG`
: The register in which to pass the old stack pointer. 0 will pass on the stack.

`OLDSPVLA_INLINEASM`
: This define must contain inline code that is called before the first vla is allocated. It has to return the current stack pointer before any vla has been allocated.

`FPVLA_REG`
: An additional register used in functions containing vlas. The backend can specify a register (usually framepointer) that can not be used in functions with vlas. Therefore, it is possible to use this register in other functions (for example, if local variables are usually addressed directly through the stackpointer).

16.9.15 Library Calls

Sometimes operations may be very complicated to generate code for (e.g. floating-point operations for machines without FPU, multiplication/division on some architectures or big data types like `long long`). Those are usually implemented by calling library functions.

vbcc can be told to generate calls to library functions for certain ICs. When defining `HAVE_LIBCALLS`, the backend must provide the function `char *use_libcall(<code>,<typf>,<typf2>)`. This function gets called with the elements `code`, `typf` and `typf2` of an IC. If `use_libcall` returns a name, this library function will be called instead of the IC. Otherwise, 0 must be returned.

All library functions have to be declared in `init_cg()` with `declare_builtin()`, supporting the following arguments:

`name` The name of the library function. Usually, a reserved identifier should be chosen (e.g. starting with `__`).

`ztyp` The return type of the function (only integral and float types are supported).

`q1typ` The type of the first parameter (only integral and float types are supported).

`q1reg` The register to pass the first argument in (0 passes via stack).

`q2typ` The type of the second parameter (only integral and float types are supported). For functions with a single parameter, use 0.

`q2reg` The register to pass the second argument in (0 passes via stack).

`nosidefx` If this is non-zero, the function will be declared to have no side-effects and allow some more optimizations.

`inline_asm`
Inline assembly can be specified for the function.

Note that not all ICs can be converted to library calls.

16.10 Changes from 0.7 Interface

The backend interface has changed in several ways since `vbcc` 0.7. The following list mentions most(all?) differences between the old and new interfaces (not including new optional features which do not have to be used):

- There are more types (`LLONG, LDOUBLE, MAXINT`). Therefore the `align[]` and `sizetab[]` arrays have dimension `MAX_TYPES+1` rather than 16.
- The representation and access of `t_min[]` and `t_max[]` has been changed.
- `zmax` replaces `zlong` as largest integer target type. `zlong` is only used when actually referring to a `long` on the target. Also, the macros for target arithmetic are available for `zmax/zumax` instead of `zlong/zulong`.
- `PUSH` ICs contain a second size (actual stack-adjustment).
- The second argument of `SHIFT` ICs has an own type.
- `DREFOBJ` objects contain the type of the dereferenced pointer (only meaningful if there are different pointer types).
- The new `CONVERT` IC replaces the series of old ICs (`CONVINT` etc.).
- `emit()`-functions have to be used to generate output rather than `fprintf()`.
- The functions `init_db()` and `cleanup_db()` have to be provided (they do not have do to anything).
- A new array `reg_prio[]` is needed and controls the order in which registers are allocated.
- The parameters of `must_convert()` have changed.
- Static functions must not use identifiers, but have to use numbered labels.

Volker Barthelmann vb@compilers.de