

vasm assembler system

Volker Barthelmann, Frank Wille

February 2026

Table of Contents

1	General	1
1.1	Introduction	1
1.2	Legal	1
1.3	Contact	1
1.4	Installation	1
2	The Assembler	3
2.1	General Assembler Options	3
2.2	Expressions	6
2.3	Symbols	8
2.4	Predefined Symbols	8
2.5	Include Files	8
2.6	Macros	9
2.7	Structures	9
2.8	Conditional Assembly	9
2.9	Known Problems	9
2.10	Credits	9
2.11	Error Messages	11
3	Standard Syntax Module	15
3.1	Legal	15
3.2	Additional options for this module	15
3.3	General Syntax	15
3.4	Operators	16
3.5	Directives	16
3.6	Known Problems	24
3.7	Error Messages	24
4	Mot Syntax Module	25
4.1	Legal	25
4.2	Additional options for this module	25
4.3	General Syntax	26
4.4	Operators	27
4.5	Directives	27
4.6	Known Problems	38
4.7	Error Messages	38
5	Madmac Syntax Module	41
5.1	Legal	41
5.2	General Syntax	41
5.3	Operators	41

5.4 Directives	42
5.5 Known Problems	45
5.6 Error Messages	46
6 Oldstyle Syntax Module	47
6.1 Legal	47
6.2 Additional options for this module	47
6.3 General Syntax	47
6.4 Operators	48
6.5 Directives	48
6.6 Structures	58
6.7 Known Problems	59
6.8 Error Messages	59
7 Test output module	61
7.1 Legal	61
7.2 Additional options for this module	61
7.3 General	61
7.4 Restrictions	61
7.5 Known Problems	61
7.6 Error Messages	61
8 ELF output module	63
8.1 Legal	63
8.2 Additional options for this module	63
8.3 General	63
8.4 Restrictions	63
8.5 Known Problems	63
8.6 Error Messages	63
9 a.out output module	65
9.1 Legal	65
9.2 Additional options for this module	65
9.3 General	65
9.4 Restrictions	65
9.5 Known Problems	65
9.6 Error Messages	66
10 COFF output module	67
10.1 Legal	67
10.2 Additional options for this module	67
10.3 General	67
10.4 Restrictions	67
10.5 Known Problems	67
10.6 Error Messages	67

11	TOS output module	69
11.1	Legal	69
11.2	Additional options for this module	69
11.3	General	69
11.4	Restrictions	69
11.5	Known Problems	70
11.6	Error Messages	70
12	GST output module	71
12.1	Legal	71
12.2	Additional options for this module	71
12.3	General	71
12.4	Restrictions	71
12.5	Known Problems	71
12.6	Error Messages	71
13	Hunk-format output module	73
13.1	Legal	73
13.2	Additional options for this module	73
13.3	General	74
13.4	Restrictions	74
13.5	Known Problems	74
13.6	Error Messages	74
14	X68k output module	77
14.1	Legal	77
14.2	Additional options for this module	77
14.3	General	77
14.4	Restrictions	77
14.5	Known Problems	77
14.6	Error Messages	77
15	O65 output module	79
15.1	Legal	79
15.2	Additional options for this module	79
15.3	General	80
15.4	Restrictions	80
15.5	Known Problems	80
15.6	Error Messages	80

16	AOF output module	81
16.1	Legal	81
16.2	General	81
16.3	Restrictions	81
16.4	Known Problems	81
16.5	Error Messages	81
17	vobj output module	83
17.1	Legal	83
17.2	Additional options for this module	83
17.3	General	83
17.4	Restrictions	84
17.5	Known Problems	84
17.6	Error Messages	84
18	Simple binary output module	85
18.1	Legal	85
18.2	Additional options for this module	85
18.3	General	86
18.4	Known Problems	86
18.5	Error Messages	86
19	Motorola S-Record output module	87
19.1	Legal	87
19.2	Additional options for this module	87
19.3	General	87
19.4	Known Problems	87
19.5	Error Messages	87
20	Intel Hex output module	89
20.1	Legal	89
20.2	Additional options for this module	89
20.3	General	89
20.4	Known Problems	89
20.5	Error Messages	89
21	C #define output module	91
21.1	Legal	91
21.2	Additional options for this module	91
21.3	General	91
21.4	Known Problems	91
21.5	Error Messages	91

22	Project Hans custom output module	93
22.1	Legal	93
22.2	General	93
22.3	Known Problems	93
22.4	Error Messages	93
23	Wozmon output module	95
23.1	Legal	95
23.2	Contact	95
23.3	Additional options for this module	95
23.4	General	95
23.5	Known Problems	95
23.6	Error Messages	95
24	MOS paper tape output module	97
24.1	Legal	97
24.2	Additional options for this module	97
24.3	General	97
24.4	Known Problems	97
24.5	Error Messages	97
25	M68k cpu module	99
25.1	Legal	99
25.2	Additional options for this module	99
25.2.1	CPU selections	99
25.2.2	Optimization options	100
25.2.3	Other options	102
25.3	General	104
25.4	Internal symbols	104
25.5	Extensions	105
25.6	Optimizations	110
25.6.1	Operand optimizations	110
25.6.2	Instruction optimizations	111
25.7	Known Problems	114
25.8	Error Messages	114
26	PowerPC cpu module	117
26.1	Legal	117
26.2	Additional options for this module	117
26.3	General	118
26.4	Extensions	118
26.5	Optimizations	118
26.6	Known Problems	119
26.7	Error Messages	119

27	c16x/st10 cpu module	121
27.1	Legal	121
27.2	Additional options for this module	121
27.3	General	121
27.4	Extensions	121
27.5	Optimizations	122
27.6	Known Problems	122
27.7	Error Messages	122
28	6502 cpu module	123
28.1	Legal	123
28.2	Additional options for this module	123
28.3	General	124
28.4	Extensions	124
28.5	Optimizations	126
28.6	Known Problems	126
28.7	Error Messages	126
29	SPC700 cpu module	127
29.1	Legal	127
29.2	Additional options for this module	127
29.3	General	127
29.4	Extensions	127
29.5	Optimizations	128
29.6	Known Problems	128
29.7	Error Messages	128
30	ARM cpu module	131
30.1	Legal	131
30.2	Additional options for this module	131
30.3	General	132
30.4	Extensions	132
30.5	Literal Pool	133
30.6	Optimizations	133
30.7	Known Problems	133
30.8	Error Messages	134
31	80x86 cpu module	135
31.1	Legal	135
31.2	Additional options for this module	135
31.3	General	135
31.4	Extensions	136
31.5	Optimizations	136
31.6	Known Problems	136
31.7	Error Messages	137

32	Z80 cpu module	139
32.1	Legal	139
32.2	Additional options for this module	139
32.3	General	140
32.4	Extensions	140
32.5	Optimisations	140
32.6	Known Problems	141
32.7	Error Messages	141
33	6800 cpu module	143
33.1	Legal	143
33.2	Additional options for this module	143
33.3	General	143
33.4	Extensions	143
33.5	Optimizations	143
33.6	Known Problems	143
33.7	Error Messages	144
34	6809/6309/68HC12 cpu module	145
34.1	Legal	145
34.2	Additional options for this module	145
34.3	General	145
34.4	Extensions	146
34.5	Optimizations	146
34.6	Known Problems	147
34.7	Error Messages	147
35	Jaguar RISC cpu module	149
35.1	Legal	149
35.2	Additional options for this module	149
35.3	General	149
35.4	Optimizations	149
35.5	Extensions	150
35.6	Known Problems	150
35.7	Error Messages	150
36	PDP11 cpu module	151
36.1	Legal	151
36.2	Additional options for this module	151
36.3	General	151
36.4	Known Problems	151
36.5	Error Messages	151

37	unSP cpu module	153
37.1	Legal	153
37.2	General	153
37.3	Extensions	153
37.4	Compatibility with other assemblers and disassemblers	153
37.5	Known Problems	154
38	SWEET16 cpu module	155
38.1	Legal	155
38.2	General	155
38.3	Known Problems	155
38.4	Error Messages	155
39	HANS cpu module	157
39.1	Legal	157
39.2	Additional options for this module	157
39.3	General	157
39.4	Known Problems	157
39.5	Error Messages	157
40	Trillek TR3200 cpu module	159
40.1	Legal	159
40.2	General	159
40.3	Extensions	159
40.4	Known Problems	159
40.5	Error Messages	159
40.6	Example	160
41	Interface	163
41.1	Introduction	163
41.2	Building vasm	163
41.2.1	Directory Structure	163
41.2.2	Adapting the Makefile	163
41.2.3	Building vasm	165
41.3	vasm global variables	166
41.4	General data structures	167
41.4.1	Source	167
41.4.2	Sections	169
41.4.3	Symbols	170
41.4.4	Register symbols	173
41.4.5	Atoms	174
41.4.6	Relocations	179
41.4.7	CPU-specific Relocations	180
41.4.8	Errors	181

41.5	Support Functions	181
41.5.1	Memory Functions	182
41.5.2	Expressions	182
41.5.3	Symbols.....	185
41.5.4	Macros, Structures and Repetitions.....	186
41.5.5	Hash Tables.....	187
41.5.6	Atoms	188
41.6	Syntax modules.....	188
41.6.1	The file <code>syntax.h</code>	188
41.6.2	The file <code>syntax.c</code>	190
41.7	CPU modules.....	192
41.7.1	The file <code>cpu.h</code>	192
41.7.2	The file <code>cpu.c</code>	195
41.8	Output modules.....	197

1 General

1.1 Introduction

vasm is a portable and retargetable assembler able to create linkable objects in different formats as well as absolute code. Different CPU-, syntax and output-modules are supported. Most common directives/pseudo-opcodes are supported (depending on the syntax module) as well as CPU-specific extensions.

The assembler supports optimizations and relaxations. For example, choosing the shortest possible branch instruction or addressing mode as well as converting a branch to an absolute jump if necessary.

It also supports target CPUs with more than 8 bits per byte (word-addressing) but requires that the host system has 8-bit bytes.

The concept is that you get a special vasm binary for any combination of CPU- and syntax-module. All output modules, which make sense for the selected CPU, are included in the vasm binary and you have to make sure to choose the output file format you need (refer to the next chapter and look for the `-F` option). The default is a test output format, only useful for debugging or analyzing the output.

1.2 Legal

vasm is copyright in 2002-2026 by Volker Barthelmann.

This archive may be redistributed without modifications and used for non-commercial purposes.

An exception for commercial usage is granted, provided that the target CPU is M68k and the target OS is AmigaOS. Resulting binaries may be distributed commercially without further licensing.

In all other cases you need my written consent.

Certain modules may fall under additional copyrights.

1.3 Contact

Responsible for the current version of vasm and contact address in case of bug reports or support requests:

- Volker Barthelmann (vb@compilers.de)
- Frank Wille (frank@phoenix.owl.de)

In case you have an issue with a specific cpu-, syntax- or output-module, please contact the module's author first, if possible.

1.4 Installation

The vasm binaries do not need additional files, so no further installation is necessary. To use vasm with vbcc, copy the binary to `vbcc/bin` after following the installation instructions for vbcc.

The vasm binaries are named `vasm<cpu>_<syntax>` with `<cpu>` representing the CPU-module and `<syntax>` the syntax-module, e.g. `vasm` for PPC with the standard syntax module is called `vasmppc_std`.

Sometimes the syntax-modifier may be omitted, e.g. `vasmppc`.

Detailed instructions how to build vasm can be found in the last chapter.

2 The Assembler

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

2.1 General Assembler Options

`vasm` is run from the command line using the following syntax:

```
vasm<target>_<syntax> [options] [sourcefile]
```

When the source file name is missing, the assembler reads the source text from `stdin` until encountering EOF (end-of-file, which is CTRL-D in Unix shells, or CTRL-\ in AmigaOS shells). Note, that most debugging formats (DWARF, etc.) no longer function with such temporary source texts.

The following options are supported by the machine independent part of `vasm`. The most important ones are:

- D<name>[=<expression>]
Defines a symbol with the name <name> and assigns the value of the expression when given. The assigned value defaults to 1 otherwise.
- F<fmt>
Use module <fmt> as output driver. See the chapter on output drivers for available formats and options.
- I<path>
Define another include path. They are searched in the order of occurrence on the command line, and always before any include paths defined in the source.
- L <listfile>
Enables generation of a listing file and directs the output into the file <listfile>.
- nocase
Disables case-sensitivity for symbols. This doesn't necessarily include macro and structure names, which depend on the syntax-module in use. Mnemonics and directives are usually case-insensitive by default.
- nosym
Strips all local symbols from the output file and doesn't keep any other symbols than those which are required for external linkage.
- o <ofile>
Write the assembler output to <ofile> rather than `a.out`.
- quiet
Do not print the copyright notice and the final statistics.
- x
Show an error message when referencing an undefined symbol. The default behaviour is to declare such symbols as externally defined.

Other options:

- depend=<type>
Print all dependencies while assembling the source with the given options. No output is generated. <type> may be the word `list` for printing one file name

in each new line, or **make** for printing a sequence of file names on a single line, suitable for Makefiles. If the output file name is given by **-o outname** then **vasm** will also print **outname:** in front of it. Note that in contrast to option **-dependall** only relative include file dependencies will be listed (which is the common case).

-dependall=<type>

Prints dependencies in the same way as **-depend**, but adds the include files using absolute paths to it.

-depfile <filename>

Used together with **-depend** or **-dependall** and instructs **vasm** to output all dependencies into a new file, instead of stdout. Additionally, code will be generated in parallel to the dependencies output.

-dwarf[=<version>]

Automatically generate DWARF debugging sections, suitable for source level debugging. When the version specification is missing, DWARF V3 will be emitted. The only difference to V2 is that it creates a **.debug_ranges** section, with address ranges for all sections, instead of using a workaround by specifying **DW_AT_low_pc=0** and **DW_AT_high_pc=~0**. Note, that when you build **vasm** from source, you may have to set your host operating system with **-Dname** in the Makefile to include the appropriate code which can determine the current work directory. Otherwise the default would be to set the current work directory to an empty string. Currently supported are: **AMIGA**, **ATARI**, **MSDOS**, **UNIX**, **_WIN32**. MacOSX works with **UNIX**.

-esc Enable escape character sequences. This will make **vasm** treat the escape character **** in string constants similar as in the C language.

-ibe Use big-endian order when reading target-bytes with more than 8 bits per byte from the host's file system (default).

-ignore-mult-inc

When the same file is included multiple times, using the same path, this is silently ignored, causing the file to be processed only once. Note, that you can still include the same file twice when using different paths to access it.

-ile Use little-endian order when reading target-bytes with more than 8 bits per byte from the host's file system.

-Lall List all symbols, including unused equates. Default is to list all labels and all used expressions only.

-Lbpl=<n>

Set the maximum number of bytes per line in a listing file to **<n>**. Defaults to 8 (**fmt=wide**).

-Lfmt=<fmt>

Set the listing file format to **<fmt>**. Defaults to **wide**. Available are: **wide**, **old**.

-Llo Show only program labels in the sorted symbol listing. Default is to list all symbols, including absolute expressions.

- Lni** Do not show included source files in the listing file (fmt=**wide**).
- Lns** Do not include symbols in the listing file (fmt=**wide**).
- maxerrors=<n>**
Sets the maximum number of errors to display before assembly is aborted. When <n> is 0 then there is no limit. Defaults to 5.
- maxmacrecurs=<n>**
Defines the maximum number of recursion levels within a macro. Defaults to 1000.
- maxpasses=<n>**
Adjusts the maximum number of passes while resolving a section. Defaults to 1500.
- no-msrcdebug**
Disable source level debugging within a macro context. When generating output with source level debug information, the debugger will show the invoking source text line instead.
- nocompdir**
Do not search for include files relative to the compile directory (where the main input source is located).
- noesc** No escape character sequences. This will make vasm treat the escape character \ as any other character. Might be useful for compatibility. This is the default in most modules.
- noialign**
Perform no automatic alignment for instructions. Note, that unaligned instructions can make your code crash when executed! Only set if you know what you are doing!
- nomsg=<n>**
Disable the informational message <n>. <n> has to be the number of a valid informational message, like an optimization message.
- nowarn=<n>**
Disable warning message <n>. <n> has to be the number of a valid warning message, otherwise an error is generated.
- obe** Write target-bytes with more than 8 bits per byte in big-endian order to the host's file system (default).
- ole** Write target-bytes with more than 8 bits per byte in little-endian order to the host's file system.
- pad=<value>**
The given padding value can be one or multiple bytes (up to the cpu-backend's address size). It is used for alignment purposes and to fill gaps between absolute ORG sections in the binary output module. Defaults to a zero-byte.
- pic** Try to generate position independent code. Every address needing relocation is flagged by an error message. This option overrides the absolute default mode

(ORG 0) of binary output modules and doesn't work if you force the assembler into absolute mode by such a directive.

-relpath Do not interpret a source path starting with '/' or '\', or including a colon, as absolute, but always attach it relative to defined include paths first.

-underscore

Add a leading underscore in front of all imported and exported (also common, weak) symbol names, just before writing the output file.

-unnamed-sections

Sections are no longer distinguished by their name, but only by their attributes. Which has the effect that when defining a second section with a different name, but same attributes as a first one, it will switch to the first, instead of starting a new section. Is enabled automatically, when using an output-module which doesn't support section names. For example: aout, tos, xfile.

-unsshift

The shift-right operator (>>) treats the value to shift as unsigned, which has the effect that only 0-bits are inserted on the left (MSB) side. The number of bits in a value depend on the target address type (refer to the appropriate cpu module documentation). This may already be the default for some syntax modules, like for example mot-syntax in Devpac-compatibility mode.

-uspc=<value>

Uninitialized memory regions, declared by "space" directives (.space in std-syntax, ds in mot-syntax, etc.) are filled with the given value. Defaults to zero.

-v Print version and copyright messages from the assembler and all its modules, then exit.

-w Hide all warning messages.

-wfail The return code of vasm will no longer be 0 (success), when there was a warning. Errors always make the return code non-zero (failure).

Note, that while most options allow an argument without any separating blank, some do require it (e.g. -o and -L).

2.2 Expressions

Standard expressions are usually evaluated by vasm's core routines rather than by one of the modules (unless this is necessary).

All expressions evaluated by the frontend are calculated in terms of target address values, i.e. the range depends on the backend. Constants which exceed the target address range may be supported by some backends up to 128 bits.

Backends also have the option to support floating point constants directly and convert them to a backend-specific format which is described in the backend's documentation.

Warning: Be aware that the quality and precision of the backend's floating point output depends on the combination of host- and backend-format! If you need absolute precision, encode the floating point constants yourself in binary.

The available operator tokens may be extended or modified by the current syntax module in use, but the default is to use the same operators as in the C programming language.

Note, though, that the operator priority and associativity may be different from C, to comply with common assembler behaviour. The expression evaluation priorities, from highest to lowest, are:

1. `+` `-` `!` `~` (unary `+`/`-` sign, not, bitwise complement)
2. `<<` `>>` (shift left, shift right)
3. `&` (bitwise and)
4. `^` (bitwise exclusive-or)
5. `|` (bitwise inclusive-or)
6. `*` `/` `%` (multiply, divide, modulo)
7. `+` `-` (add, subtract)
8. `<` `>` `<=` `>=` (less, greater, less or equal, greater or equal)
9. `==` `!=` (equality and inequality)
10. `&&` (logical and)
11. `||` (logical or)

Operands are integral values of the target address type. They can either be specified as integer constants of different bases (see the documentation on the syntax module to see how the base is specified) or character constants. Character constants are introduced by `'` or `"` and have to be terminated by the same character that started them.

Multiple characters are allowed and a constant is built according to the endianness of the target.

When the `-esc` option was specified, or automatically enabled by a syntax module, `vasm` interprets escape character sequences as in the C language:

<code>\\</code>	Produces a single <code>\</code> .
<code>\b</code>	The bell character.
<code>\e</code>	Escape character (27).
<code>\f</code>	Form feed.
<code>\n</code>	Line feed.
<code>\r</code>	Carriage return.
<code>\t</code>	Tabulator.
<code>\"</code>	Produces a single <code>"</code> .
<code>\'</code>	Produces a single <code>'</code> .
<code>\<octal-digits></code>	One character with the code specified by the digits as octal value.
<code>\x<hexadecimal-digits></code>	One character with the code specified by the digits as hexadecimal value.
<code>\X<hexadecimal-digits></code>	Same as <code>\x</code> .

Note, that the default behaviour of vasm has changed since V1.7! Escape sequence handling has been the default in older versions. This was changed to improve compatibility with other assemblers. Use `-esc` to assemble sources with escape character sequences. It is still the default in the `std` syntax module, though.

2.3 Symbols

You can define as many symbols as your available memory permits. A symbol may have any length and can be of global or local scope. Internally, there are three types of symbols:

Expression

These symbols are usually not visible outside of the source, unless they are explicitly exported. Note, that some object file formats use exported expressions referencing an imported symbol to represent indirect symbols.

Label Labels are always addresses within a program section. By default they have local scope for the linker.

Imported These symbols are externally defined and must be resolved by the linker.

2.4 Predefined Symbols

Beginning with vasm V1.5c at least one expression symbol is always defined to allow conditional assembly depending on the assembler being used: `__VASM`. Its value depends on the selected cpu module.

Since V1.8i there may be a second internal symbol which reflects the format of the paths in the host file system. Currently there may be one of:

`__UNIXFS` Host file system uses Unix-style paths.

`__MSDOSFS`

Host file system uses MS-DOS-, Windows-, Atari-style paths.

`__AMIGAFS`

Host file system uses AmigaDOS-style paths.

Note that such a path-style symbol must be configured by a `-D` option while compiling vasm from source. Refer to the section about building vasm (Interface chapter) for a listing of all supported host OS options.

There may be other internal symbols, which are defined by the syntax- or by the cpu module.

2.5 Include Files

Vasm supports include files and defining include paths. Whether this functionality is available depends on the syntax module, which has to provide the appropriate directives.

On startup vasm defines one or two default include paths: the current work directory and, when the main source is not located there, the compile directory.

Include paths are searched in the following order:

1. Current work directory.
2. Compile directory (path to main source).

3. Paths specified by `-I` in the order of occurrence on the command line.
4. Paths specified by directives in the source text (in the order of occurrence).

Additionally, all the relative paths, defined by `-I` or directives, are first appended to the current work directory name, then to the compile directory name, while searching for an include file.

Searching for include files in paths based on the compile directory can be completely disabled by `-nocompdir`.

2.6 Macros

Macros are supported by `vasm`, but the directives for defining them have to be implemented in the syntax module. The assembler core supports 9 macro arguments by default to be passed in the operand field, which can be extended to any number by the syntax module. They can be referenced within the macro either by name (`\name`) or by number (`\1` to `\9`), or both, depending on the syntax module. Recursions and early exits are supported.

Refer to the selected syntax module for more details.

2.7 Structures

Vasm supports structures, but the directives for defining them have to be implemented in the syntax module.

2.8 Conditional Assembly

Has to be provided completely by the syntax module.

2.9 Known Problems

Some known module-independent problems of `vasm` at the moment:

- None.

2.10 Credits

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

- Jordan Zebor
- Joseph Zatarski
- Brian Woodruff
- Frank Wille
- Jim Westfall
- Bernard Thibault
- Dimitri Theulings
- Steven Tattersall
- Yannick Stamm

- Jens Schönfeld
- Ross
- Henryk Richter
- Sebastian Pachuta
- Thorsten Otto
- Esben Norby
- Tom Noorduin
- Gunther Nikl
- George Nakos
- Timm S. Mueller
- Gareth Morris
- Dominic Morris
- Garry Marshall
- Jean-Paul Mari
- Mauricio Muñoz Lucero
- Grzegorz Mazur
- Jörg van de Loo
- Robert Leffmann
- Andreas Larsson
- Miro Kropáček
- Olav Krömeke
- Christoph Krc
- Chester Kollschen
- Richard Körber
- Mikael Kalms
- Mark Jones
- Bert Jahn
- Daniel Illgen
- Jerome Hubert
- Matthew Hey
- Stefan Haubenthal
- Søren Hannibal
- John Hankinson
- Philippe Guichardon
- Luis Panadero Guardado
- Romain Giot
- Daniel Gerdgren
- François Galea
- Tom Duin

- Adrien Destugues
- Kieran Connell
- Jae Choon Cha
- Fernando Cabrera
- Patrick Bricout
- Matthias Bock
- Simone Bevilacqua
- Karoly Balogh
- Patrik Axelsson
- Anomie-p

2.11 Error Messages

The frontend has the following error messages:

- 1: illegal operand types
- 2: unknown mnemonic `<%s>`
- 3: unknown section `<%s>`
- 4: no current section specified
- 5: internal error `%d` in line `%d` of `%s`
- 6: symbol `<%s>` redefined
- 7: `%c` expected
- 8: cannot resolve section `<%s>`, maximum number of passes reached
- 9: instruction not supported on selected architecture
- 10: number or identifier expected
- 11: could not initialize `%s` module
- 12: multiple input files
- 13: could not open `<%s>` for input
- 14: could not open `<%s>` for output
- 15: unknown option `<%s>`
- 16: `%s` module doesn't support `%d`-bit bytes
- 17: could not initialize output module `<%s>`
- 18: out of memory
- 19: symbol `<%s>` recursively defined
- 20: fail: `%s`
- 21: section offset is lower than current pc
- 22: target data type overflow (`%d` bits)
- 23: undefined symbol `<%s>`
- 24: trailing garbage after option `-%c`
- 25: missing macro parameters
- 26: missing end directive for macro `"%s"`

- 27: macro definition inside macro "%s"
- 28: maximum number of %d macro arguments exceeded
- 29: option %s was specified twice
- 30: read error on <%s>
- 31: expression must be constant
- 32: initialized data in bss
- 33: missing end directive in repeat-block
- 34: #%d is not a valid warning message
- 35: relocation not allowed
- 36: illegal escape sequence \%c
- 37: no current macro to exit
- 38: internal symbol %s redefined by user
- 39: illegal relocation
- 40: label name conflicts with mnemonic
- 41: label name conflicts with directive
- 42: division by zero
- 43: illegal macro argument
- 44: reloc org is already set
- 45: reloc org was not set
- 46: address space overflow
- 47: bad file-offset argument
- 48: assertion "%s" failed: %s
- 49: cannot declare structure within structure
- 50: no structure
- 51: instruction has been auto-aligned
- 52: macro name conflicts with mnemonic
- 53: macro name conflicts with directive
- 54: non-relocatable expression in equate <%s>
- 55: initialized data in offset section
- 56: illegal structure recursion
- 57: maximum number of macro recursions (%d) reached
- 58: data has been auto-aligned
- 59: register symbol <%s> redefined
- 60: cannot evaluate constant huge integer expression
- 61: cannot evaluate floating point expression
- 62: imported symbol <%s> was not referenced
- 63: symbol <%s> already defined with %s scope
- 64: unexpected "else" without "if"
- 65: unexpected "endif" without "if"

- 66: maximum if-nesting depth exceeded (%d levels)
- 67: "endif" missing for conditional block started at %s line %d
- 68: repeatedly defined symbol <%s>
- 69: macro <%s> does not exist
- 70: register <%s> does not exist
- 71: register symbol <%s> has wrong type
- 72: cannot mix positional and keyword arguments
- 73: undefined macro argument name
- 74: required macro argument %d was left out
- 75: label <%s> redefined
- 76: base %d numerical term expected
- 77: section stack overflow
- 78: section stack is empty
- 79: illegal value for option: %s
- 80: %s backend does not support floating point
- 81: unknown listing file format "%s" ignored
- 82: cannot export equate based on imported symbol: <%s>
- 83: label definition not allowed here
- 84: label defined on the same line as a new section
- 85: no debug output possible with source from stdin
- 86: external symbol <%s> must not be defined
- 87: missing definition for symbol <%s>
- 88: additional macro arguments ignored (expecting %d)
- 89: macro previously defined at line %d of %s

3 Standard Syntax Module

This chapter describes the standard syntax module which is available with the extension `std`.

3.1 Legal

This module is written in 2002-2026 by Volker Barthelmann and is covered by the vasm copyright without modifications.

3.2 Additional options for this module

This syntax module provides the following additional options:

- `-ac` Immediately allocate common symbols in the `.bss/.sbss` section and define them as externally visible.
- `-align` Enforces the backend's natural alignment for all data directives (`.word`, `.long`, `.float`, etc.).
- `-gas` Enable GNU-as compatibility mode. Currently this will only prevent labels prefixed by a dot to be recognized as local labels and turns `.org` into a pure section-offset.
- `-nodotneeded` Recognize assembly directives without a leading dot (`.`).
- `-sdlimit=<n>` Put data up to a maximum size of `n` bytes into the small-data sections. Default is `n=0`, which means the function is disabled.

3.3 General Syntax

Labels always have to be terminated by a colon (`:`), therefore they don't necessarily have to start at the first column of a line.

Local labels may either be preceded by a `'.'` (unless option `-gas` was given) or terminated by `'$'`, and consist out of digits only. These labels exist and keep their value between two global label definitions.

A special form of reusable "local" labels, independent of global labels, may be defined by using a single digit from 0 to 9. You can reference the nearest previous digit-label with `Nb` and the nearest following digit-label with `Nf`, where `N` is such a digit.

Make sure that you don't define a label on the same line as a directive for conditional assembly (`if`, `else`, `endif`)! This is not supported.

The operands are separated from the mnemonic by whitespace. Multiple operands are separated by comma (`,`).

The character used to introduce comments is usually the semicolon (`;`). Except for the following backends, which change it to a hash (`#`) character: `ppc`, `vidcore`, `x86`.

Example:

```
mylabel:  inst.q1.q2 op1,op2,op3  # comment
```

In expressions, numbers starting with 0x or 0X are hexadecimal (e.g. 0xfb2c). 0b or 0B introduce binary numbers (e.g. 0b1100101). Other numbers starting with 0 are assumed to be octal numbers, e.g. 0237. All numbers starting with a non-zero digit are decimal, e.g. 1239.

C-like escape characters in string constants are allowed by default, unless disabled by `-noesc`.

3.4 Operators

The following operators are new to vasm's standard:

! Bitwise Or-Not (infix operator). `a!b` corresponds to `a|~b` using standard operators.

3.5 Directives

All directives are case-insensitive. The following directives are supported by this syntax module (if the CPU- and output-module allow it):

`.2byte <exp1>[,<exp2>...]`

See `.uahalf`.

`.4byte <exp1>[,<exp2>...]`

See `.uaword`.

`.8byte <exp1>[,<exp2>...]`

See `.uaquad`.

`.ascii <exp1>[,<exp2>,<string1>"...]`

See `.byte`.

`.abort <message>`

Print an error and stop assembly immediately.

`.asciiz "<string1>":["<string2>"]...]`

See `.string`.

`.align <bitorbyte_count>[,<fill>][,<maxpad>]`

Depending on the current CPU backend `.align` either behaves like `.balign` (x86) or like `.p2align` (PPC).

`.balign <byte_count>[,<fill>][,<maxpad>]`

Insert as many fill bytes as required to reach an address which is dividable by `<byte_count>`. For example `.balign 2` would make an alignment to the next 16-bit boundary, on a target with 8-bit addressable memory. The padding bytes are initialized by `<fill>`, when given. The optional third argument defines a maximum number of padding bytes to use. When more are needed then the alignment is not done at all.

`.balignl <byte_count>[,<fill>][,<maxpad>]`

Works like `.balign`, with the only difference that the optional fill value can be specified as a 32-bit word. Padding locations which are not already 32-bit aligned, will cause a warning and padded by zero-bytes.

`.balignw <byte_count>[,<fill>][,<maxpad>]`

Works like `.balign`, with the only difference that the optional fill value can be specified as a 16-bit word. Padding locations which are not already 16-bit aligned, will cause a warning and padded by zero-bytes.

`.byte <exp1>[,<exp2>,"<string1>"...]`

Assign the integer or string constant operands into successive 8-bit memory cells in the current section. Any combination of integer and character string constant operands is permitted.

`.comm <symbol>,<size>[,<align>]`

Defines a common symbol which has a size of `<size>` bytes. The final size and alignment is assigned by the linker, which will use the highest size and alignment values of all common symbols with the same name found. A common symbol is usually allocated in the `.bss` section of the final executable. In case the optional `<align>` argument was not specified the backend's default alignment for the given size will be used.

`.double <exp1>[,<exp2>...]`

Parse one or more IEEE double precision floating point expressions and write them into successive blocks of 64 bits into memory using the backend's endianness.

`.else` Assemble the following block only if the previous `.if` condition was false.

`.elseif <exp>`

Same as `.else` followed by `.if`, but without the need for an `.endif`. Avoids nesting.

`.endif` Ends a block of conditional assembly.

`.endm` Ends a macro definition.

`.endr` Ends a repetition block.

`.equ <symbol>,<expression>`

See `.set`.

`.equiv <symbol>,<expression>`

Assign the `<expression>` to `<symbol>` similar to `.equ` and `.set`, but signals an error when `<symbol>` has already been defined.

`.err <message>`

Print a user error message. Do not create an output file.

`.extern <symbol>[,<symbol>...]`

See `.global`.

`.fail <expression>`

Cause a warning when `<expression>` is greater or equal 500. Otherwise cause an error.

`.file "string"`

Set the filename of the input source. This may be used by some output modules. By default, the input filename passed on the command line is used.

`.float <exp1>[,<exp2>...]`

Parse one or more IEEE single precision floating point expressions and write them into successive blocks of 32 bits into memory using the backend's endianness.

`.global <symbol>[,<symbol>...]`

Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.

`.globl <symbol>[,<symbol>...]`

See `.global`.

`.half <exp1>[,<exp2>...]`

Assign the values of the operands into successive 16-bit words of memory in the current section using the backend's endianness.

`.hidden <symbol>[,<symbol>...]`

Overrides the symbol's ELF default visibility and sets it to "hidden", which means the symbols are not visible to other components. The default visibility is otherwise defined by the symbol's binding.

`.if <expression>`

Conditionally assemble the following lines if <expression> is non-zero.

`.ifeq <expression>`

Conditionally assemble the following lines if <expression> is zero.

`.ifne <expression>`

Conditionally assemble the following lines if <expression> is non-zero.

`.ifgt <expression>`

Conditionally assemble the following lines if <expression> is greater than zero.

`.ifge <expression>`

Conditionally assemble the following lines if <expression> is greater than zero or equal.

`.iflt <expression>`

Conditionally assemble the following lines if <expression> is less than zero.

`.ifle <expression>`

Conditionally assemble the following lines if <expression> is less than zero or equal.

`.ifb <operand>`

Conditionally assemble the following lines when <operand> is completely blank, except an optional comment.

`.ifnb <operand>`

Conditionally assemble the following lines when <operand> is non-blank.

`.ifc <string1>,<string2>`

Conditionally assemble the following lines when <string1> matches <string2>. Empty strings are allowed. Quotes are optional.

- .ifnc <string1>,<string2>**
Conditionally assemble the following lines when <string1> differs from <string2>. Empty strings are allowed. Quotes are optional.
- .ifdef <symbol>**
Conditionally assemble the following lines if <symbol> is defined.
- .ifndef <symbol>**
Conditionally assemble the following lines if <symbol> is undefined.
- .incbin <file>**
Inserts the binary contents of <file> into the object code at this position. When the file size (in 8-bit bytes) is not aligned with the size of a target-byte the missing bits are automatically appended and assumed to be zero. As vasm's internal target-byte endianness for more than 8 bits per byte is big-endian, included binary files are assumed to have the same endianness. Otherwise you have to specify `-ile` to tell vasm that they use little-endian target-bytes (on your 8-bit bytes host file system).
- .incdir <path>**
Add another path to search for include files to the list of known paths. Paths defined with `-I` on the command line are searched first.
- .include <file>**
Include source text of <file> at this position.
- .int <exp1>[,<exp2>...]**
Assign the values of the operands into successive words of memory in the current section using the target's endianness and address size.
- .internal <symbol>[,<symbol>...]**
Overrides the symbol's ELF default visibility and sets it to "internal", which means the symbols are considered to be hidden (not visible to other components), and that some extra, processor specific processing must be performed. The default visibility is otherwise defined by the symbol's binding.
- .irp <symbol>[,<val>...]**
Iterates the block between `.irp` and `.endr` for each <val>. The current <val>, which may be embedded in quotes, is assigned to `\symbol`. If no value is given, then the block is assembled once, with `\symbol` set to an empty string.
- .irpc <symbol>[,<val>...]**
Iterates the block between `.irp` and `.endr` for each character in each <val>, and assign it to `\symbol`. If no value is given, then the block is assembled once, with `\symbol` set to an empty string.
- .lcomm <symbol>,<size>[,<alignment>]**
Allocate <size> bytes of space in the .bss section and assign the value to that location to <symbol>. If <alignment> is given, then the space will be aligned to an address having <alignment> low zero bits or 2, whichever is greater. <symbol> may be made globally visible by the `.globl` directive.
- .list**
The following lines will appear in the listing file, when enabled.

`.local <symbol>[,<symbol>...]`

Flag <symbol> as a local symbol, which means that <symbol> is local for the current file and invisible to other modules in the linking process.

`.long <exp1>[,<exp2>...]`

Assign the values of the operands into successive 32-bit words of memory in the current section using the backend's endianness.

`.macro <name> [<argname1>[=<default>] [,<argname2>...]]`

Defines a macro, which can be referenced by <name>. The macro definition is closed by an `.endm` directive. The argument names, which may be passed to this macro, must be declared directly following the macro name, separated by white-space. You can define an optional default value in the case an argument is left out. Note that macro names are case-insensitive while the argument names are case-sensitive. Within the macro context arguments are referenced by `\argname`. The special argument `\@` inserts a unique id, useful for defining labels. `\()` may be used as a separator between the name of a macro argument and the subsequent text.

`.nolist` This line and the following lines will not be visible in a listing file.

`.org <exp>[,<fill>]`

Before any section directive, and in absence of the `-gas` option, <exp> defines the absolute start address of the following code and <fill> has no meaning. Within a relocatable section <exp> defines the relative offset from the start of this section for the subsequent code. The optional <fill> value is only valid within a section and is used to fill the space to the new program counter (defaults to zero). When <exp> starts with a current-pc symbol followed by a plus (+) operator, then the directive just reserves space (filled with zero).

`.p2align <bit_count>[,<fill>][,<maxpad>]`

Insert as many fill bytes as required to reach an address where <bit_count> low order bits are zero. For example `.p2align 2` would make an alignment to the next 32-bit boundary, when the target has 8-bit addressable memory. The padding bytes are initialized by <fill>, when given. The optional third argument defines a maximum number of padding bytes to use. When more are needed then the alignment is not done at all.

`.p2alignl <bit_count>[,<fill>][,<maxpad>]`

Works like `.p2align`, with the only difference that the optional fill value can be specified as a 32-bit word. Padding locations which are not already 32-bit aligned, will cause a warning and padded by zero-bytes.

`.p2alignw <bit_count>[,<fill>][,<maxpad>]`

Works like `.p2align`, with the only difference that the optional fill value can be specified as a 16-bit word. Padding locations which are not already 16-bit aligned, will cause a warning and padded by zero-bytes.

`.popsection`

Restore the top section from the internal section-stack. Also refer to `.pushsection`.

`.protected <symbol>[,<symbol>...]`

Overrides the symbol's ELF default visibility and sets it to "protected", which means that any references to the symbols from within the components that defines them must be resolved to the definition in that component, even if a definition in another component would normally preempt this. The default visibility is otherwise defined by the symbol's binding.

`.pushsection <name>[, "<attributes>" [[, @<type>] | [, %<type>] | [, <mem_flags>]]]`

Works exactly like `.section`, but additionally pushes the previously active section onto an internal stack, where it may be restored from by the `.popsection` directive.

`.quad <exp1>[, <exp2>...]`

Assign the values of the operands into successive quadwords (64-bit) of memory in the current section using the backend's endianness.

`.rept <expression>`

Repeats the assembly of the block between `.rept` and `.endr` <expression> number of times. <expression> has to be positive.

`.section <name>[, "<attributes>" [[, @<type>] | [, %<type>] | [, <mem_flags>]]]`

Starts a new section named <name> or reactivates an old one. When attributes are given for an already existing section, they must match exactly. The "<attributes>" string may consist of the following characters:

Section Contents:

a	section is allocated in memory
c	section has code
d	section has initialized data
u	section has uninitialized data
i	section has directives (info or offsets section)
n	section can be discarded
R	remove section at link time

Section Protection:

r	section is readable
w	section is writable
x	section is executable
s	section is shareable

Section Alignment: A digit, which is ignored. The assembler will automatically align the section to the highest alignment restriction used within.

Memory attributes:

C	load section to Chip RAM (AmigaOS hunk format)
F	load section to Fast RAM (AmigaOS hunk format)

z load section to zero/direct-page (6502, 65816, 680x, etc.)

Any other attribute will still be accepted by vasm and passed to the output driver (which might ignore it).

The optional **<type>** argument is mainly used for ELF output and may be introduced either by a **'%'** or a **'@'** character. Allowed are:

progbits This is the default value, which means the section data occupies space in the file and may have initialized data.

nobits These sections do not occupy any space in the file and will be allocated filled with zero bytes by the OS loader.

When the optional, non-standard, **<mem_flags>** argument is given it defines a 32-bit memory attribute, which defines where to load the section (platform specific). The memory attributes are currently only used in the hunk-format output module.

.set <symbol>,<expression>

Create a new program symbol with the name **<symbol>** and assign to it the value of **<expression>**. If **<symbol>** is already assigned, it will contain a new value from now on.

.size <symbol>,<size>

Defines the size in bytes associated with the given **<symbol>**. This information is only used by some object file formats (for example ELF) and typically sets the size of function symbols.

.short <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit words of memory in the current section using the backend's endianness.

.single <exp1>[,<exp2>...]

Parse one or more IEEE single precision floating point expressions and write them into successive blocks of 32 bits into memory using the backend's endianness.

.skip <exp>[,<fill>]

Insert **<exp>** zero or **<fill>** bytes into the current section.

.space <exp>[,<fill>]

Insert **<exp>** zero or **<fill>** bytes into the current section.

.stabs "<name>",<type>,<other>,<desc>,<exp>

Add a stab-entry for debugging, including a symbol-string and an expression.

.stabn <type>,<other>,<desc>,<exp>

Add a stab-entry for debugging, without a symbol-string.

.stabd <type>,<other>,<desc>

Add a stab-entry for debugging, without symbol-string and value.

.string "<string1>"[, "<string2>"...]

Like **.byte**, but adds a terminating zero-byte.

.swbeg <op>

Just for compatibility. Do nothing.

.type <symbol>,<type>

Set type of symbol named <symbol> to <type>, which must be one of:

1: **Object**

2: **Function**

3: **Section**

4: **File**

The predefined symbols **@object** and **@function** are available for this purpose. Only used by some object file formats (for example ELF).

.uahalf <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.ulong <exp1>[,<exp2>...]

Assign the values of the operands into successive 32-bit areas of memory in the current section regardless of current alignment.

.uaquad <exp1>[,<exp2>...]

Assign the values of the operands into successive 64-bit areas of memory in the current section regardless of current alignment.

.uashort <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.uaword <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit areas of memory in the current section regardless of current alignment.

.weak <symbol>[,<symbol>...]

Flag <symbol> as a weak symbol, which means that <symbol> is visible to all modules in the linking process and may be replaced by any global symbol with the same name. When a weak symbol remains undefined its value defaults to 0.

.word <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit words of memory in the current section using the backend's endianness.

.zero <exp>[,<fill>]

Insert <exp> zero or <fill> bytes into the current section.

Predefined section directives:

.bss .section ".bss","aurw"

.data .section ".data","adrw"

.dpage .section ".dpage","adrwz"

.rodata .section ".rodata","adr"

```
.sbss      .section ".sbss","aurw"  
.sdata     .section ".sdata","adrw"  
.sdata2    .section ".sdata2","adr"  
.stab      .section ".stab","dr"  
.stabstr   .section ".stabstr","dr"  
.text      .section ".text","acrX"  
.tocd      .section ".tocd","adrw"
```

3.6 Known Problems

Some known problems of this module at the moment:

- None.

3.7 Error Messages

This module has the following error messages:

- 1001: mnemonic expected
- 1002: invalid extension
- 1003: no space before operands
- 1004: too many closing parentheses
- 1005: missing closing parentheses
- 1006: missing operand
- 1007: scratch at end of line
- 1008: section flags expected
- 1009: invalid data operand
- 1010: memory flags expected
- 1011: identifier expected
- 1012: assembly aborted
- 1013: unexpected "%s" without "%s"
- 1014: pointless default value for required parameter <%s>
- 1015: invalid section type ignored, assuming progbits
- 1019: syntax error
- 1021: section name expected
- 1022: .fail %lld encountered
- 1023: .fail %lld encountered
- 1024: alignment too big

4 Mot Syntax Module

This chapter describes the Motorola syntax module, mostly used for the M68k and ColdFire families of CPUs, which is available with the extension `mot`.

4.1 Legal

This module is written in 2002-2025 by Frank Wille and is covered by the vasm copyright without modifications.

4.2 Additional options for this module

This syntax module provides the following additional options:

- `-align` Enables natural alignment for data (e.g. `dc`, `ds`) and offset directives (`rs`, `so`, `fo`).
- `-allmp` Makes all 35 macro arguments available. Default are 9 arguments (`\1` to `\9`). More arguments can be accessed through `\a` to `\z`, which may conflict with escape characters or named arguments, therefore they are not enabled by default.
- `-cnop=<code>`
Sets the two-byte code used for alignment padding with `cnop` in code sections. Defaults to `0x4e71` on M68k.
- `-devpac` Devpac-compatibility mode. Only directives known to Devpac are recognized.
 - Enables natural alignment for data and structure offsets (see option `-align`).
 - Predefines offset symbols `__RS`, `__SO` and `__FO` as 0, which otherwise are undefined until first referenced.
 - Disable escape codes handling in strings (see `-noesc`).
 - Enable dots within identifiers (see `-ldots`).
 - Up to 35 macro arguments (see `-allmp`).
 - Do not use NOP instructions when aligning code (see `-cnop=`).
 - Allow a label definition on the same line as a section directive. Note, that in contrast to Devpac this label is not ignored but set to the first address in the section!
- `-ldots` Allow dots (.) within all identifiers.
- `-localu` Local symbols are prefixed by `'_'` instead of `'.'`. For Devpac compatibility, which offers a similar option.
- `-nolocpfx`
Disables local symbols to be recognized by their prefix (usually `'.'`). This allows global symbols to be defined with it. The `'$'` suffix for local symbols still works.
- `-phxass` PhxAss-compatibility mode. Only directives known to PhxAss are recognized.
 - Enable escape codes handling in strings (see `-esc`).

- Macro names are case-insensitive.
 - Up to 35 macro arguments (see **-allmp**).
 - Allow whitespace in operands.
 - Enable dots within identifiers (see **-ldots**).
 - Defines the symbol `_PHXASS_` with value 2 (to differentiate from the real PhxAss with value 1).
 - When no output file name is given, construct it from the input name.
 - Allow a label definition on the same line as a section directive.
- spaces** Allow whitespace characters in the operand field. Otherwise a whitespace would start the comment field there.
- warncomm** Warn about all lines, which have comments in the operand field, introduced by a whitespace character. For example in: `dc.w 1 + 2.`

4.3 General Syntax

Labels must either start at the first column of a line or have to be terminated by a colon (:). In the first case the mnemonic or assembler directive has to be separated from the label by whitespace (not required in any case, e.g. with the `=` directive). A double colon (::) automatically makes the label externally visible (refer to directive `xdef`).

Local labels are either prefixed by `'.'` or suffixed by `'$'`. For the rest, any alphanumeric character including `'_'` is allowed. Local labels are valid between two global label definitions.

Otherwise dots (.) are not allowed within a label by default, unless the option **-ldots** or **-devpac** was specified. Even then, labels ending on `.b`, `.w` or `.l` can't be defined.

It is possible to refer to any local symbol in the source by preceding its name with the name of the last previously defined global symbol: `global_name\local_name`. This is for PhxAss compatibility only, and is no recommended style. Does not work in a macro, as it conflicts with macro arguments.

Make sure that you don't define a label on the same line as a directive for conditional assembly (if, else, endif)! This is not supported and leads to undefined behaviour.

Qualifiers are appended to the mnemonic, separated by a dot (if the CPU-module supports qualifiers). The operands are separated from the mnemonic by whitespace. Multiple operands are separated by comma (,).

In any case, mnemonics and directives must not start on the first column!

The operand field, which is separated by whitespace from the mnemonic/directive field, must not contain any whitespace characters itself, as long as the option **-spaces** was not specified.

Comments can be introduced everywhere by the characters `;` or `*`. The rest of the line will be ignored. Also everything following the operand field, separated by a whitespace, will be regarded as comment (unless **-spaces** was given). Be careful with `*`, which is either recognized as the "current pc symbol" or as a multiplication operation in any operand expression

Example:

```
mylabel inst.q op1,op2,op3 ;comment
```

In expressions, numbers starting with \$ are hexadecimal (e.g. \$fb2c). % introduces binary numbers (e.g. %1100101). Numbers starting with @ are assumed to be octal numbers, e.g. @237. All numbers starting with a digit are decimal, e.g. 1239.

4.4 Operators

The following operators are changed from vasm's standard:

=	Boolean Equality. Additional to standard ==.
//	Modulo. Additional to standard %.
!	Bitwise Or. Additional to standard .
~	Bitwise Exclusive Or. Additional to standard ^.

4.5 Directives

The following directives are supported by this syntax module (provided the CPU- and output-module support them):

```
<symbol> = <expression>
    Equivalent to <symbol> equ <expression>.

<symbol> =.s <expression>
    Equivalent to <symbol> fequ.s <expression>. PhxAss compatibility.

<symbol> =.d <expression>
    Equivalent to <symbol> fequ.d <expression>. PhxAss compatibility.

<symbol> =.x <expression>
    Equivalent to <symbol> fequ.x <expression>. PhxAss compatibility.

<symbol> =.p <expression>
    Equivalent to <symbol> fequ.p <expression>. PhxAss compatibility.

align <bitcount>
    Insert as many zero bytes as required to reach an address where <bitcount> low
    order bits are zero. For example align 2 would make an alignment to the next
    32-bit boundary.

assert <expression>[,<message>]
    Display an error with the optional <message> when the expression is false.

blk.b <exp>[,<fill>]
    Equivalent to dcb.b <exp>,<fill>.

blk.d <exp>[,<fill>]
    Equivalent to dcb.d <exp>,<fill>.

blk.l <exp>[,<fill>]
    Equivalent to dcb.l <exp>,<fill>.
```

<code>blk.q <exp>[,<fill>]</code>	Equivalent to <code>dcb.q <exp>,<fill></code> .
<code>blk.s <exp>[,<fill>]</code>	Equivalent to <code>dcb.s <exp>,<fill></code> .
<code>blk.w <exp>[,<fill>]</code>	Equivalent to <code>dcb.w <exp>,<fill></code> .
<code>blk.x <exp>[,<fill>]</code>	Equivalent to <code>dcb.x <exp>,<fill></code> .
<code>bss</code>	Equivalent to section <code>bss</code> , <code>bss</code> .
<code>bss_c</code>	Equivalent to section <code>bss_c</code> , <code>bss</code> , <code>chip</code> .
<code>bss_f</code>	Equivalent to section <code>bss_f</code> , <code>bss</code> , <code>fast</code> .
<code>cargs [#<offset>,<symbol1>[.<size1>][,<symbol2>[.<size2>]]...</code>	Defines <code><symbol1></code> with the value of <code><offset></code> . Further symbols on the line, separated by comma, will be assigned the <code><offset></code> plus the size of the previous symbol. The size defaults to 2. Valid optional size extensions are: <code>.b</code> , <code>.w</code> , <code>.l</code> , where <code>.l</code> results in a size of 4, the others 2. The <code><offset></code> argument defaults to the target's address size (4 for M68k) when omitted.
<code>clrfo</code>	Reset stack-frame offset counter to zero. See <code>fo</code> directive.
<code>clrso</code>	Reset structure offset counter to zero. See <code>so</code> directive.
<code>cnop <offset>,<alignment></code>	Insert as many padding bytes as required to reach an address which can be divided by <code><alignment></code> . Then add <code><offset></code> padding bytes. May fill the alignment- and padding-bytes with no-operation instructions for certain cpus. See option <code>-cnop</code> .
<code>code</code>	Equivalent to section <code>code</code> , <code>code</code> .
<code>code_c</code>	Equivalent to section <code>code_c</code> , <code>code</code> , <code>chip</code> .
<code>code_f</code>	Equivalent to section <code>code_f</code> , <code>code</code> , <code>fast</code> .
<code>comm <symbol>,<size></code>	Create a common symbol with the given size. The alignment is always 32 bits.
<code>comment</code>	Starting with the operand field everything is ignored and seen as a comment. There is only one exception, when the operand contains <code>HEAD=</code> . Then the following expression is passed to the TOS output module via the symbol <code>'TOSFLAGS'</code> , to define the Atari specific TOS flags.
<code>cseg</code>	Equivalent to section <code>code</code> , <code>code</code> .
<code>data</code>	Equivalent to section <code>data</code> , <code>data</code> .
<code>data_c</code>	Equivalent to section <code>data_c</code> , <code>data</code> , <code>chip</code> .
<code>data_f</code>	Equivalent to section <code>data_f</code> , <code>data</code> , <code>fast</code> .

- db** <exp1>[,<exp2>,"<string1>",<string2>'...]
 Equivalent to **dc.b** for ArgAsm, BAsm, HX68, Macro68, ProAsm, etc. compatibility. Does not exist in PhxAss- or Devpac-compatibility mode.
- dc.b** <exp1>[,<exp2>,"<string1>",<string2>'...]
 Assign the integer or string constant operands into successive bytes of memory in the current section. Any combination of integer and character string constant operands is permitted.
- dc.d** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 64-bit words of memory in the current section, using the IEEE double precision format when specifying them as floating point constants.
- dc.l** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 32-bit words of memory in the current section.
- dc.p** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 96-bit words of memory in the current section, using the Packed Decimal format when specifying them as floating point constants.
- dc.q** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 64-bit words of memory in the current section.
- dc.s** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 32-bit words of memory in the current section, using the IEEE single precision format when specifying them as floating point constants.
- dc.w** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 16-bit words of memory in the current section.
- dc.x** <exp1>[,<exp2>...]
 Assign the values of the operands into successive 96-bit words of memory in the current section, using the IEEE extended precision format when specifying them as floating point constants.
- dcb.b** <exp>[,<fill>]
 Insert <exp> zero or <fill> bytes into the current section.
- dcb.d** <exp>[,<fill>]
 Insert <exp> zero or <fill> 64-bit words into the current section. <fill> may also be a floating point constant which is then written in IEEE double precision format.
- dcb.l** <exp>[,<fill>]
 Insert <exp> zero or <fill> 32-bit words into the current section.
- dcb.q** <exp>[,<fill>]
 Insert <exp> zero or <fill> 64-bit words into the current section.

`dcb.s <exp>[,<fill>]`

Insert <exp> zero or <fill> 32-bit words into the current section. <fill> may also be a floating point constant which is then written in IEEE single precision format.

`dcb.w <exp>[,<fill>]`

Insert <exp> zero or <fill> 16-bit words into the current section.

`dcb.x <exp>[,<fill>]`

Insert <exp> zero or <fill> 96-bit words into the current section. <fill> may also be a floating point constant which is then written in IEEE extended precision format.

`dl <exp1>[,<exp2>...]`

Equivalent to `dc.l` for ArgAsm, BAsm, HX68, Macro68, ProAsm, etc. compatibility. Does not exist in PhxAss- or Devpac-compatibility mode.

`dr.b <exp1>[,<exp2>...]`

Calculates <expN> - <current pc value> and stores it into successive bytes of memory in the current section.

`dr.w <exp1>[,<exp2>...]`

Calculates <expN> - <current pc value> and stores it into successive 16-bit words of memory in the current section.

`dr.l <exp1>[,<exp2>...]`

Calculates <expN> - <current pc value> and stores it into successive 32-bit words of memory in the current section.

`ds.b <exp>`

Equivalent to `dcb.b <exp>,0`.

`ds.d <exp>`

Equivalent to `dcb.d <exp>,0`.

`ds.l <exp>`

Equivalent to `dcb.l <exp>,0`.

`ds.q <exp>`

Equivalent to `dcb.q <exp>,0`.

`ds.s <exp>`

Equivalent to `dcb.s <exp>,0`.

`ds.w <exp>`

Equivalent to `dcb.w <exp>,0`.

`ds.x <exp>`

Equivalent to `dcb.x <exp>,0`.

`dseg` Equivalent to `section data,data`.

`dw <exp1>[,<exp2>...]`

Equivalent to `dc.w` for ArgAsm, BAsm, HX68, Macro68, ProAsm, etc. compatibility. Does not exist in PhxAss- or Devpac-compatibility mode.

- dx.b <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.b <exp>,0**.
- dx.d <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.d <exp>,0**.
- dx.l <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.l <exp>,0**.
- dx.q <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.q <exp>,0**.
- dx.s <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.s <exp>,0**.
- dx.w <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.w <exp>,0**.
- dx.x <exp>**
Tries to allocate space in the DataBss portion of a code or data section. Otherwise equivalent to **dcb.x <exp>,0**.
- echo <"string"|exp>[,<"string"|exp>]...**
Prints one or more strings or expressions to stdout, terminated by a newline. Strings are identified by single- or double-quotes. In PhxAss-compatibility mode only a single string can be printed.
- inline** End a block of isolated local labels, started by **inline**.
- elif <exp>**
This is a real else-if directive! Not supported by Devpac. It's the same as **else** followed by **if**, but without the need for a matching **endif** directive. Avoids nesting.
- else** Assemble the following lines if the previous **if** condition was false.
- elseif** Same as **else**, for compatibility!
- end** Assembly will terminate with this line. The subsequent source text is ignored.
- endif** Ends a section of conditional assembly.
- endm** Ends a macro definition.
- endr** Ends a repetition block.
- <symbol> equ <expression>**
Define a new program symbol with the name <symbol> and assign to it the value of <expression>. Defining <symbol> twice will cause an error.

<symbol> equ.s <expression>
 Equivalent to **<symbol> frequ.s <expression>**. PhxAss compatibility.

<symbol> equ.d <expression>
 Equivalent to **<symbol> frequ.d <expression>**. PhxAss compatibility.

<symbol> equ.x <expression>
 Equivalent to **<symbol> frequ.x <expression>**. PhxAss compatibility.

<symbol> equ.p <expression>
 Equivalent to **<symbol> frequ.p <expression>**. PhxAss compatibility.

erem Ends an outcommented block. Assembly will continue.

even Aligns to an even address. Equivalent to **cnop 0,2**.

fail <message>
 Show an error message including the **<message>** string. Do not generate an output file.

<symbol> frequ.s <expression>
 Define a new program symbol with the name **<symbol>** and assign to it the floating point value of **<expression>**. Defining **<symbol>** twice will cause an error. The extension is for Devpac-compatibility, but will be ignored.

<symbol> frequ.d <expression>
 Equivalent to **<symbol> frequ.s <expression>**.

<symbol> frequ.x <expression>
 Equivalent to **<symbol> frequ.s <expression>**.

<symbol> frequ.p <expression>
 Equivalent to **<symbol> frequ.s <expression>**.

<label> fo.<size> <expression>
 Assigns the current value of the stack-frame offset counter to **<label>**. Afterwards the counter is decremented by the instruction's **<size>** multiplied by **<expression>**. Any valid M68k size extension is allowed for **<size>**: b, w, l, q, s, d, x, p. The offset counter can also be referenced directly under the name **__F0**.

idnt <name>
 Sets the file or module name in the generated object file to **<name>**, when the selected output module supports it. By default, the input filename passed on the command line is used.

if <expression>
 Conditionally assemble the following lines if **<expression>** is non-zero.

if1 Just for compatibility. Not really supported, as vasm parses a source text only once. Always true.

if2 Just for compatibility. Not really supported, as vasm parses a source text only once. Always false.

ifeq <expression>
 Conditionally assemble the following lines if **<expression>** is zero.

ifne <expression>
 Conditionally assemble the following lines if <expression> is non-zero.

ifgt <expression>
 Conditionally assemble the following lines if <expression> is greater than zero.

ifge <expression>
 Conditionally assemble the following lines if <expression> is greater than zero or equal.

iflt <expression>
 Conditionally assemble the following lines if <expression> is less than zero.

ifle <expression>
 Conditionally assemble the following lines if <expression> is less than zero or equal.

ifb <operand>
 Conditionally assemble the following lines when <operand> is completely blank, except for an optional comment.

ifnb <operand>
 Conditionally assemble the following lines when <operand> is non-blank.

ifc <string1>,<string2>
 Conditionally assemble the following lines if <string1> matches <string2>.

ifnc <string1>,<string2>
 Conditionally assemble the following lines if <string1> does not match <string2>.

ifd <symbol>
 Conditionally assemble the following lines if <symbol> is defined.

ifnd <symbol>
 Conditionally assemble the following lines if <symbol> is undefined.

ifmacrod <macro>
 Conditionally assemble the following line if <macro> is defined.

ifmacrond <macro>
 Conditionally assemble the following line if <macro> is undefined.

ifp1 Just for compatibility. Equivalent to **if1**.

iif <expression> <statement>
 Conditionally assemble the <statement> following <expression>. IIF stands for Immediate IF. If the value of <expression> is non-zero then <statement> is assembled. No ENDC must be used in conjunction with this directive. The <statement> cannot include a label, but a label may precede the IIF directive. For example:

```
foo IIF bar equ 42
```

The **foo** label will be assigned with **42** if **bar** evaluates to true, otherwise **foo** will be assigned with the current program counter. Assigning a value in the IIF <statement> using the equal (=) operator, while the option **-spaces** was

given, cannot work, because the equal operator will be evaluated as part of the expression. I.e. `foo IIF 1+1 = 42` works, but `foo IIF 1 + 1 = 42`, when the option `-spaces` was specified, won't, as `= 42` is evaluated as part of the expression.

incbin <filename>[,<offset>[,<length>]]

Inserts the binary contents of <filename> into the object code at this position. When <offset> is specified, then the given number of bytes will be skipped at the beginning of the file. The optional <length> argument specifies the maximum number of bytes to be read from that file.

incdir <path>

Add another path to search for include files to the list of known paths. Paths defined with `-I` on the command line are searched first.

include <filename>

Include source text of <filename> at this position. When the file name specified has no absolute path, then search it in all defined paths in the order of occurrence, starting with the current work directory.

inline Local labels in the following block are isolated from previous local labels and those after **einline**.

list The following lines will appear in the listing file, if it was requested.

llen <len>

Set the line length in a listing file to a maximum of <len> characters. Currently without any effect.

local Separates two blocks of local labels. Which means, local labels from above this directive may be reused.

macro <name>

Defines a macro which can be referenced by <name>. For compatibility, the <name> may alternatively appear on the left side of the **macro** directive, starting on the first column. Then the operand field is ignored. The macro definition is terminated by an **endm** directive. When calling a macro you may pass up to 9 arguments, separated by comma. These arguments are referenced within the macro context as `\1` to `\9`. Parameter `\0` is set to the macro's first qualifier (mnemonic extension), when given. In Devpac- and PhxAss-compatibility mode, or with option `-allmp`, up to 35 arguments are accepted, where argument 10-35 can be referenced by `\a` to `\z`. In case you have a macro argument which contains commas or spaces you may enclose it between `<` and `>` characters. A `>` character may still be included by writing `>>`, or when embedded within a string. (Note that strings are ignored in Devpac compatibility mode.)

Special macro parameters:

`\@` Insert a unique id, useful for defining labels. Every macro call gets its own unique id.

`\@!` Push the current unique id onto a global id stack, then insert it.

`\@?` Push the current unique id below the top element of the global id stack, then insert it.

<code>\@@</code>	Pull the top element from the global id stack and insert it. The macro's current unique id is not affected by this operation.
<code>\#</code>	Insert the number of arguments that have been passed to this macro. Equivalent to the contents of the symbol <code>NARG</code> .
<code>\?n</code>	Insert the length of the <code>n</code> 'th macro argument.
<code>\.</code>	Insert the argument which is selected by the current value of the <code>CARG</code> symbol (first argument, when <code>CARG</code> is 1).
<code>\+</code>	Works like <code>\.</code> , but increments the value of <code>CARG</code> after that.
<code>\-</code>	Works like <code>\.</code> , but decrements the value of <code>CARG</code> after that.
<code>\<symbolname></code>	Inserts the current decimal value of the absolute symbol <code>symbolname</code> .
<code>\<\$symbolname></code>	Inserts the current hexadecimal value of the absolute symbol <code>symbolname</code> , without leading <code>\$</code> .
<code>mexit</code>	Leave the current macro and continue with assembling the parent context. Note that this directive also resets the level of conditional assembly to a state before the macro was invoked; which means that it also works as a 'break' command on all new <code>if</code> directives.
<code>msource on/off</code>	Enable or disable source level debugging within a macro context. It can be used before one or more macro definitions. When off, the debugger will show the invoking source text line instead. Defaults to <code>on</code> . Also numeric expressions like 0 or 1 are allowed. Note, that this directive currently only has a meaning when using the <code>-linedebug</code> option with the hunk-format output module (<code>-Fhunk</code>).
<code>nolist</code>	This line and the following lines will not be visible in a listing file.
<code>nopage</code>	Never start a new page in the listing file. This implementation will only prevent emitting the formfeed code.
<code>nref <symbol>[,<symbol>...]</code>	Flag <code><symbol></code> as externally defined, similar to <code>xref</code> , but also indicate that references can be optimized to base-relative addressing modes, when possible. This directive is only present in PhxAss-compatibility mode.
<code>odd</code>	Aligns to an odd address. Equivalent to <code>cnop 1,2</code> . Bugs: Note that this is not a real <code>odd</code> directive, as it wastes two bytes when the address is already odd.
<code>offset [<expression>]</code>	Switches to a special offset-section, similar to a <code>section</code> directive, although its contents is not included in the output. Its labels may be referenced as absolute offset symbols. Can be used to define structure offsets. The optional <code><expression></code> gives the start offset for this section. When missing, the last offset of the previous offset-section is used, or 0. <code><expression></code> must evaluate as a constant!

- org** <expression>
Sets the base address for the subsequent code. Note that it is allowed to embed such an absolute ORG block into a section. Return to relocatable mode with any new **section** directive. Although, in Devpac compatibility mode the previous section will stay absolute.
- output** <name>
Sets the output file name to <name> when no output name was given on the command line. A special case for Devpac-compatibility is when <name> starts with a '.' and an output name was already given. Then the current output name gets <name> appended as an extension. When an extension already exists, then it is replaced.
- page**
Start a new page in the listing file (not implemented). Make sure to start a new page when the maximum page length is reached.
- plen** <len>
Set the page length for a listing file to <len> lines. Currently ignored.
- popsection**
Restore the top section from the internal section-stack and activate it. Also refer to **pushsection**.
- printt** <string>[,<string>...]
Prints <string> to stdout. Every additional string into a new line. Quotes are optional.
- printv** <expression>[,<expression>...]
Evaluate <expression> and print it to stdout out in hexadecimal, decimal, ASCII and binary format.
- public** <symbol>[,<symbol>...]
Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.
- pushsection**
Pushes the current section onto an internal stack, where it may be restored from by the **popsection** directive.
- rem**
The assembler will ignore everything from encountering the **rem** directive until an **erem** directive was found.
- rept** <expression>[,<symbol>]
Repeats the assembly of the block between **rept** and **endr** <expression> number of times. <expression> should be positive. Negative values are regarded as 0. The internal symbol REPTN always holds the iteration counter of the inner repeat loop, starting with 0. REPTN is -1 outside of any repeat block. The optional SET-symbol <symbol> receives a copy of the current iteration counter, when given (this is a non-standard extension!).
- rorg** <expression>[,<fill>]
Sets the program counter to an offset relative to the start of the current section, as defined by <expression>. The new program counter (section offset) must

not be smaller than the current one. Any space will be padded by the optional `<fill>` value, or zero.

`<label> rs.<size> <expression>`

Works like the `so` directive, with the only difference that the offset symbol is named `__RS`.

`rseven` Align the structure offset counter (`__RS`) to an even count.

`rsreset` Equivalent to `clrso`, but the symbol manipulated is `__RS`.

`rsset <expression>`

Sets the structure offset counter (`__RS`) to `<expression>`. See `rs` directive.

`section <name>[,<sec_type>][,<mem_type>]`

Starts a new section named `<name>` or reactivates an old one. `<sec_type>` defines the section type and may be `code`, `text` (same as `code`), `data` or `bss`. If the selected output format does not support section names (like "aout", "tos" or "xfile"), then a missing `<sec_type>` argument makes vasm interpret the first argument, `<name>`, as section type instead. Otherwise a missing `<sec_type>` defaults to a `code` section with the given name. The optional `<mem_type>` has currently only a meaning for the hunk-format output module and defines a 32-bit memory attribute which specifies where to load the section. `<mem_type>` is either a numerical constant or one of the keywords `chip` (for Chip-RAM) or `fast` (for Fast-RAM). Optionally it is also possible to attach the suffix `_C`, `_F` or `_P` to the `<sec_type>` argument for defining the memory type.

`<symbol> set <expression>`

Create a new symbol with the name `<symbol>` and assign the value of `<expression>`. If `<symbol>` is already assigned, it will contain a new value from now on.

`setfo <expression>`

Sets the stack-frame offset counter (`__F0`) to `<expression>`. See `fo` directive.

`setso <expression>`

Sets the structure offset counter (`__S0`) to `<expression>`. See `so` directive.

`showoffset [<text>]`

Print current section offset (or absolute address) to the console, preceded by the optional `<text>` (may use quotes). PhxAss compatibility. Do not use in new code.

`<label> so.<size> <expression>`

Assigns the current value of the structure offset counter to `<label>`. Afterwards the counter is incremented by the instruction's `<size>` multiplied by `<expression>`. Any valid M68k size extension is allowed for `<size>`: `b`, `w`, `l`, `q`, `s`, `d`, `x`, `p`. The offset counter can also be referenced directly under the name `__S0`.

`spc <lines>`

Output `<lines>` number of blank lines in the listing file. Currently without any effect.

text Equivalent to **section code,code**.

ttl <name>
 Devpac/PhxAss syntax. Set current page title for the listing file.

<name> ttl
 Motorola syntax. Set current page title for the listing file.

weak <symbol>[,<symbol>...]
 Flag <symbol> as a weak symbol, which means that <symbol> is visible to all modules in the linking process, but may be replaced by any global symbol with the same name. When a weak symbol remains undefined its value defaults to 0.

xdef <symbol>[,<symbol>...]
 Flag <symbol> as a global symbol, which means that <symbol> is visible to all modules in the linking process. See also **public**.

xref <symbol>[,<symbol>...]
 Flag <symbol> as externally defined, which means it has to be imported from another module into the linking process. See also **public**.

4.6 Known Problems

Some known problems of this module at the moment:

- **odd** directive wastes two bytes, when address is already odd.
- Some listing file directives have no effect.
- Macro parameter replacement is also done when a line is commented out. This is a problem for special codes which modify the internal state, like \@@, \@!, \@?, \+, \-.
- **echo**, **printt** and **printv** do not work when the source text doesn't contain any real code.

4.7 Error Messages

This module has the following error messages:

- 1001: mnemonic expected
- 1002: invalid extension
- 1003: no space before operands
- 1004: too many closing parentheses
- 1005: missing closing parentheses
- 1006: missing operand
- 1007: garbage at end of line
- 1008: syntax error
- 1009: invalid data operand
- 1010: , expected
- 1011: identifier expected
- 1012: directive has no effect

- 1013: unexpected "%s" without "%s"
- 1014: illegal section type
- 1015: macro id insert on empty stack
- 1016: illegal memory type
- 1017: macro id stack overflow
- 1018: macro id pull without matching push
- 1019: check comment
- 1020: invalid numeric expansion
- 1021: inline without inline
- 1022: missing %c
- 1023: maximum inline nesting depth exceeded (%d)
- 1024: skipping instruction in struct init
- 1025: last %d bytes of string constant have been cut
- 1026: conditional assembly for pass 1 is not really supported, assuming true
- 1027: conditional assembly for pass 2 is not really supported, assuming false
- 1028: modified memory attributes for section "%s"

5 Madmac Syntax Module

This chapter describes the madmac syntax module, which is compatible to the MadMac assembler syntax, written by Landon Dyer for Atari and improved later to support Jaguar and JRISC. It is mainly intended for Atari's 6502, 68000 and Jaguar systems.

5.1 Legal

This module is written in 2015-2023 by Frank Wille and is covered by the vasm copyright without modifications.

5.2 General Syntax

A statement may contain up to four fields which are identified by order of appearance and terminating characters. The general form is:

```
label:      operator      operand(s)      ; comment
```

Labels must not start at the first column, as they are identified by the mandatory terminating colon (:) character. A double colon (::) automatically makes the label externally visible.

Labels preceded by '.' have local scope and are only valid between two global labels.

Equate directives, starting in the operator field, have a symbol without terminating colon in the first field, left of the operator. The equals-character (=) can be used as an alias for equ. A double-equals (==) automatically makes the symbol externally visible.

```
symbol      equate      expression      ; comment
```

Identifiers, like symbols or labels, may start with any upper- or lower-case character, a dot (.), question-mark (?) or underscore (_). The remaining characters may be any alphanumeric character, a dollar-sign (\$), question-mark (?) or underscore (_).

The operands are separated from the operator by whitespace. Multiple operands are separated by comma (,).

Comments are introduced by the comment character ;. The asterisk (*) can be used at the first column to start a comment. The rest of the line will be ignored.

In expressions, numbers starting with \$ are hexadecimal (e.g. \$fb2c). % introduces binary numbers (e.g. %1100101). Numbers starting with @ are assumed to be octal numbers, e.g. @237. All other numbers starting with a digit are decimal, e.g. 1239.

NOTE: Unlike the original Madmac assembler all expressions are evaluated following the usual mathematical operator priorities.

C-like escape characters are supported in strings.

5.3 Operators

The following operators are changed from vasm's standard:

= Boolean Equality. Replaces standard ==.

5.4 Directives

The following directives are supported by this syntax module (if the CPU- and output-module allow it). Note that all directives, besides the equals-character, may be optionally preceded by a dot (.).

<symbol> = <expression>

Equivalent to **<symbol> equ <expression>**.

<symbol> == <expression>

Equivalent to **<symbol> equ <expression>**, but declare **<symbol>** as externally visible.

abs [<expression>]

Equivalent to **offset** for compatibility with older Madmac versions. Note that **abs** is not available for the jagrisc cpu backend as it conflicts with an instruction name.

assert <expression>[,<expression>...]

Assert that all conditions are true (non-zero), otherwise issue a warning.

bss

The following data (space definitions) are going into the BSS section. The BSS section cannot contain any initialized data.

data

The following data are going into the data section, which usually contains pre-initialized data and no executable code.

dc <exp1>[,<exp2>...]

Equivalent to **dc.w**.

dc.b <exp1>[,<exp2>,<string1>,<string2>...]

Assign the integer or string constant operands into successive bytes of memory in the current section. Any combination of integer and character string constant operands is permitted.

dc.i <exp1>[,<exp2>...]

Assign the values of the operands into successive 32-bit words of memory in the current section. In contrast to **dc.l** the high and low half-words will be swapped as with the Jaguar-RISC **movei** instruction.

dc.l <exp1>[,<exp2>...]

Assign the values of the operands into successive 32-bit words of memory in the current section.

dc.w <exp1>[,<exp2>...]

Assign the values of the operands into successive 16-bit words of memory in the current section.

dcb

Equivalent to **dcb.w**.

dcb.b <exp>[,<fill>]

Insert **<exp>** zero or **<fill>** bytes into the current section.

dcb.l <exp>[,<fill>]

Insert **<exp>** zero or **<fill>** 32-bit words into the current section.

<code>dcb.w <exp>[,<fill>]</code>	Insert <code><exp></code> zero or <code><fill></code> 16-bit words into the current section.
<code>dphrase</code>	Align the program counter to the next integral double phrase boundary (16 bytes).
<code>ds <exp></code>	Equivalent to <code>dcb.w <exp>,0</code> .
<code>ds.b <exp></code>	Equivalent to <code>dcb.b <exp>,0</code> .
<code>ds.l <exp></code>	Equivalent to <code>dcb.l <exp>,0</code> .
<code>ds.w <exp></code>	Equivalent to <code>dcb.w <exp>,0</code> .
<code>else</code>	Else-part of a conditional-assembly block. Refer to 'if'.
<code>end</code>	End the assembly of the current file. Parsing of an include file is terminated here and assembling of the parent source commences. It also works to break the current conditional block, repetition or macro.
<code>endif</code>	Ends a block of conditional assembly.
<code>endm</code>	Ends a macro definition.
<code>endr</code>	Ends a repetition block.
<code><symbol> equ <expression></code>	Define a new program symbol with the name <code><symbol></code> and assign to it the value of <code><expression></code> . Defining <code><symbol></code> twice will cause an error.
<code>even</code>	Align the program counter to an even value, by inserting a zero-byte when it is odd.
<code>exitm</code>	Exit the current macro (proceed to <code>endm</code>) at this point and continue assembling the parent context. Note that this directive also resets the level of conditional assembly to a state before the macro was invoked (which means that it works as a 'break' command on all new <code>if</code> directives).
<code>extern <symbol>[,<symbol>...]</code>	Declare the given symbols as externally defined. Internally there is no difference to <code>globl</code> , as both declare the symbols, no matter if defined or not, as externally visible.
<code>globl <symbol>[,<symbol>...]</code>	Declare the given symbols as externally visible in the object file for the linker. Note that you can have the same effect by using a double-colon (:) on labels or a double-equal (==) on equate-symbols.
<code>if <expression></code>	Start of block of conditional assembly. If <code><expression></code> is true, the block between 'if' and the matching 'endif' or 'else' will be assembled. When false, ignore all lines until an 'else' or 'endif' directive is encountered. It is possible to leave such a block early from within an include file (with <code>end</code>) or

a macro (with `endm`). Note, that `<expression>` must be constant and defined during the first pass, otherwise this is an error.

`ifdef <symbol>`

Conditionally assembles the following lines when the symbol is defined at this point during the first pass. Note, that this is no official Madmac directive and makes your source incompatible.

`ifndef <symbol>`

Conditionally assembles the following lines when the symbol is undefined at this point during the first pass. Note, that this is no official Madmac directive and makes your source incompatible.

`iif <expression>, <statement>`

A single-line conditional assembly. The `<statement>` will be parsed when `<expression>` evaluates to true (non-zero). `<statement>` may be a normal source line, including labels, operators and operands.

`incbin "<file>"`

Inserts the binary contents of `<file>` into the object code at this position.

`include "<file>"`

Include source text of `<file>` at this position.

`list`

The following lines will appear in the listing file, if it was requested.

`long`

Align the program counter to the next integral longword boundary (4 bytes), by inserting as many zero-bytes as needed.

`macro <name> [<argname>[,<argname>...]]`

Defines a macro which can be referenced by `<name>` (case-sensitive). The macro definition is terminated by an `endm` directive and may be exited by `exitm`. When calling a macro you may pass up to 64 arguments, separated by comma. The first ten arguments are referenced within the macro context as `\1` to `\9` and `\0` for the tenth. Optionally you can specify a list of argument names, which are referenced with a leading backslash character (`\`) within the macro. The special code `\~` inserts a unique id, useful for defining labels. `\#` is replaced by the number of arguments. `\!` writes the the size-qualifier (M68k) including the dot. `\?argname` expands to 1 when the named argument is specified and non-empty, otherwise it expands to 0. It is also allowed to enclose argument names in curly braces, which is useful in situations where the argument name is followed by another valid identifier character.

`macundef <name>[,<name>...]`

Undefine one or more already defined macros, making them unknown for the following source to assemble.

`nlist`

This line and the following lines will not be visible in a listing file.

`nolist`

The following lines will not be visible in a listing file.

`offset [<expression>]`

Switches to a special offset-section. The contents of such a section is not included in the output. Their labels may be referenced as absolute offset symbols.

Can be used to define structure offsets. The optional `<expression>` gives the start offset for this section. Defaults to zero when omitted. `<expression>` must evaluate as a constant!

org `<expression>`

Sets the base address for the subsequent code and switch into absolute mode. Such a block is terminated by any section directive or by `.68000`.

phrase Align the program counter to the next integral phrase boundary (8 bytes).

print `<expression>[,<expression>...]`

Prints strings and formatted expressions to the assembler's console. `<expression>` is either a string in quotes or an expression, which is optionally preceded by special format flags:

Several flags can be used to format the output of expressions. The default is a 16-bit signed decimal.

`/x` hexadecimal

`/d` signed decimal

`/u` unsigned decimal

`/w` 16-bit word

`/l` 32-bit longword

For example:

```
.print "Value: ", /d/l xyz
```

qphrase Align the program counter to the next integral quad phrase boundary (32 bytes).

rept `<expression>`

The block between **rept** and **endr** will be repeated `<expression>` times, which has to be positive.

`<symbol>` **set** `<expression>`

Create a new symbol with the name `<symbol>` and assign the value of `<expression>`. If `<symbol>` is already assigned, it will contain a new value from now on.

text The following code and data is going into the text section, which usually is the first program section, containing the executable code.

5.5 Known Problems

Some known problems of this module at the moment:

- Not all Madmac, smac and extended Jaguar-Madmac directives are supported.
- Expressions are not evaluated left-to-right, but mathematically correct.
- Square-brackets (`[]`) are currently not supported to prioritize terms, as an alternative for parentheses.
- Functions (`^^func`) are currently not supported.

5.6 Error Messages

This module has the following error messages:

- 1001: malformed immediate-if
- 1003: cannot export local symbol
- 1004: no space before operands
- 1005: too many closing parentheses
- 1006: missing closing parentheses
- 1007: missing operand
- 1008: garbage at end of line
- 1009: unknown print format flag '%c'
- 1010: invalid data operand
- 1011: print format corrupted
- 1012: identifier expected
- 1014: unexpected "%s" without "%s"

6 Oldstyle Syntax Module

This chapter describes the oldstyle syntax module suitable for some 8-bit CPUs (6502, 65816, 680x, 68HC1x, Z80, etc.), which is available with the extension `oldstyle`.

6.1 Legal

This module is written in 2002-2026 by Frank Wille and is covered by the vasm copyright without modifications.

6.2 Additional options for this module

This syntax module provides the following additional options:

- `-autoexp` Automatically export all non-local symbols, making them visible to other modules during linking.
- `-ast` Allow the asterisk (*) for starting comments in the first column. This disables the possibility to set the code origin with `*=addr` in the first column.
- `-dotdir` Directives have to be preceded by a dot (.).
- `-i` Ignore everything after a blank in the operand field and treat it as a comment. This option is only available when the backend does not separate its operands with blanks as well.
- `-ldots` Allow dots (.) within all identifiers.
- `-noc` Disable C-style constant prefixes.
- `-noi` Disable intel-style constant suffixes.
- `-sect` Enables the additional section directives `text`, `data` and `bss`, which switch to their respective section type. The original `text` directive for creating string-constants and the `data` directive for creating byte-constants are no longer available. But there are still other directives for the same purpose.
- `-spcequ` Allow assignment directives (like `=`, `EQU`, etc.) for symbols which do not start on the first column.

6.3 General Syntax

Labels starting at the first column may be terminated by a colon (:). Either a terminating colon, or an assign directive like `=`, `EQU` or `SET`, is required when the label is preceded by whitespace.

Local labels are introduced by `'.'` or terminated by `'$'`. For the rest, any alphanumeric character, including `'_'`, is allowed. Local labels are valid between two global label definitions.

It is allowed, but not recommended, to refer to any local symbol starting with `'.'` in the source, by prefixing it with the name of the last previously defined global symbol: `global_name.local_name`.

The option `-ldots` allows dots (.) within labels and other identifiers, but disables the above mentioned feature.

Anonymous labels are supported by defining them with a single ':' at the beginning of a line. They may be referenced by ':' followed directly by one or more '+' or '-' signs. A + selects the first anonymous label following the point of reference. A ++ selects the second anonymous label in that direction, and so on. A - selects the first anonymous label before the point of reference. Example:

```
:      jmp      :-      ;infinite loop
```

The opcode field (mnemonics, directives, macros) cannot start at the first column and must be separated by whitespace from the label field, except the label is followed by a : or =.

The operand field is again separated from the opcode field by whitespace. Multiple operands are separated by comma (,), or in some backends by whitespace.

Make sure that you don't define a label on the same line as a directive for conditional assembly (if, else, endif)! This is not supported and leads to undefined behaviour.

Some CPU backends may support multiple statements (directives or mnemonics) per line, separated by a special character (e.g. : for Z80).

Comments are introduced by the comment character (;), or the first blank following the operand field when option -i was given. The rest of the line will be ignored.

Example:

```
mylabel instr op1,op2 ;comment
```

In expressions, numbers starting with \$ are hexadecimal (e.g. \$fb2c). For Z80 also & may be used as a hexadecimal prefix, but make sure to avoid conflicts with the and-operator (either by using parentheses or blanks). % introduces binary numbers (e.g. %1100101). Numbers starting with @ are assumed to be octal numbers, e.g. @237 (except for Z80, where it means binary). A special case is a digit followed by a #, which can be used to define an arbitrary base between 2 and 9 (e.g. 4#3012). Intel-style constant suffixes are supported: h for hexadecimal, d for decimal, o or q for octal and b for binary. Hexadecimal intel-style constants must start with a digit (prepend 0, when required). Also C-style prefixes are supported for hexadecimal (0x) and binary (0b). All other numbers starting with a digit are decimal, e.g. 1239. The one character following a ' or " is converted into ASCII code. A closing quote behind that character is optional in expressions. Not optional for strings.

The current-PC symbol is *, unless redefined by a CPU backend (e.g. Z80 sets \$).

6.4 Operators

The following operators are changed from vasm's standard:

= Boolean Equality. Additional to standard ==.

6.5 Directives

Most data directives, like **byt**, **dfb**, **db**, **word**, **dfw**, **dw**, etc. may optionally be written without any operand. In this case they are treated like space directives, which just increment the program counter by the appropriate number of bytes.

Note, that the size of multi-byte data directives, like **dw**, **word**, **dl**, etc., depends on the CPU backend. Example: **dl** has 24 bits for the 65816.

The following directives are supported by this syntax module (if the CPU- and output-module allow it):

<symbol> = <expression>

Equivalent to **<symbol> equ <expression>**.

abyte <modifier>,<exp1>[,<exp2>,"<string1>"...]

Write the integer or string constants into successive 8-bit memory cells of the current section while modifying each expression (and string-character) by the modifier expression. When the modifier contains the special **._** symbol, then it is a placeholder for any expression from the line. Otherwise the modifier will be just added to each element. Any combination of integer and character string constants is permitted.

addr <exp1>[,<exp2>...]

Assign the values of the operands into successive words of memory in the current section, using the target's endianness and address pointer size. Note that **addr** is not available for 6809. You may use the alternative directive **da** instead.

align <bitcount>

Insert as many zero bytes as required to reach an address where **<bit_count>** low order bits are zero. For example **align 2** would make an alignment to the next 32-bit boundary on a target with 8-bit addressable memory.

asc <exp1>[,<exp2>,"<string1>"...]

Equivalent to **ascii <exp1>[,<exp2>,"<string1>"...]**.

ascii <exp1>[,<exp2>,"<string1>"...]

Assign the integer or string constant operands into successive 8-bit memory cells of the current section. Any combination of integer and character string constant operands is permitted.

asciiz "<string1>"[, "<string2>"...]

Defines one or multiple string constants into successive 8-bit memory cells of the current section, where each string will be automatically terminated by a zero-byte.

assert <expression>[,<message>]

Display an error with the optional **<message>** when the expression is false.

binary <file>

Inserts the binary contents of **<file>** into the object code at this position.

blk <exp>[,<fill>]

Insert **<exp>** zeroed or **<fill>** bytes into the current section.

blk1 <exp>[,<fill>]

Insert **<exp>** zeroed or **<fill>** long words into the current section- The endianness and the number of bits per word depend on the target CPU. For example, will be 24 bits for 6502 and 65816.

blkw <exp>[,<fill>]

Insert **<exp>** zeroed or **<fill>** words into the current section. The endianness and the number of bits per word depend on the target CPU, but the size will usually be the size of two bytes.

bss <exp> Equivalent to **blk** <exp>,0. (Not available with option **-sect**.)

bss With option **-sect**: switches to a bss section with attributes "aurw".

bsz <exp>[,<fill>]
Equivalent to **blk** <exp>[,<fill>].

byt <exp1>[,<exp2>,"<string1>"...]
Assign the integer or string constant operands into successive 8-bit memory cells of the current section. Any combination of integer and character string constant operands is permitted.

byte <exp1>[,<exp2>,"<string1>"...]
Equivalent to **byt** <exp1>[,<exp2>,"<string1>"...].

cond <expression>
Conditionally assemble the following lines if <expression> is non-zero.

di8 <exp1>[,<exp2>...]
Assign the values of the operands into successive 8-bit memory cells of the current section.

di16 <exp1>[,<exp2>...]
Assign the values of the operands into successive 16-bit words of memory in the current section, using the target's endianness.

di24 <exp1>[,<exp2>...]
Assign the values of the operands into successive 24-bit words of memory in the current section, using the target's endianness.

di32 <exp1>[,<exp2>...]
Assign the values of the operands into successive 32-bit words of memory in the current section, using the target's endianness.

di64 <exp1>[,<exp2>...]
Assign the values of the operands into successive 64-bit words of memory in the current section, using the target's endianness.

da <exp1>[,<exp2>...]
Assign the values of the operands into successive words of memory in the current section, using the target's endianness and address pointer size.

data <exp1>[,<exp2>,"<string1>"...]
Equivalent to **byt** <exp1>[,<exp2>,"<string1>"...]. (Not available with option **-sect**.)

data With option **-sect**: switches to a data section with attributes "adrw".

db <exp1>[,<exp2>,"<string1>"...]
Equivalent to **byt** <exp1>[,<exp2>,"<string1>"...].

dc <exp>[,<fill>]
Equivalent to **blk** <exp>[,<fill>].

defb <exp1>[,<exp2>,"<string1>"...]
Equivalent to **byte** <exp1>[,<exp2>,"<string1>"...].

defc <symbol> = <expression>
 Define a new program symbol with the name <symbol> and assign to it the value of <expression>. Defining <symbol> twice will cause an error.

defl <exp1>[,<exp2>...]
 Assign the values of the operands into successive long words of memory in the current section, using the endianness of the target CPU. The number of bits per long word also depends on the CPU. For example, will be 24 bits for 6502 and 65816.

defp <exp1>[,<exp2>...]
 Assign the values of the operands into successive words of memory in the current section, using the target's endianness and address pointer size.

defm "string"
 Equivalent to **fcc** "<string>".

defw <exp1>[,<exp2>...]
 Equivalent to **word** <exp1>[,<exp2>...].

dfb <exp1>[,<exp2>,"<string1>"...]
 Equivalent to **byte** <exp1>[,<exp2>,"<string1>"...].

dfw <exp1>[,<exp2>...]
 Equivalent to **word** <exp1>[,<exp2>...].

defs <exp>[,<fill>]
 Equivalent to **blk** <exp>[,<fill>].

dend Ends an offset-section started by **dsect** and restores the previously active section.

dephase Equivalent to **rend**.

dl <exp1>[,<exp2>...]
 Assign the values of the operands into successive long words of memory in the current section, using the endianness of the target CPU. The number of bits per long word also depends on the CPU. For example, will be 24 bits for 6502 and 65816.

ds <exp>[,<fill>]
 Equivalent to **blk** <exp>[,<fill>].

dsb <exp>[,<fill>]
 Equivalent to **blk** <exp>[,<fill>].

dsect Starts an 'offset-section' (the original directive in ADE was called 'dummy-section') which does not generate any code in the output file. Its only purpose is to define absolute labels. Within a **dsect** block you may use **org** directives to set a new offset, which defaults to zero for the first **dsect** otherwise. Following **dsect** sections continue with the last offset from the former. Such an offset-section block is closed by the **dend** directive, which restores the previous 'real' section.

dsl <exp>[,<fill>]
 Equivalent to **blk1** <exp>[,<fill>].

dsw <exp>[,<fill>]
 Equivalent to **blkw** <exp>[,<fill>].

dw <exp1>[,<exp2>...]
 Equivalent to **word** <exp1>[,<exp2>...].

end Assembly will terminate behind this line.

endc Ends a section of conditional assembly.

endif Ends a section of conditional assembly.

el Equivalent to **else**.

else Assemble the following lines when the previous **if**-condition was false.

ei Equivalent to **endif**. (Not available for Z80 CPU)

inline End a block of isolated local labels, started by **inline**.

endm Ends a macro definition.

endmac Ends a macro definition.

endmacro Ends a macro definition.

endr Ends a repetition block.

endrep Ends a repetition block.

endrepeat
 Ends a repetition block.

endstruct
 Ends a structure definition.

endstructure
 Ends a structure definition.

<symbol> eq <expression>
 Equivalent to **<symbol> equ <expression>**.

<symbol> equ <expression>
 Define a new program symbol with the name <symbol> and assign to it the value of <expression>. Defining <symbol> twice will cause an error.

exitmacro
 Exit the current macro (proceed to **endm**) at this point and continue assembling the parent context. Note, that this directive also resets the level of conditional assembly to a state before the macro was invoked (which means that it works as a 'break' command on all new **if** directives).

extern <symbol>[,<symbol>...]
 See **global**.

even Aligns to an even address. Equivalent to **align 1**.

fail <message>
Show an error message including the <message> string. Do not generate an output file.

fi
Equivalent to **endif**.

fill <exp>
Equivalent to **blk** <exp>,0.

fcbl <exp1>[,<exp2>,"<string1>"...]
Equivalent to **byte** <exp1>[,<exp2>,"<string1>"...].

fcc "<string>"
Puts a single string constant into successive 8-bit memory cells of the current section. The string delimiters may be any printable ASCII character.

fcs "<string>"
Works like **fcc** and **defm**, but additionally sets the most significant bit of the last byte. This can be used as a string terminator on some systems.

fdbl <exp1>[,<exp2>,"<string1>"...]
Assign the values of the operands into successive double-bytes of memory in the current section, using the endianness of the target CPU.

global <symbol>[,<symbol>...]
Flag <symbol> as an external symbol, which means that <symbol> is visible to all modules in the linking process. It may be either defined or undefined.

if <expression>
Conditionally assemble the following lines if <expression> is non-zero.

ifblank <something>
Conditionally assemble the following lines if there are non-blank characters in the operand, which are not a comment.

ifnblank <something>
Conditionally assemble the following lines if there are any non-blank, non-comment characters in the operand.

ifdef <symbol>
Conditionally assemble the following lines if <symbol> is defined.

ifndef <symbol>
Conditionally assemble the following lines if <symbol> is undefined.

ifd <symbol>
Conditionally assemble the following lines if <symbol> is defined.

ifnd <symbol>
Conditionally assemble the following lines if <symbol> is undefined.

ifeq <expression>
Conditionally assemble the following lines if <expression> is zero.

ifne <expression>
Conditionally assemble the following lines if <expression> is non-zero.

- ifgt** <expression>
Conditionally assemble the following lines if <expression> is greater than zero.
- ifge** <expression>
Conditionally assemble the following lines if <expression> is greater than zero or equal.
- iflt** <expression>
Conditionally assemble the following lines if <expression> is less than zero.
- ifle** <expression>
Conditionally assemble the following lines if <expression> is less than zero or equal.
- ifused** <symbol>
Conditionally assemble the following lines if <symbol> has been previously referenced in an expression or in a parameter of an opcode. Issue a warning, when <symbol> is already defined. Note that **ifused** does not work, when the symbol has only been used in the following lines of the source.
- incbin** <file>[,<offset>[,<nbytes>]]
Inserts the binary contents of <file> into the object code at this position. When <offset> is specified, then the given number of 8-bit bytes will be skipped at the beginning of the file. The optional <nbytes> argument specifies the maximum number of 8-bit bytes to be read from that file. When the file size (in 8-bit bytes) is not aligned with the size of a target-byte the missing bits are automatically appended and assumed to be zero. As vasm's internal target-byte endianness for more than 8 bits per byte is big-endian, included binary files are assumed to have the same endianness. Otherwise you have to specify **-ile** to tell vasm that they use little-endian target-bytes (on your 8-bit bytes host file system).
- incdir** <path>
Add another path to search for include files to the list of known paths. Paths defined with **-I** on the command line are searched first.
- include** <file>
Include source text of <file> at this position.
- inline**
Local labels in the following block are isolated from previous local labels and those after **einline**.
- mac** <name>
Equivalent to **macro** <name>. (Not available for unSP CPU)
- list**
The following lines will appear in the listing file, if it was requested.
- local** <symbol>[,<symbol>...]
Flag <symbol> as a local symbol, which means that <symbol> is local for the current file and invisible to other modules in the linking process.
- macro** <name>[,<argname>...]
Defines a macro which can be referenced by <name>. The <name> may also appear at the left side of the **macro** directive, starting on the first column. The macro definition is closed by an **endm** directive. When calling a macro you

may pass up to 9 arguments, separated by comma. These arguments are referenced within the macro context as \1 to \9, or optionally by named arguments, which you have to specify in the operand. Argument \0 is set to the macro's first qualifier (mnemonic extension), when given. The special argument \@ inserts an underscore followed by a six-digit unique id, useful for defining labels. \() may be used as a separator between the name of a macro argument and the subsequent text. \<symbolname> inserts the current decimal value of the absolute symbol `symbolname`.

mdat <file>
Equivalent to `incbin` <file>.

needs <expression>
Equivalent to `symdepend` <expression>.

nolist This line and the following lines will not be visible in a listing file.

org [#]<expression>
Sets the base address for the subsequent code. This is equivalent to `*=<expression>`. An optional # is supported for compatibility reasons.

phase <expression>
Equivalent to `rorg` <expression>.

repeat <expression>[,<symbol>]
Equivalent to `rept` <expression>.

rept <expression>[,<symbol>]
Repeats the assembly of the block between `rept` and `endr` <expression> number of times. <expression> has to be positive. The internal symbol `__RPTCNT` always holds the iteration counter of the inner repeat loop, starting with 0. `__RPTCNT` is -1 outside of any repeat block. The optional SET-symbol <symbol> receives a copy of the current iteration counter, when given.

reserve <exp>
Equivalent to `blk` <exp>,0.

rend Ends a `rorg` block of label relocation. Following labels will be based on `org` again.

rmb <exp>[,<fill>]
Equivalent to `blk` <exp>[,<fill>]. (Not available for 6502 CPU.)

roffs <expression>
Sets the program counter <expression> bytes behind the start of the current section. The new program counter must not be smaller than the current one. The space will be padded with zeros.

rorg <expression>
Relocate all labels between `rorg` and `rend` based on the new origin from <expression>.

ds8 <exp>[,<fill>]
Insert <exp> zeroed or <fill> 8-bit words into the current section.

ds16 <exp>[,<fill>]

Insert <exp> zeroed or <fill> 16-bit words into the current section, using the target's endianness.

ds24 <exp>[,<fill>]

Insert <exp> zeroed or <fill> 24-bit words into the current section, using the target's endianness.

ds32 <exp>[,<fill>]

Insert <exp> zeroed or <fill> 32-bit words into the current section, using the target's endianness.

ds64 <exp>[,<fill>]

Insert <exp> zeroed or <fill> 64-bit words into the current section, using the target's endianness.

section <name>[, "<attributes>"]

Starts a new section named <name> or reactivate an old one. If attributes are given for an already existing section, they must match exactly. The section's name will also be defined as a new symbol, which represents the section's start address. The "<attributes>" string may consist of the following characters:

Section Contents:

c	section has code
d	section has initialized data
u	section has uninitialized data
i	section has directives (info section)
n	section can be discarded
R	remove section at link time
a	section is allocated in memory

Section Protection:

r	section is readable
w	section is writable
x	section is executable
s	section is shareable

Additional Attributes:

f	mark section for far-addressing
z	mark section for near-addressing (e.g. direct/zero-page for 6502/65816)

When attributes are missing they are automatically set for the section names **text**, **data**, **rodata**, **bss**, **.text**, **.data**, **.rodata** and **.bss**. Otherwise they default to **"acrwX"**.

<symbol> set <expression>
 Create a new symbol with the name <symbol> and assign the value of <expression>. If <symbol> was already assigned by **set** before, it will hold the new value from now on.

spc <exp> Equivalent to **blk <exp>,0**.

str "<string1>"[, "<string2>"...]
 Like **asciiz** and **string**, but adds a terminating carriage return (ASCII code 13) instead of a zero-byte.

string "<string1>"[, "<string2>"...]
 Defines one or multiple string constants into successive 8-bit memory cells of the current section, where each string will be automatically terminated by a zero-byte.

struct <name>
 Defines a structure which can be referenced by <name>. Labels within a structure definition can be used as field offsets. They will be defined as local labels of <name> and can be referenced through <name>.<label>. All directives are allowed, but instructions will be ignored when such a structure is used. Data definitions can be used as default values when the structure is used as initializer. The structure name, <name>, is defined as a global symbol with the structure's size. A structure definition is ended by **endstruct**.

structure <name>
 Equivalent to **struct <name>**.

symdepend <expression>
 Declare the current section being dependent on an externally defined symbol from <expression>. In object file formats which support it, this will generate an external symbol reference without any actual relocation being performed (R_NONE in ELF).

text "<string>"
 Equivalent to **fcc "<string>"**. Not available with option **-sect**.

text With option **-sect**: switches to a code section with attributes **"acrx"**.

weak <symbol>[, <symbol>...]
 Flag <symbol> as a weak symbol, which means that <symbol> is visible to all modules in the linking process and may be replaced by any global symbol with the same name. When a weak symbol remains undefined its value defaults to 0.

wor <exp1>[, <exp2>...]
 Equivalent to **word <exp1>[, <exp2>...]**.

wrd <exp1>[, <exp2>...]
 Equivalent to **word <exp1>[, <exp2>...]**.

word <exp1>[, <exp2>...]
 Assign the values of the operands into successive words of memory in the current section, using the endianness of the target CPU. The number of bits per word also depends on the CPU, but will usually have the size of two bytes.

```
xdef <symbol>[,<symbol>...]
    See global.

xlib <symbol>[,<symbol>...]
    See global.

xref <symbol>[,<symbol>...]
    See global.

zmb <exp>[,<fill>]
    Equivalent to blk <exp>[,<fill>].
```

6.6 Structures

The oldstyle syntax is able to manage structures. Structures can be defined in two ways:

```
mylabel struct[ure]
    <fields>
endstruct[ure]
```

or:

```
struct[ure] mylabel
    <fields>
endstruct[ure]
```

Any directive is allowed to define the structure fields. Labels can be used to define offsets into the structure. The initialized data is used as default value, whenever no value is given for a field when the structure is referenced.

Some examples of structure declarations:

```
struct point
x    db 4
y    db 5
z    db 6
endstruct
```

This will create the following labels:

```
point.x ; 0    offsets
point.y ; 1
point.z ; 2
point   ; 3    size of the structure
```

The structure can be used by optionally redefining the field values:

```
point1 point
point2 point 1, 2, 3
point3 point ,,4
```

is equivalent to

```
point1
    db 4
    db 5
    db 6

point2
```

```
                                db 1
                                db 2
                                db 3
point3
                                db 4
                                db 5
                                db 4
```

6.7 Known Problems

Some known problems of this module at the moment:

- Addresses assigned with `org` or to the current pc symbol '*' (on the Z80 the pc symbol is '\$') must be constant.
- Expressions in any form of conditional `if` directive must be constant.

6.8 Error Messages

This module has the following error messages:

- 1001: syntax error
- 1002: invalid extension
- 1003: no space before operands
- 1004: too many closing parentheses
- 1005: missing closing parentheses
- 1006: missing operand
- 1007: garbage at end of line
- 1008: %c expected
- 1009: invalid data operand
- 1010: , expected
- 1011: identifier expected
- 1012: illegal escape sequence %c
- 1013: unexpected "%s" without "%s"
- 1014: dsect already active
- 1015: dend without dsect
- 1016: missing dend
- 1018: maximum inline nesting depth exceeded (%d)
- 1017: inline without inline
- 1021: cannot open binary file "%s"
- 1023: alignment too big
- 1024: label <%s> has already been defined
- 1025: skipping instruction in struct init
- 1026: last %d bytes of string constant have been cut

7 Test output module

This chapter describes the test output module which can be selected with the `-Ftest` option.

7.1 Legal

This module is written in 2002 by Volker Barthelmann and is covered by the vasm copyright without modifications.

7.2 Additional options for this module

This output module provides no additional options.

7.3 General

This output module outputs a textual description of the contents of all sections. It is mainly intended for debugging.

7.4 Restrictions

None.

7.5 Known Problems

Some known problems of this module at the moment:

- None.

7.6 Error Messages

This module has the following error messages:

- None.

8 ELF output module

This chapter describes the ELF output module which can be selected with the `-Felf` option.

8.1 Legal

This module is written in 2002-2016 by Frank Wille and is covered by the vasm copyright without modifications.

8.2 Additional options for this module

`-keepempty`

Do not delete empty sections without any symbol definition.

8.3 General

This output module outputs the **ELF** (Executable and Linkable Format) format, which is a portable object file format and works for a variety of 32- and 64-bit operating systems.

8.4 Restrictions

The ELF output format, as implemented in vasm, currently supports the following architectures:

- ARM
- i386
- Jaguar RISC
- M68k, ColdFire
- PowerPC
- x86_64

The supported relocation types depend on the selected architecture.

8.5 Known Problems

Some known problems of this module at the moment:

- None.

8.6 Error Messages

This module has the following error messages:

- 3002: output module doesn't support cpu <name>
- 3003: write error
- 3005: reloc type <m>, size <n>, mask <mask> (symbol <sym> + <offset>) not supported
- 3006: reloc type <n> not supported
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx

9 a.out output module

This chapter describes the a.out output module which can be selected with the `-Faout` option.

9.1 Legal

This module is written in 2008-2016,2020-2025 by Frank Wille and is covered by the vasm copyright without modifications.

9.2 Additional options for this module

`-aoutalign=<nbytes>`

Align the beginning of the `.data` and `.bss` section in the output to a multiple of `nbytes`. Without this option the default for the current CPU is used.

`-mid=<machine id>`

Sets the MID field of the a.out header to the specified value. The MID defaults to 2 (Sun020 big-endian) for M68k and to 100 (PC386 little-endian) for x86.

9.3 General

This output module emits the `a.out` (assembler output) format, which is an older 32-bit format for Unix-like operating systems, originally invented by AT&T.

9.4 Restrictions

The `a.out` output format, as implemented in vasm, currently supports the following architectures:

- M68k
- i386

The following standard relocations are supported by default:

- absolute, 8, 16, 32 bits
- pc-relative, 8, 16, 32 bits
- base-relative

Standard relocation table entries occupy 8 bytes and don't include an addend, so they are not suitable for most RISC CPUs. The extended relocation format occupies 12 bytes and also allows more relocation types.

9.5 Known Problems

Some known problems of this module at the moment:

- The extended relocation format is not supported.

9.6 Error Messages

This module has the following error messages:

- 3004: section attributes <attr> not supported
- 3008: output module doesn't allow multiple sections of the same type
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx

10 COFF output module

This chapter describes the `coff` output module which can be selected with the `-Fcoff` option.

10.1 Legal

This module is written in 2025 by Jean-Paul Mari and is covered by the `vasm` copyright without modifications.

10.2 Additional options for this module

`-fmagic=<magic id>`

Sets the magic field of the `coff` header to an unsigned 16 bits value. The magic default is based on the `CPU=` define used.

10.3 General

This output module emits the `coff` (assembler output) format, which is a format for executable, object code, and shared library computer files used on Unix like systems. It was introduced in Unix System V, replaced the previously used `a.out` format.

10.4 Restrictions

The `coff` output format, as implemented in `vasm`, currently supports the following architectures:

- `arm`
- `m68k`
- `ppc`
- `x86`

10.5 Known Problems

Some known problems of this module at the moment:

- The debug information is not supported.
- The section's names must fit on 8 characters.
- The BSS section is saved in the object file.

10.6 Error Messages

This module has the following error messages:

- 3002: output module doesn't support `cpu %s`
- 3005: reloc type `<m>`, size `<n>`, mask `<mask>` (symbol `<sym>` + `<offset>`) not supported

11 TOS output module

This chapter describes the TOS output module, which can be selected with option `-Ftos` to generate Atari TOS executable files, or with option `-Fdri` to generate DRI-format object files. Additionally you can generate absolute Z-file executables for the Sharp X68000, when specifying option `-zfile` together with `-Ftos`.

11.1 Legal

This module is written in 2009-2016,2020-2024 by Frank Wille and is covered by the vasm copyright without modifications.

11.2 Additional options for this module

`-szbx` Use the SozobonX extension, which allows symbol names with unlimited length in DRI objects and executables. Overrides the HiSoft extension.

`-stdsymbols`
 Do not write HiSoft extended symbol names. Cut names after 8 characters.

These options are valid for the `tos` module only:

`-monst` Write Devpac "MonST"-compatible symbols.

`-tos-flags=<flags>`
 Sets the flags field in the TOS file header. Defaults to 0. Overwrites a TOS flags definition in the assembler source.

`-zfile=<load-address>`
 Outputs an absolute Z-file for Sharp X68000 computers to be loaded at `<load-address>` in memory.

11.3 General

`-Ftos` The TOS executable file format is used on Atari 16/32-bit computers with 68000 up to 68060 CPU running TOS, MiNT or any compatible operating system. The symbol table is in DRI format and may use HiSoft (default) or SozobonX extended symbol names.

`-Fdri` The object file format defined by Digital Research for Atari M68k systems. May use SozobonX extended symbol names.

11.4 Restrictions

- These file formats only support a single Text (code), Data and BSS section.
- For `tos` all symbols must be defined, otherwise the generation of the executable fails. Unknown symbols are listed by vasm.
- The only relocations allowed in `tos` are 32-bit absolute. For `dri` all 16- and 32-bit absolute and PC-relative relocations are supported. 16-bit base-relative appears as a 16-bit absolute symbol reference.

- The maximum symbol length is 8 characters only. The `tos` format increases the maximum length to 22 by using an extension created by HiSoft, unless forbidden by `-stdsymbols`. With `-szbx` you may enable the SozobonX extension for unlimited length - but you need a linker which supports that format.
- Symbol references in `dri` object files are limited to a maximum of 8192 symbols.

All these restrictions are defined by the file format itself.

11.5 Known Problems

Some known problems of this module at the moment:

- None.

11.6 Error Messages

This module has the following error messages:

- 3004: section attributes `<attr>` not supported
- 3005: reloc type `%d`, size `%d`, mask `0x%lx` (symbol `%s + 0x%lx`) not supported
- 3006: reloc type `%d` not supported
- 3007: undefined symbol `<%s>`
- 3008: output module doesn't allow multiple sections of the same type
- 3009: undefined symbol `<%s>` at `%s+%#lx`, reloc type `%d`
- 3010: section `<%s>`: alignment padding (`%lu`) not a multiple of `%lu` at `0x%llx`
- 3011: weak symbol `<%s>` not supported by output format, treating as global
- 3020: too many symbols for selected output file format
- 3024: section `<%s>`: maximum size of `0x%llx` bytes exceeded (`0x%llx`)

12 GST output module

This chapter describes the `gst` output module which can be selected with the `-Fgst` option.

12.1 Legal

This module is written in 2023 by Frank Wille and is covered by the `vasm` copyright without modifications.

12.2 Additional options for this module

None.

12.3 General

This module outputs the GST object file format by GST Software, which was used by several development tools on the Atari M68k computers. For example by the GST assembler and Devpac.

12.4 Restrictions

- Although there can be multiple sections, they don't have a type.
- Maximum length of section and symbol names is 255 characters.
- Only absolute, pc-relative and common-symbol relocations are supported at the moment.
- Possible relocation sizes are 8, 16 and 32 bits.

12.5 Known Problems

Some known problems of this module at the moment:

- Needs testing.
- Missing support for base-relative relocations.

12.6 Error Messages

This module has the following error messages:

- 3002: output module doesn't support `cpu %s`
- 3011: weak symbol `<%s>` not supported by output format, treating as global

13 Hunk-format output module

This chapter describes the AmigaOS hunk-format output module which can be selected with the `-Fhunk` option to generate objects and with the `-Fhunkexe` option to generate executable files.

13.1 Legal

This module is written in 2002-2025 by Frank Wille and is covered by the vasm copyright without modifications.

13.2 Additional options for this module

`-dbg-globloc`

Adds all local labels to the debug symbol hunk, by trying to construct a unique name out of the previous global label and the original local label name: `global$local`. The default is to write global labels only.

`-dbg-local`

Adds all local labels with their original name to the debug symbol hunk. The default is to write global labels only.

`-hunkpad=<code>`

Sets a two-byte code used for aligning a code hunk to the next 32-bit border. Defaults to 0x4e71 for M68k code sections, to allow linking of functions which extend over two object files. Otherwise it defaults to zero.

`-keepempty`

Do not delete empty sections without any symbol definition.

`-kick1hunks`

Use only those hunk types and external reference types which have been valid at the time of Kickstart 1.x, for compatibility with old assembler sources and old linkers. For example: no longer differentiate between absolute and relative references. In executables it will prevent the assembler from using 16-bit relocation offsets in hunks and rejects 32-bit PC-relative relocations.

`-linedebug`

Automatically generate an SAS/C-compatible LINE DEBUG hunk for the input source. Overrides any line debugging directives from the source text.

`-noabspath`

Do not make absolute paths from source file names for the LINE DEBUG format. Except the path was already specified as absolute on the command line or via a directive.

These options are valid for the `hunkexe` module only:

`-databss` Try to shorten sections in the output file by removing zero words without relocation from the end. This technique is only supported by AmigaOS 2.0 and higher.

13.3 General

This module outputs the **hunk** object (standard for **M68k** and extended for **PowerPC**) and **hunkexe** executable format, which is a proprietary file format used by AmigaOS and WarpOS.

The **hunkexe** module will generate directly executable files, without the need for another linker run. But you have to make sure that there are no undefined symbols, common symbols, or unusual relocations (e.g. small data) left.

It is allowed to define sections with the same name but different attributes. They will be regarded as different entities.

13.4 Restrictions

The **hunk**/**hunkexe** output format is only intended for **M68k** and **PowerPC** cpu modules and will abort when used otherwise.

The **hunk** module supports the following relocation types:

- absolute, 32-bit
- absolute, 16-bit
- absolute, 8-bit
- relative, 8-bit
- relative, 14-bit (mask 0xffff) for PPC branch instructions.
- relative, 16-bit
- relative, 24-bit (mask 0x3ffff) for PPC branch instructions.
- relative, 32-bit
- base-relative, 16-bit
- common symbols are supported as 32-bit absolute and relative references

The **hunkexe** module supports absolute and relative 32-bit relocations only.

13.5 Known Problems

Some known problems of this module at the moment:

- The **hunkexe** module won't process common symbols and allocate them in a BSS section. Use a real linker for that.

13.6 Error Messages

This module has the following error messages:

- 3001: multiple sections not supported by this format
- 3002: output module doesn't support cpu <name>
- 3003: write error
- 3004: section attributes <attr> not supported
- 3005: reloc type <m>, size <n>, mask <mask> (symbol <sym> + <offset>) not supported
- 3006: reloc type <n> not supported

- 3009: undefined symbol <%s> at %s+%#lx, reloc type %d
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
- 3011: weak symbol <%s> not supported by output format, treating as global
- 3014: data definition following a databss space directive
- 3016: absolute file path exceeds maximum size of %d characters
- 3017: converting NONE relocation <%s> to 8-bit ABS with zero addend
- 3018: no additional space in section to convert NONE relocation <%s> to ABS
- 3019: section <%s>: kickstart 1.x cannot initialize bss sections >256k to zero
- 3023: unaligned relocation offset at %s+%#lx
- 3024: section <%s>: maximum size of %#llx bytes exceeded (%#llx)
- 3025: section <%s>: memory flags %#lx have been ignored

14 X68k output module

This chapter describes the Xfile output module which can be selected with the `-Fxfile` option. Refer to the TOS output module for generating absolute Z-file executables.

14.1 Legal

This module is written in 2018,2020,2021,2024 by Frank Wille and is covered by the vasm copyright without modifications.

14.2 Additional options for this module

`-exec=<symbol>`

Use the given label `<symbol>` as entry point of the program. Omitting this option will define the execution address to be the same as the code section's base address.

`-loadhigh`

Set the load mode in the header to `high address (2)`.

14.3 General

This module outputs the Xfile executable file format, which is used on Sharp X68000 16/32-bit computer with 68000 up to 68040 CPU.

14.4 Restrictions

- The source must not define more than one code, data and bss section each. More complex sources with `.rdata` or `.stack` sections require a linker.
- All symbols must be defined, otherwise the generation of the executable fails. Unknown symbols are listed by vasm.
- The only relocations allowed in this format are 32-bit absolute.

14.5 Known Problems

Some known problems of this module at the moment:

- None.

14.6 Error Messages

This module has the following error messages:

- 3004: section attributes `<attr>` not supported
- 3005: reloc type `%d`, size `%d`, mask `0x%lx` (symbol `%s` + `0x%lx`) not supported
- 3006: reloc type `%d` not supported
- 3007: undefined symbol `<%s>`
- 3008: output module doesn't allow multiple sections of the same type
- 3009: undefined symbol `<%s>` at `%s+%#lx`, reloc type `%d`

- 3011: weak symbol `<%s>` not supported by output format, treating as global
- 3024: section `<%s>`: maximum size of `0x%llx` bytes exceeded (`0x%llx`)

15 O65 output module

This chapter describes the O65 binary relocation format V1.3 for the 6502 family, as defined by Andre Fachat on 6502.org. Option `-Fo65` outputs object files suitable for another linker pass, while `-Fo65exe` outputs executable files for an O65 loader. The difference is just a flag which declares the file being an object, and vasm will make sure that the load-addresses of sections in an executable will be consecutive and do not overlay.

15.1 Legal

This module is written in 2021 by Frank Wille and is covered by the vasm copyright without modifications.

15.2 Additional options for this module

- `-bss=<addr>`
Sets a start address for the **bss** section.
- `-data=<addr>`
Sets a start address for the **data** section.
- `-fopts` Enable informational header options, generated by the assembler: file name, assembler name and version, creation date.
- `-foauthor=<name>`
Write author's name to the header options.
- `-foname=<name>`
Write file name to the header options. Overwrites the real file name, which would be set by `-fopts`.
- `-paged` Make the output file use paged alignment and simplified paged relocations.
- `-secalign=<align>`
Set minimum alignment for all sections as number of least significant bits which have to be zero. **align** may be 0, 1, 2, 8. The default behaviour is to use the maximum alignment given by the input sections.
- `-stack=<stacksize>`
Store required stack size in the header.
- `-text=<addr>`
Sets a start address for the **text** section.
- `-zero=<addr>`
Sets a start address for the **zero** (zero/direct page) section.

These options are valid for the `o65exe` module only:

- `-bsszero` Set a flag in the header which requests automatic clearing of the **bss** section.

15.3 General

This output module outputs the `o65` object file and `o65exe` executable file format for 6502-family processors and the 65816. The processor type is determined by the selected CPU of the active backend and stored in the header.

The `o65exe` module generates executable files for a `o65`-loader, which is present in some 6502 operating systems (e.g. Linux, SMOS, OS/A65). Unresolved symbols are allowed in `o65` object- and executable-files. In the latter case the `o65`-loader is responsible to resolve them. Common symbols, weak symbol and most relocation types, except absolute addresses, are not supported by `o65`.

The `o65` format recognizes four different sections by their attributes or name:

- text, sections which have executable code and/or are not writable (`acrx`).
- data, sections which have initialized data and may be read and written (`adrw`).
- bss, sections which have uninitialized data and may be read and written (`aurw`).
- zero, sections which have uninitialized data, may be read and written, and do not have the string "`bss`" anywhere in their name (`aurwz`).

Up to two absolute sections (`ORG` directive) can be stored in the `text` and `data` slots, in the order of occurrence.

15.4 Restrictions

Currently the `o65/o65exe` output module is only intended to work with the 6502 cpu module and will abort when used otherwise.

It supports all relocation types defined by `o65`, which are:

- 0x20: absolute, 8-bit value or low-byte of 16-bit address
- 0x40: absolute, 8-bit high-byte of 16-bit address
- 0x80: absolute, 16-bit address
- 0xa0: absolute, 8-bit segment-byte of 24-bit address
- 0xc0: absolute, 24-bit address

Common or weak symbols are not supported.

15.5 Known Problems

Some known problems of this module at the moment:

- Needs a better way to set start addresses for the sections in the assembler.

15.6 Error Messages

This module has the following error messages:

- 3004: section attributes `<attr>` not supported
- 3008: output module doesn't allow multiple sections of the same type (`%s`)
- 3011: weak symbol `<%s>` not supported by output format, treating as global
- 3013: reloc type `%d`, mask `%#lx` to symbol `%s + %#lx` does not fit into `%u` bits
- 3015: file option `%d` max size exceeded: `%lu`

16 AOF output module

This chapter describes the ARM (or Acorn) Object Format output module which can be selected with the `-Faof` option.

16.1 Legal

This module is written in 2025 by Frank Wille and is covered by the vasm copyright without modifications.

16.2 General

This module outputs the AOF object file format in version 310, as described by the ARM DUI0041C Copyright 1997, 1998 by ARM Limited. It is used by some older ARM architectures, like Acorn RISC computers or the 3DO. It supports 26- and 32-bit ARM architectures up to AA4, including Thumb mode.

16.3 Restrictions

- 26 or 32 bits.
- Always generates Type-2 relocations.

16.4 Known Problems

Some known problems of this module at the moment:

- Some code area attributes are never set, because vasm is lacking the information: reentrant, software stack checking, ARM/Thumb interworking.
- No debugging table support.
- No entry location.
- No strong symbols.

16.5 Error Messages

This module has the following error messages:

- 3005: reloc type %d, size %d, mask 0x%lx (symbol %s + 0x%lx) not supported
- 3024: section <%s>: maximum size of 0x%llx bytes exceeded (0x%llx)

17 vobj output module

This chapter describes the simple binary output module which can be selected with the `-Fvobj` option.

17.1 Legal

This module is written in 2002-2025 by Volker Barthelmann and is covered by the vasm copyright without modifications.

17.2 Additional options for this module

- `-vobj2` Use vobj format version 2 which generally reduces file size.
- `-vobj3` Version 3 supports absolute section addresses and indirect symbols.

17.3 General

This output module outputs the `vobj` object format, a simple portable proprietary object file format of `vasm`.

The Format is defined as follows:

```
Header
    .byte 0x56,0x4f,0x42,0x4a
    .byte flags
    Bits 0-1:
        1: BIGENDIAN
        2: LITTLEENDIAN
    Bits 2-7:
        VOBJ-Version (0-based)
    .number bitsperbyte
    .number bytespertaddr
    .string cpu
    .number nsections [1-based]
    .number nsymbols [1-based]

nsymbols
    .string name
    .number type
    .number flags
    .number secindex
    .number val
    .number size

nsections
    .string name
    .string attr
    .number flags
```

```

    .number address (entry is present with version 3+ and flags&ABSOLUTE only)
    .number align
    .number size
    .number nrelocs
    .number databytes
    .byte[databytes]

nrelocs [standard|special]
standard
    .number type
    .number byteoffset
    .number bitoffset
    .number size
    .number mask
    .number addend
    .number symbolindex | 0 (sectionbase)

special
    .number type
    .number size
    .byte[size]

.number:[taddr]
    .byte 0--127 [0--127]
    .byte 128-191 [x-0x80 bytes little-endian], fill remaining with 0
    .byte 192-255 [x-0xC0 bytes little-endian], fill remaining with 0xff
    (.byte 192-255 is supported by vobj version 2+ only.)

```

17.4 Restrictions

None.

17.5 Known Problems

Some known problems of this module at the moment:

- None.

17.6 Error Messages

This module has the following error messages:

- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
- 3026: output module requires option -vobj3 to support indirect symbols

18 Simple binary output module

This chapter describes the simple binary output module which can be selected with the `-Fbin` option.

18.1 Legal

This module is written in 2002-2025 by Volker Barthelmann and Frank Wille and is covered by the vasm copyright without modifications.

18.2 Additional options for this module

`-apple-bin`

Prepends a 4-byte header with load-address and length to the raw binary file, as used by the `AppleCommander` to write such files onto a DOS 3.3 disk image.

`-atari-com`

Writes an Atari DOS COM header preceding the output file. It has a standard header (0xFFFF), which is followed by any number of sections. Each section starts with two little-endian words defining the address of the first and last byte in memory.

`-coalesced`

Do not pad the space between separate org-blocks, but output all of them in a coalesced manner (sorted by address).

`-cbm-prg`

Writes a Commodore PRG header preceding the output file, which consists of two bytes in little-endian order, defining the load address of the program.

`-coco-ml`

Writes a Tandy Color Computer machine language file, which has a header with load address and length for each section and is terminated by a trailer with the execution address.

`-dragon-bin`

Writes a Dragon DOS header preceding the output file, where the file type is set to \$02 for binary. The load address is taken from the first section's start address. This will also be the execute-address, when not specified otherwise. Refer to option `-exec`.

`-exec=<symbol>`

Use the given symbol `<symbol>` as entry point of the program, for those output format headers which support it. Otherwise this option will be silently ignored. Omitting this option will usually define the execution address to be the same as the load address.

`-foenix-pgx`

Writes a simple, single-segment format for the 65C02- and 65816-based Foenix computers. The header defines the program's load address, which is also the start address.

`-foenix-pgz`

Write a multi-segment format for Foenix computers. The format is derived from binary format used by Western Design Center's C compiler. Every segment is

stored with load address and size, and there is also a start address defined. There is a 24-bit ('Z') and a 32-bit ('z') format which will be selected according to the target CPU.

-join[=<address>]

To be able to write multiple sections as a raw binary file this option invokes a mini-linker, which joins all sections with ascending addresses, regarding their alignment. The start address defaults to 0 when missing. Only simple PC-relative and absolute relocations are supported at the moment.

-oric-mc Writes a machine code file header for Oric-1, Oric-Atmos and compatible systems. It includes the file type and name, as well as the first and last address of the program to load. Note, that the name defaults to the output file name, limited to 15 characters. A ".tap" extension will be removed automatically.

-oric-mcx

Same as **-oric-mc**, but sets the auto-execute flag in the header.

-start=<address>

Set the start address for the default section, when no **section** or **org** directive was given.

18.3 General

This output module outputs the contents of all sections as simple binary data, by default without any header or additional information. When there are multiple sections, they must not overlap. Gaps between sections or org-blocks are filled with zero bytes, when not using a special header format, like Atari COM. The padding can be avoided by option **-coalesced**. Undefined symbols are not allowed.

18.4 Known Problems

Some known problems of this module at the moment:

- None.

18.5 Error Messages

This module has the following error messages:

- 3001: sections <%s>:%llx-%llx and <%s>:%llx-%llx must not overlap
- 3007: undefined symbol <%s>
- 3010: section <%s>: alignment padding (%lu) not a multiple of %lu at 0x%llx
- 3013: reloc type %d, mask 0x%lx to symbol %s + 0x%lx does not fit into %u bits
- 3021: all sections are absolute, nothing to relocate

19 Motorola S-Record output module

This chapter describes the Motorola srecord output module which can be selected with the `-Fsrec` option.

19.1 Legal

This module is written in 2015 by Joseph Zatarski and is covered by the vasm copyright without modifications.

19.2 Additional options for this module

- `-crlf` Enforce Carriage-Return and Line-Feed ("`\r\n`") line endings. Default is to use the host's line endings.
- `-exec[=<symbol>]`
 Use the given symbol `<symbol>` as entry point of the program. This start address will be written into the trailer record, which is otherwise zero. When the symbol assignment is omitted, then the default symbol `start` will be used.
- `-s19` Writes S1 data records and S9 trailers with 16-bit addresses.
- `-s28` Writes S2 data records and S8 trailers with 24-bit addresses.
- `-s37` Writes S3 data records and S7 trailers with 32-bit addresses. This is the default setting.

19.3 General

This output module outputs the contents of all sections in Motorola srecord format, which is a simple ASCII output of hexadecimal digits. Each record starts with 'S' and a one-digit ID. It is followed by the data and terminated by a checksum and a newline character. Every section starts with a new header record.

19.4 Known Problems

Some known problems of this module at the moment:

- A new header is written for every new section. This may cause compatibility issues.

19.5 Error Messages

This module has the following error messages:

- 3007: undefined symbol `<%s>`
- 3010: section `<%s>`: alignment padding (`%lu`) not a multiple of `%lu` at `0x%llx`
- 3012: address `0x%llx` out of range for selected format

20 Intel Hex output module

This chapter describes the Intel Hex output module which can be selected with the `-Fihex` option.

20.1 Legal

This module is written in 2020 by Rida Dzhaafar and is covered by the vasm copyright without modifications.

20.2 Additional options for this module

- `-crlf` Enforce Carriage-Return and Line-Feed ("`\r\n`") line endings. Default is to use the host's line endings.
- `-i8hex` Selects a format supporting 16-bit address space (default).
- `-i16hex` Selects a format supporting 20-bit address space.
- `-i32hex` Selects a format supporting 32-bit address space.
- `-record-size=<n>`
 Sets the number of bytes per record to `n`. Defaults to 32 bytes.

20.3 General

This output module outputs the contents of all sections in Intel hex format, which is a simple ASCII output of hexadecimal digits.

20.4 Known Problems

Some known problems of this module at the moment:

- None?

20.5 Error Messages

This module has the following error messages:

- 3001: sections `<%s>:%llx-%llx` and `<%s>:%llx-%llx` must not overlap
- 3002: output module doesn't support cpu `%s`
- 3007: undefined symbol `<%s>`
- 3012: address `0x%llx` out of range for selected format

21 C #define output module

This chapter describes the C #define output module which can be selected with the `-Fcdef` option.

21.1 Legal

This module is written in 2020 by Volker Barthelmann and is covered by the vasm copyright without modifications.

21.2 Additional options for this module

There are currently no additional options for this output module.

21.3 General

This output module outputs the values of global absolute symbols as a series of `#define` directives that can be included in a C compiler. No code is generated.

21.4 Known Problems

Some known problems of this module at the moment:

- None.

21.5 Error Messages

This module has no error messages:

22 Project Hans custom output module

This chapter describes the custom output module for the project "Hans" which can be selected with the `-Fhans` option.

22.1 Legal

This module is written in 2024 by Yannik Stamm and is covered by the vasm copyright without modifications.

22.2 General

This output module outputs the contents of all sections in a custom verbose object format. It is mainly thought to be used together with a custom linker.

22.3 Known Problems

Some known problems of this module at the moment:

- None?

22.4 Error Messages

This module has the following error messages:

- 3022: expression type of symbol %s not supported

23 Wozmon output module

This chapter describes the wozmon output module which can be selected with the `-Fwoz` option.

23.1 Legal

This module is written in 2023 by anomie-p and is covered by the vasm copyright without modifications.

23.2 Contact

The author of this module may be contacted for bug reports:

- anomie-p (anomie-p@protonmail.com)

23.3 Additional options for this module

There are no additional options for this module.

23.4 General

This output module outputs the contents of all sections as wozmon monitor commands, which is a simple ASCII output of hexadecimal digits.

The output is suitable for an ascii transfer via serial connection to a system running wozmon. Character and/or line delays are likely to be necessary for a successful transfer.

The wozmon command parser converts up to sixteen bit hexadecimal values. An error containing the maximum out of range address is reported if a sixteen bit address space is exceeded.

23.5 Known Problems

Some known problems of this module at the moment:

- None.

23.6 Error Messages

This module has the following error messages:

- 3001: sections `<%s>:%llx-%llx` and `<%s>:%llx-%llx` must not overlap
- 3002: output module doesn't support cpu `%s`
- 3007: undefined symbol `<%s>`
- 3012: address `0x%llx` out of range for selected format

24 MOS paper tape output module

This chapter describes the MOS paper tape output module which can be selected with the `-Fpap` option.

24.1 Legal

This module was written in 2024 by Dimitri Theulings and is covered by the vasm copyright without modifications.

24.2 Additional options for this module

- `-strict` Enforce strict compliance with MOS paper tape format. In particular, in strict mode six NULL characters (ASCII 0x00) are appended after the carriage return (ASCII 0x0D) and line feed (ASCII 0x0A) for each record. Additionally the file is terminated with XOFF (ASCII 0x13).
- `-record-size=<number>`
 Set the number of data bytes per record. Defaults to 24.
- `-start=<address>`
 Set the start address for the default section, when no `section` or `org` directive was given.

24.3 General

This module outputs the contents of all sections in MOS Paper Tape format, which is a simple ASCII output of hexadecimal digits. Each record starts with `;` followed by a single byte indicating number of data bytes contained in the record (by default 24). The record's starting address high (1 byte, 2 characters), starting address low (1 byte, 2 characters), and data (n bytes, $2n$ characters) follow. Each record is terminated by the record's check-sum (2 bytes, 4 characters), a carriage return (ASCII 0x0D), line feed (ASCII 0x0A). The final record does not contain data bytes and instead lists the number of records, followed by a checksum.

In `strict` mode, each record is followed by six NULL characters (ASCII 0x00) and the file is terminated with XOFF (ASCII 0x13).

24.4 Known Problems

There are currently no known problems.

24.5 Error Messages

This module may return the following error messages:

- 3001: sections `<%s>:%llx-%llx` and `<%s>:%llx-%llx` must not overlap
- 3002: output module doesn't support cpu `%s`
- 3007: undefined symbol `<%s>`
- 3012: address `0x%llx` out of range for selected format

25 M68k cpu module

This chapter documents the backend for the Motorola M68k/CPU32/ColdFire microprocessor family.

25.1 Legal

This module is written in 2002-2026 by Frank Wille and is covered by the vasm copyright without modifications.

25.2 Additional options for this module

Note, that the order on the command line may be important when specifying options. For example, if you specify `-devpac` compatibility mode behind enabling some optimization options, the Devpac-mode might disable these optimizations again.

This module provides the following additional options:

25.2.1 CPU selections

- `-m68000` Generate code for the MC68000 CPU (default).
- `-m68008` Generate code for the MC68008 CPU.
- `-m68010` Generate code for the MC68010 CPU.
- `-m68020` Generate code for the MC68020 CPU.
- `-m68030` Generate code for the MC68030 CPU.
- `-m68040` Generate code for the MC68040 CPU.
- `-m68060` Generate code for the MC68060 CPU.
- `-m68020up` Generate code for the MC68020-68060 CPU. Be careful with instructions like PFLUSHA, which exist on 68030 and 68040/060 with a different opcode (vasm will use the 040/060 version).
- `-m68080` Generate code for the Apollo Core AC68080 CPU. Note, that using register banking (using AMMX registers instead of `d0 - d7` or `a0 - a7`) in general purpose instructions might not yet work correctly with current Apollo Cores (at least up to 2024). This will change with future Core releases. Register banking for FPU instructions should be final already.
- `-mcpu32` Generate code for the CPU32 family (MC6833x, MC6834x, etc.).
- `-mcf5...`
- `-m5...` Generate code for a ColdFire family CPU. The following types are recognized: 5202, 5204, 5206, 520x, 5206e, 5207, 5208, 5210a, 5211a, 5212, 5213, 5214, 5216, 5224, 5225, 5232, 5233, 5234, 5235, 523x, 5249, 5250, 5253, 5270, 5271, 5272, 5274, 5275, 5280, 5281, 528x, 52221, 52553, 52230, 52231, 52232, 52233, 52234, 52235, 52252, 52254, 52255, 52256, 52258, 52259, 52274, 52277, 5307, 5327, 5328, 5329, 532x, 5372, 5373, 537x, 53011, 53012, 53013, 53014, 53015, 53016, 53017, 5301x, 5407, 5470, 5471, 5472, 5473, 5474, 5475, 547x, 5480, 5481, 5482, 5483, 5484, 5485, 548x, 54450, 54451, 54452, 54453, 5445x.

- mcfv2** Generate code for the V2 ColdFire core. This option selects ISA_A (no hardware division or MAC), which is the most limited ISA supported by 5202, 5204 and 5206. All other ColdFire chips are backwards compatible to V2.
- mcfv3** Generate code for the V3 ColdFire core. This option selects ISA_A+, hardware division MAC and EMAC instructions, which are supported by nearly all V3 CPUs, except the 5307.
- mcfv4** Generate code for the V4 ColdFire core. This option selects ISA_B and MAC as supported by the 5407.
- mcfv4e** Generate code for the V4e ColdFire core. This option selects ISA_B, USP-, FPU-, MAC- and EMAC-instructions (no hardware division) as supported by all 547x and 548x CPUs.
- m68851** Generate code for the MC68851 MMU. May be used in combination with another **-m** option.
- m68881** Generate code for the MC68881 FPU. May be used in combination with another **-m** option.
- m68882** Generate code for the MC68882 FPU. May be used in combination with another **-m** option.
- no-fpu** Ignore any FPU options or directives, which has the effect that no 68881/2 FPU instructions will be accepted. This option can override the default behaviour of **-gas** enabling the FPU.

25.2.2 Optimization options

- no-opt** Disable all optimizations. Can be seen as a main switch to ignore all other optimization options on the command line and in the source.
- opt-allbra** When specified the assembler will also try to optimize branch instructions which already have a valid size extension. This option is automatically enabled in **-phxass** mode.
- opt-brajump** Translate relative branch instructions, whose destination is in a different section, into absolute jump instructions.
- opt-clr** Enables optimization from **MOVE #0,<ea>** into **CLR <ea>** for the MC68000. Note that **CLR** will execute a read-modify-write cycle on the 68000, so it is disabled by default. With 68010 and higher this is a generic standard optimization.
- opt-div** Unsigned immediate divisors, which are a power of two (from 2 to 256), are optimized to shifts. Divisions by 1 are replaced by **TST.L Dn** (32-bit) or **MVZ.W Dn,Dn** (16-bit, ColdFire only). Divisions by -1 are replaced by **NEG.L Dn** (32-bit) or by a combination of **NEG.W Dn** and **MVZ.W Dn,Dn** (16-bit, ColdFire only). This optimization will leave the flags in a different state as can normally be expected after a division instruction.

-opt-fconst

Floating point constants are loaded with the lowest precision possible. This means that `FMOVE.D #1.0,FP0` would be optimized to `FMOVE.S #1.0,FP0`, or even `FMOVE.W #1,FP0`, because it is faster and shorter at the same precision. The optimization will be performed on all FPU instructions with immediate addressing mode. When an FDIV-family instruction (`FSDIV`, `FDDIV`, `FSGLDIV`) is detected it will additionally be checked if the immediate constant is a power of 2 and then converted into `FMUL #1/c,FPn`.

-opt-jbra

`JMP` and `JSR` instructions to external labels will be converted into `BRA.L` and `BSR.L`, when the selected CPU is 68020 or higher (or CPU32).

-opt-lsl Allows optimization of `LSL #1` into `ADD`. It is also needed to optimize `ASL #2` and `LSL #2` into two `ADD` instructions (together with **-opt-speed**). These optimizations can modify the V-flag, which may not be intended.

-opt-movem

Enables optimization from `MOVEM <ea>,Rn` into `MOVE <ea>,Rn` (or the other way around). May also optimize `MOVEM` with two registers into two separate `MOVE` instructions, when advantageous for the currently selected CPU. This optimization will modify the flags when the destination is no address register.

-opt-mul Immediate multiplication factors, which are a power of two (from 2 to 256), are optimized to shifts. Multiplications with zero are replaced by a `MOVEQ #0,Dn`, with -1 are replaced by a `NEG.L Dn` and with 1 by `EXT.L Dn` or `TST.L Dn` (long-form). Not all optimizations are available for all cpu types (e.g. `MULU.W` can only be optimized on ColdFire by using the `MVZ.W` instruction). This optimization will leave the flags in a different state as can normally be expected after a multiplication instruction, and the size of the optimized code may be bigger than before in some situations (e.g. `MULS.W #4,Dn`). So the latter will additionally require the **-opt-speed** flag.

-opt-nmoveq

Optimizes `MOVE.L #x,Dn` into a combination of `MOVEQ` and `NEG.W`, which works for ranges from `$ff81<=x<=$ffff` and `$ffff0001<=x<=$ffff0080`. Note that this optimization flips the N-flag!

-opt-pea Enables optimization from `MOVE #x,-(SP)` into `PEA x`. This optimization will leave the flags unmodified, which might not be intended.

-opt-size

Optimize for size, even if this would make the code slower. This enables for example optimization of `MOVE.L #x,Dn` into `MOVEQ #x>>n,Dn + LSL.W #n,Dn`. It is mostly used together with other optimization flags.

-opt-speed

Optimize for speed, even if this would increase code size. For example it enables optimization of `ASL.W #2,Dn` into two `ADD.W Dn,Dn` instructions. Or `MULS.W #-4,Dn` into `EXT.L Dn + ASL.L #2,Dn + NEG.L Dn`.

-opt-st Enables optimization from `MOVE.B #-1,<ea>` into `ST <ea>`. This optimization will leave the flags unmodified, which might not be intended.

- opt-<option>**
 Alternatively, you can use **-opt-** followed by a Devpac-compatible option, as described under the **OPT** directive. Example: **-opt-o1-** disables branch optimization in the same way as an **OPT o1-** directive would do at the top of the source text.
- sc** Small code model. All **JMP** and **JSR** instructions to external labels will be converted into 16-bit PC-relative jumps.
- sd** References to labels in a small data section (named "**__MERGED**") are optimized into base-relative addressing mode, using the current base register set by an active **NEAR** directive. This option is automatically enabled in **-phxass** mode.
- showcrit** Print all critical optimizations which have side effects. Among those are **-opt-lsl**, **-opt-mul**, **-opt-st**, **-opt-pea**, **-opt-movem** and **-opt-clr**.
- showopt** Print all optimizations and translations vasm is doing (same as **opt ow+**).
- warnabs16** Show a warning for every access to an absolute 16-bit address.
- warnabs32** Show a warning for every access to an absolute 32-bit address. This doesn't include section labels.
- warnunaligned** Show a warning, if the target address for absolute (16/32-bit) or 16-bit PC-relative addressing modes is odd and the operation size is not 8-bit. No warnings for accessing external symbols. The warning may not make sense for all instructions (e.g. **LEA**).

In its default setting (no **-devpac** or **-phxass** option) vasm performs the following optimizations:

- Absolute to PC-relative.
- Branches without explicit size.
- Displacements (32 to 16 bit, **(0,An)** to **(An)**, etc).
- Optimize floating point constants to the lowest possible precision.
- Many instruction optimizations which are safe.

25.2.3 Other options

- conv-brackets**
 Brackets ('[' and ']') in an operand are automatically treated like parentheses ('(' and ')') as long as the CPU is 68000 or 68010. This is a compatibility option for some old assemblers.
- devpac** All options are initially set to be Devpac compatible. Which means that all optimizations are disabled, no debugging symbols will be written and vasm will warn about any optimization being done. When symbol output is enabled by

`opt d+`, then the TOS symbol table defaults to standard DRI format (limited to 8 characters). Shift-right operations are performed using an unsigned 32-bit value. Other options are the same as `vasm`'s defaults. The symbol `__G2` is defined, which contains information about the selected cpu type. The symbol `__LK` reflects the type of output file to generate. Which is 0 for TOS executables, 1 for DRI objects, 2 for GST objects, 3 for AmigaDOS objects and 4 for AmigaDOS executables. All other formats are represented by 99, as they are unknown to Devpac. It will also automatically enable `-guess-ext` and `-nodpc`.

- `-elfregs` Register names have to be prefixed by a '%' to prevent confusion with symbol names.
- `-extsd` Recognize small data references in all 020+ extended addressing modes using 16-bit displacements on the base register. By default only the 16-bit address register displacement addressing mode can be used with small data (for compatibility reasons).
- `-gas` Enable additional GNU-as compatibility mnemonics, like `mov`, `movm` and `jra`. Also accepts `|` instead of `;` for comments. GNU-as compatibility mode selects the 68020 CPU and 68881/2 FPU by default and enables `-opt-jbra`.
- `-guess-ext` Accept illegal size extensions for an instruction, as long as the instruction is unsized or there is just a single size possible. This is the default setting in PhxAss and Devpac compatibility mode.
- `-kick1hunks` Prevents optimization of `JMP/JSR` to 32-bit PC-relative (`BRA/BSR`), which requires a relocation type that is missing for Kickstart 1.x.
- `-nodpc` Do not attempt to encode absolute PC-displacements directly. Example: `10(PC)`
- `-no-typechk` Do not check the size and type of expressions (`OPT t-`). Example: `dc.b 300`
- `-phxass` PhxAss-compatibilty mode. The "current PC symbol" (e.g. `*` in `mot-syntax` module) is set to the instruction's address + 2 whenever an instruction is parsed. According to the current cpu setting the symbols `__CPU`, `__FPU` and `__MMU` are defined. `JMP/JSR (label,PC)` will never be optimized (into a branch, for example). It will also automatically enable `-opt-allbra`, `-sd` and `-guess-ext`.
- `-rangewarnings` Values which are out of range usually produce an error. With this option the errors 2030, 2033, 2037 and 2040 will be displayed as a warning, allowing the user to create an object file.
- `-regsymredef` Allow redefining register symbols with `EQR`. This should only be used for compatibility with old sources. Not many assemblers support that.
- `-sdreg=<n>` Set the small data base register to `An`. `<n>` is valid between 0 and 6.

-sgs Additionally allow immediate operands to be prefixed by `&` instead of just by `#`. This syntax was used by the SGS assembler.

25.3 General

This backend accepts M68k and CPU32 instructions as described in Motorola's M68000 family Programmer's Reference Manual. Additionally it supports ColdFire instructions as described in Motorola's ColdFire Microprocessor Family Programmer's Reference Manual. It also support the Apollo 68080 Core ISA, which is an FPGA CPU designed by the Apollo Team. Note, that this ISA is still under development and subject to change.

The syntax for the scale factor in ColdFire MAC instructions is `<<` for left- and `>>` for right-shift. The scale factor may be appended as an optional operand, when needed. Example: `mac d0.l,d1.u,<<`.

The mask flag in MAC instructions is written as `&` and is appended directly to the effective address operand. Example: `mac d0,d1,(a0)&,d2`.

The register list in MOVEM, FMOVEM, etc. instructions may optionally be specified by an immediate addressing mode, using a 16 or 8 (FMOVEM) bit register mask constant. Example (push no registers): `movem.l #0,-(sp)`.

The target address type is 32 bit. Floating point constants in instructions and data are supported and encoded in IEEE format.

Default alignment for instructions is 2 bytes. The default alignment for data is 2 bytes, when the data size is larger than 8 bits. Note, that data is not auto-aligned unless you specify the `-align` option (or use Devpac-compatibility mode: `-devpac`).

25.4 Internal symbols

Depending on the selected cpu type the `__VASM` symbol will have a value defined by the following bits:

bit 0	MC68000 instruction set. Also used by MC6830x, MC68322, MC68356.
bit 1	MC68010 instruction set.
bit 2	MC68020 instruction set.
bit 3	MC68030 instruction set.
bit 4	MC68040 instruction set.
bit 5	MC68060 instruction set.
bit 6	MC68881 or MC68882 FPU.
bit 7	MC68851 PMMU.
bit 8	CPU32. Any MC6833x or MC6834x CPU.
bit 9	ColdFire ISA_A.
bit 10	ColdFire ISA_A+.
bit 11	ColdFire ISA_B.
bit 12	ColdFire ISA_C.

bit 13	ColdFire hardware division support.
bit 14	ColdFire MAC instructions.
bit 15	ColdFire enhanced MAC instructions.
bit 16	ColdFire USP register.
bit 17	ColdFire FPU instructions.
bit 18	ColdFire MMU instructions.
bit 20	Apollo Core AC68080 instruction set.

The following symbols are defined for compatibility with other assemblers, so their function is not described here.

Devpac	__G2, __LK
PhxAss	__CPU, __FPU, __MMU, __OPTC
BAsm	_MOVEMBYTES, __MOVEMREGS

25.5 Extensions

This backend extends the selected syntax module by the following directives:

.sdreg <An>	Equivalent to near <An>.
basereg <expression>, <An>	Starts a block of base-relative addressing through register An (remember that A7 is not allowed as a base register). The developer has to make sure that <expression> is placed into An first, while the assembler automatically subtracts <expression>, which is usually a program label with an optional offset, from each displacement in a (d, An) addressing mode. basereg has priority over the near directive. Its effect can be suspended with the endb directive. It is allowed to use several base registers in parallel.
cpu32	Generate code for the CPU32 family.
endb <An>	Ends a basereg block and suspends its effect onto the specified base register An . It may be reused with a different base expression thereafter (refer to basereg).
far	Disables small data (base-relative) mode. All data references will be absolute.
fpu <cpID>	Enables 68881/68882 FPU code generation. The <cpID> is inserted into the FPU instructions to select the correct coprocessor. Note that <cpID> is always 1 for the on-chip FPUs in the 68040 and 68060. A <cpID> of zero will disable FPU code generation.
initnear	Initializes the selected small data base register. In contrast to PhxAss, where this directive comes from, just a reference to _LinkerDB is generated, which has to be resolved by a linker: lea _LinkerDB, An

machine <cpu_type>

Makes the assembler generate code for <cpu_type>, which can be the following: 68000, 68010, 68020, 68030, 68040, 68060, 68080, 68851, 68881, 68882, cpu32. And various ColdFire CPUs, starting with 5...

mc68000 Generate code for the MC68000 CPU.

mc68010 Generate code for the MC68010 CPU.

mc68020 Generate code for the MC68020 CPU.

mc68030 Generate code for the MC68030 CPU.

mc68040 Generate code for the MC68040 CPU.

mc68060 Generate code for the MC68060 CPU.

ac68080 Generate code for the Apollo Core AC68080 FPGA CPU.

mcf5... Generate code for a ColdFire CPU. The recognized models are listed in the assembler-options section.

near [<An>]

Enables small data (base-relative) mode and sets the base register to **An**. **near** without an argument will reactivate a previously defined small data mode, which might have been switched off by a **far** directive.

near code All **JMP** and **JSR** instructions to external labels will be converted into 16-bit PC-relative jumps. The small code mode can be switched off by a **far** directive.

opt <option>[,<option>...]

Sets Devpac-compatible options. When option **-phxass** is given, then it will parse PhxAss options instead (which is discouraged for new code, so there is no detailed description here). Most supported Devpac2-style options are always suffixed by a **+** or **-** to enable or disable the option:

- a** Automatically optimize absolute to PC-relative references. Default is off in Devpac-comptability mode, otherwise on.
- c** Case-sensitivity for all symbols and macros. Default is on. Note, that although you can change case-sensitivity multiple times in a source text, you cannot expect to access symbols from a part with different case-sensitivity.
- d** Include all symbols for debugging in the output file. May also generate line debugging information in some output formats. Default is off in Devpac-comptability mode, otherwise on.
- l** Generate a linkable object file. The default is defined by the selected output format via the assembler's **-F** option. This option was supported by Devpac-Amiga only.
- o** Enable all optimizations (o1 to o12), or disable all optimizations. The default is that all are disabled in Devpac-compatibility mode and enabled otherwise. When running in native vasm mode this option will also enable PC-relative (**opt a**) and the following safe vasm-specific optimizations (see below): **og**, **of**.

o1	Optimize branches without an explicit size extension.
o2	Standard displacement optimizations (e.g. (0,An) -> (An)).
o3	Optimize absolute addresses to short words.
o4	Optimize <code>move.l</code> to <code>moveq</code> .
o5	Optimize <code>add #x</code> and <code>sub #x</code> into their quick forms.
o6	No effect in vasm.
o7	Replace <code>bra.b</code> by a 2-byte no-operation instruction, like <code>lea (a6),a6</code> , when branching to the next instruction. Note: <code>nop</code> is not really a no-operation instruction on 68040 and higher.
o8	Optimize 68020+ base displacements to 16 bit.
o9	Optimize 68020+ outer displacements to 16 bit.
o10	Optimize <code>add/sub #x,An</code> to <code>lea</code> .
o11	Optimize <code>lea (d,An),An</code> to <code>addq/subq</code> .
o12	Optimize <code><op>.l #x,An</code> to <code><op>.w #x,An</code> .
ow	Show all optimizations being performed. Default is on in Devpac-compatibility mode, otherwise off.
p	Check if code is position independent. This will cause an error on every relocation entry being required. Default is off.
s	Include symbols in listing file. Default is on.
t	Check size and type of all expressions. Default is on.
w	Show assembler warnings. Default is on.
x	For Amiga hunk format objects <code>x+</code> strips local symbols from the symbol table (symbols without <code>xdef</code>). For Atari TOS executables this will enable the extended (HiSoft) DRI symbol table format, which allows symbols with up to 22 characters. DRI standard only supports 8 characters.

Devpac options without +/- suffix:

10	
11	
12	Nonzero selects object file, zero selects executable file format, when Atari-TOS (<code>-Ftos</code>) or Amiga-hunk output format (<code>-Fhunk</code>) was set on the command line. This option was supported by Devpac-Atari only and its original function was to select TOS-executable (0), DRI-object (1) or GST-object (2) output. For GST objects use (<code>-Fgst</code>) instead.

`p=<type>[/<type>]`

Sets the CPU type to any model vasm supports (Devpac only allows 68000-68040, 68332, 68881, 68882 and 68851).

Also the following Devpac3-style options are supported:

<code>autopc</code>	Corresponds to <code>a+</code> .
<code>case</code>	Corresponds to <code>c+</code> .
<code>chkpc</code>	Corresponds to <code>p+</code> .
<code>debug</code>	Corresponds to <code>d+</code> .
<code>syntab</code>	Corresponds to <code>s+</code> .
<code>type</code>	Corresponds to <code>t+</code> .
<code>warn</code>	Corresponds to <code>w+</code> .
<code>xdebug</code>	Corresponds to <code>x+</code> .
<code>noautopc</code>	Corresponds to <code>a-</code> .
<code>nocase</code>	Corresponds to <code>c-</code> .
<code>nochkpc</code>	Corresponds to <code>p-</code> .
<code>nodebug</code>	Corresponds to <code>d-</code> .
<code>nosyntab</code>	Corresponds to <code>s-</code> .
<code>notype</code>	Corresponds to <code>t-</code> .
<code>nowarn</code>	Corresponds to <code>w-</code> .
<code>noxdebug</code>	Corresponds to <code>x-</code> .

The following options are vasm specific and should not be used when writing portable source text. Using `opt o+` or `opt o-` in Devpac mode only toggles `og` and `of`.

<code>oa</code>	Automatically optimize absolute Apollo destination operands to PC-relative references (requires 68080 code-generation enabled).
<code>ob</code>	Convert absolute jumps to external labels into long-branches (refer to <code>-opt-jbra</code>).
<code>oc</code>	Enable optimizations to CLR (refer to <code>-opt-clr</code>).
<code>od</code>	Enable optimization of divisions into shifts (refer to <code>-opt-div</code>).
<code>of</code>	Enable immediate float constant optimizations (refer to <code>-opt-fconst</code>).
<code>og</code>	Enable generic vasm optimizations. This includes all safe optimizations which cannot be controlled by another option.
<code>oj</code>	Enable branch to jump translations (refer to <code>-opt-brajmp</code>).
<code>ol</code>	Enable shift optimizations to ADD (refer to <code>-opt-lsl</code>).
<code>om</code>	Enable MOVEM optimizations (refer to <code>-opt-movem</code>).
<code>on</code>	Enable small data optimizations. References to absolute symbols in a small data section (named " <code>--MERGED</code> ") are optimized into a base-relative addressing mode (refer to <code>-sd</code>).

<code>op</code>	Enable optimizations to PEA (refer to <code>-opt-pea</code>).
<code>oq</code>	Optimizes <code>MOVE.L</code> into a combination of <code>MOVEQ</code> and <code>NEG.W</code> (refer to <code>-opt-nmoveq</code>).
<code>os</code>	Optimize for speed before optimizing for size (refer to <code>-opt-speed</code>).
<code>ot</code>	Enable optimizations to ST (refer to <code>-opt-st</code>).
<code>ox</code>	Enable optimization of multiplications into shifts (refer to <code>-opt-mul</code>).
<code>oz</code>	Enable optimization for size, even if the code becomes slower (refer to <code>-opt-size</code>).

The default state is 'off' for all these vasm specific options, except for `of` and `og`, which are 'on'.

The following directives are only available for the Motorola syntax module:

`<symbol> equ <Rn>`

Define a new symbol named `<symbol>` and assign the data or address register `Rn`, which can be used from now on in operands. When 68080 code generation is enabled, also `Bn` base address registers and `En` vector registers are allowed to be assigned. Note that a register symbol must be defined before it can be used!

`<symbol> eql <reglist>`

Equivalent to `<symbol> reg <reglist>`.

`<symbol> feqr <FPn>`

Define a new symbol named `<symbol>` and assign the FPU register `FPn`, which can be used from now on in operands. Note that a register symbol must be defined before it can be used!

`<symbol> feql <reglist>`

Equivalent to `<symbol> freg <reglist>`.

`<symbol> freg <reglist>`

Defines a new symbol named `<symbol>` and assign the FPU register list `<reglist>` to it. Registers in a list must be separated by a slash (/) and ranges or registers can be defined by using a hyphen (-). No character at all represents an empty register list. Optionally you may specify the `<reglist>` as an 8-bit register mask constant (`fp0` is bit 0, `fp7` is bit 7). Examples for valid FPU register lists are: `fp0-fp7`, `fp1-3/fp5/fp7`, `fpiar/fpcr`.

`<symbol> reg <reglist>`

Defines a new symbol named `<symbol>` and assign the register list `<reglist>` to it. Registers in a list must be separated by a slash (/) and ranges or registers can be defined by using a hyphen (-). No character at all represents an empty register list. Optionally you may specify the `<reglist>` as a 16-bit register mask constant (`d0` is bit 0, `a7` is bit 15). Examples for valid register lists are: `d0-d7/a0-a6`, `d3-6/a0/a1/a4-5`.

25.6 Optimizations

25.6.1 Operand optimizations

This backend performs the following operand optimizations:

- (0,An) optimized to (An).
- (d16,An) translated to (bd32,An,ZDn.w), when d16 is not between -32768 and 32767 and the selected CPU allows it (68020 up or CPU32).
- (d16,PC) translated to (bd32,PC,ZDn.w), when d16 is not between -32768 and 32767 and the selected CPU allows it (68020 up or CPU32).
- (d8,An,Rn) translated to (bd,An,Rn), when d8 is not between -128 and 127 and the selected CPU allows it (68020 up or CPU32).
- (d8,PC,Rn) translated to (bd,PC,Rn), when d8 is not between -128 and 127 and the selected CPU allows it (68020 up or CPU32).
- <exp>.l optimized to <exp>.w, when <exp> is absolute and between -32768 and 32767.
- <exp>.w translated to <exp>.l, when <exp> is a program label or absolute and not between -32768 and 32767.
- (0,An,...) optimized to (An,...) (which means the base displacement will be suppressed). This allows further optimization to (An), when the index is suppressed.
- (bd16,An,...) translated to (bd32,An,...), when bd16 is not between -32768 and 32767.
- (bd32,An,...) optimized to (bd16,An,...), when bd16 is between -32768 and 32767.
- (bd32,An,ZRn) optimized to (d16,An), when bd32 is between -32768 and 32767, and the index is suppressed (zero-Rn).
- (An,ZRn) optimized to (An), when the index is suppressed.
- (0,PC,...) optimized to (PC,...) (which means the base displacement will be suppressed).
- (bd16,PC,...) translated to (bd32,PC,...), when bd16 is not between -32768 and 32767.
- (bd32,PC,...) optimized to (bd16,PC,...), when bd16 is between -32768 and 32767.
- (bd32,PC,ZRn) optimized to (d16,PC), when bd32 is between -32768 and 32767, and the index is suppressed (zero-Rn).
- ([0,Rn,...],...) optimized to ([An,...],...) (which means the base displacement will be suppressed).
- ([bd16,Rn,...],...) translated to ([bd32,An,...],...), when bd16 is not between -32768 and 32767.
- ([bd32,Rn,...],...) optimized to ([bd16,An,...],...), when bd32 is between -32768 and 32767.
- ([...],0) optimized to ([...]) (which means the outer displacement will be suppressed).
- ([...],od16) translated to ([...],od32), when od16 is not between -32768 and 32767.
- ([...],od32) translated to ([...],od16), when od32 is between -32768 and 32767.

Note that an operand optimization will only take place when a displacement's size was not enforced by the developer through an explicit size extension (e.g. (4.1,a0))!

25.6.2 Instruction optimizations

This backend performs the following instruction optimizations and translations:

- `<op>.L #x,An` optimized to `<op>.W #x,An`, when `x` is between -32768 and 32767.
- `ADD.? #x,<ea>` optimized to `ADDQ.? #x,<ea>`, when `x` is between 1 and 8.
- `ADD.? #x,<ea>` optimized to `SUBQ.? #x,<ea>`, when `x` is between -1 and -8.
- `ADD.L #x,<ea>` optimized to `ADDIW.L #x,<ea>` for Apollo, when `x` is between -32768 and 32767.
- `ADDA.? #0,An` and `SUBA.? #0,An` will be deleted.
- `ADDA.? #x,An` translated to `LEA (x,An),An`, when `x` is between -32768 and 32767.
- `ANDI.L #$fff,Dn` optimized to `MVZ.B Dn,Dn`, for ColdFire ISA_B/C or `EXTUB.L Dn` for Apollo.
- `ANDI.L #$ffff,Dn` optimized to `MVZ.W Dn,Dn`, for ColdFire ISA_B/C or `EXTUW.L Dn` for Apollo.
- `ANDI.? #0,<ea>` optimized to `CLR.? <ea>`, when allowed by the option `-opt-clr` or a different CPU than the MC68000 was selected.
- `ANDI.? #-1,<ea>` optimized to `TST.? <ea>`.
- `ASL.? #1,Dn` optimized to `ADD.? Dn,Dn` for 68000 and 68010.
- `ASL.? #2,Dn` optimized into a sequence of two `ADD.? Dn,Dn` for 68000 and 68010, when the operation size is either byte or word and the options `-opt-speed` and `-opt-lsl` are given.
- `B<cc> <label>` translated into a combination of `B!<cc> **8` and `JMP <label>`, when `<label>` is not defined in the same section (and option `-opt-brajmp` is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
- `B<cc> <label>` is automatically optimized to 8-bit, 16-bit or 32-bit (68020 up, CPU32, MCF5407 only), whatever fits best. When the selected CPU doesn't support 32-bit branches it will try to change the conditional branch into a `B!<cc> **8` and `JMP <label>` sequence.
- `BRA <label>` translated to `JMP <label>`, when `<label>` is not defined in the same section (and option `-opt-brajmp` is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
- `BSR <label>` translated to `JSR <label>`, when `<label>` is not defined in the same section (and option `-opt-brajmp` is given), or outside the range of -32768 to 32767 bytes from the current address when the selected CPU is not 68020 up, CPU32 or ColdFire ISA_B/C.
- `<cp>B<cc> <label>` is automatically optimized to 16-bit or 32-bit, whatever fits best. `<cp>` means coprocessor and is P for the PMMU and F for the FPU.
- `CLR.L Dn` optimized to `MOVEQ #0,Dn`.

- `CMP.? #0,<ea>` optimized to `TST.? <ea>`. The selected CPU type must be MC68020 up, ColdFire or CPU32 to support address register direct as effective address (`<ea>`).
- `CMP.L #x,<ea>` optimized to `CMPIW.L #x,<ea>` for Apollo, when `x` is between -32768 and 32767.
- `DIVS.W/DIVU.W #1,Dn` optimized to `MVZ.W Dn,Dn`, for ColdFire ISA-B/C (`-opt-div`).
- `DIVS.W #-1,Dn` optimized to the sequence of `NEG.W Dn` and `MVZ.W Dn,Dn` (`-opt-div` and `-opt-speed`).
- `DIVS.L/DIVU.L #1,Dn` optimized to `TST.L Dn` (`-opt-div`).
- `DIVS.L #-1,Dn` optimized to `NEG.L Dn` (`-opt-div`).
- `DIVU.L #2..256,Dn` optimized to `LSR.L #x,Dn` (`-opt-div`).
- `EORI.? #-1,<ea>` optimized to `NOT.? <ea>`.
- `EORI.? #0,<ea>` optimized to `TST.? <ea>`.
- `FMOVEM.? <reglist>` is deleted when the register list was empty.
- `FxDIV.? #m,FPn` optimized to `FxMUL.? #1/m,FPn` when `m` is a power of 2 and option `-opt-fconst` is given.
- `JMP <label>` optimized to `BRA.? <label>`, when `<label>` is defined in the same section and in the range of -32768 to 32767 bytes from the current address. Note that `JMP (<lab>,PC)` is never optimized, with the intention to preserve jump-tables.
- `JSR <label>` optimized to `BSR.? <label>`, when `<label>` is defined in the same section and in the range of -32768 to 32767 bytes from the current address. Note that `JSR (<lab>,PC)` is never optimized, with the intention to preserve jump-tables.
- `LEA 0,An` optimized to `SUBA.L An,An`.
- `LEA (0,An),An` and `LEA (An),An` will be deleted.
- `LEA (d,An),An` is optimized to `ADDQ.L #d,An` when `d` is between 1 and 8 and to `SUBQ.L #-d,An` when `d` is between -1 and -8.
- `LEA (d,Am),An` will be translated into a combination of `MOVEA` and `ADDA.L` for 68000 and 68010, when `d` is lower than -32768 or higher than 32767. The `MOVEA` will be omitted when `Am` and `An` are identical. Otherwise `-opt-speed` is required.
- `LINK.L An,#x` optimized to `LINK.W An,#x`, when `x` is between -32768 and 32767.
- `LINK.W An,#x` translated to `LINK.L An,#x`, when `x` is not between -32768 and 32767 and selected CPU supports this instruction.
- `LSL.? #1,Dn` optimized to `ADD.? Dn,Dn` for 68000 and 68010, when option `-opt-lsl` is given.
- `LSL.? #2,Dn` optimized into a sequence of two `ADD.? Dn,Dn` for 68000 and 68010, when the operation size is either byte or word and the options `-opt-speed` and `-opt-lsl` are given.
- `MOVE.? #0,<ea>` optimized to `CLR.? <ea>`, when allowed by the option `-opt-clr` or a different CPU than the MC68000 was selected.
- `MOVE.? #x,-(SP)` optimized to `PEA x`, when allowed by the option `-opt-pea`. The move-size must not be byte (`.b`).
- `MOVE.B #-1,<ea>` optimized to `ST <ea>`, when allowed by the option `-opt-st`.
- `MOVE.L #x,Dn` optimized to `MOVEQ #x,Dn`, when `x` is between -128 and 127.

- `MOVE.L #x,Dn` optimized to the sequence of `MOVEQ #x>>1,Dn` and `ADD.W Dn,Dn`, when $128 \leq x \leq 254$ and x is even.
- `MOVE.L #x,Dn` optimized to the sequence of `MOVEQ #x^$ff,Dn` and `NOT.B Dn`, when $128 \leq x \leq 255$ and the option `-opt-nmoveq` was set.
- `MOVE.L #x,Dn` optimized to the sequence of `MOVEQ #x>>16,Dn` and `SWAP Dn`, when $\$10000 \leq x \leq \$7f0000$ or $\$ff80ffff \leq x \leq \$fffeffff$.
- `MOVE.L #x,Dn` optimized to the sequence of `MOVEQ #-((int16_t)x),Dn` and `NEG.W Dn`, when $\$ff81 \leq x \leq \$ffff$ or $\$ffff0001 \leq x \leq \$ffff0080$ and the option `-opt-nmoveq` was set.
- `MOVE.L #x,Dn` optimized to the sequence of `MOVEQ #x>>n,Dn` and `LSL.W #n,Dn`, when $0 \leq x \leq \$7fff$ and the difference between the highest and lowest bit set is less than 8. `-opt-size` needs to be set together with standard optimizations.
- `MOVE.L #x,<ea>` optimized to `MOV3Q #x,<ea>`, for ColdFire ISA_B and ISA_C, when x is -1 or between 1 and 7.
- `MOVEA.? #0,An` optimized to `SUBA.L An,An`.
- `MOVEA.L #x,An` optimized to `MOVEA.W #x,An`, when x is between -32768 and 32767.
- `MOVEA.L #label,An` optimized to `LEA label,An`, which could allow further optimization to `LEA label(PC),An`.
- `MOVEM.? <reglist>` is deleted, when the register list was empty.
- `MOVEM.? <ea>,An` optimized to `MOVE.? <ea>,An`, when the register list only contains a single address register.
- `MOVEM.? <ea>,Rn` optimized to `MOVE.? <ea>,Rn` and `MOVEM.? Rn,<ea>` optimized to `MOVE.? Rn,<ea>`, when allowed by the option `-opt-movem` or when just loading an address register.
- `MOVEM.? <ea>,Rm/Rn` and `MOVEM.? Rm/Rn,<ea>` are optimized into a sequence of two `MOVE` instructions when advantageous for the currently selected CPU. For example, for 68000 and 68010 it is no advantage to optimize `MOVEM Rm/Rn,-(An)`, and addressing modes with displacements or absolute addresses are optimized for 68040 only (may additionally require `-opt-speed`).
- `MULS.?/MULU.? #0,Dn` optimized to `MOVEQ #0,Dn` (`-opt-mul`).
- `MULS.?/MULU.? #1,Dn` is deleted (`-opt-mul`).
- `MULS.W #-1,Dn` optimized to the sequence `EXT.L Dn` and `NEG.L Dn` (`-opt-mul` and `-opt-speed`).
- `MULS.L #-1,Dn` optimized to `NEG.L Dn` (`-opt-mul`).
- `MULS.W #2..256,Dn` optimized to the sequence `EXT.L Dn` and `ASL.L #x,Dn` (`-opt-mul` and `-opt-speed`).
- `MULS.W #-2..-256,Dn` optimized to the sequence `EXT.L Dn`, `ASL.L #x,Dn` and `NEG.L Dn` (`-opt-mul` and `-opt-speed`).
- `MULS.L #2..256,Dn` optimized to `ASL.L #x,Dn` (`-opt-mul`).
- `MULS.L #-2..-256,Dn` optimized to the sequence `ASL.L #x,Dn` and `NEG.L Dn` (`-opt-mul` and `-opt-speed`).
- `MULU.W #2..256,Dn` optimized to the sequence `MVZ.W Dn,Dn` and `ASL.L #x,Dn` for ColdFire ISA_B/C (`-opt-mul` and `-opt-speed`).

- MULU.L #2..256,Dn optimized to LSL.L #x,Dn (-opt-mul).
- MVZ.? #x,Dn and MVS.? #x,Dn are optimized to MOVEQ #x,Dn.
- ORI.? #0,<ea> optimized to TST.? <ea>.
- SUB.? #x,<ea> optimized to SUBQ.? #x,<ea>, when x is between 1 and 8.
- SUB.? #x,<ea> optimized to ADDQ.? #x,<ea>, when x is between -1 and -8.
- SUB.L #x,<ea> optimized to ADDIW.L #-x,<ea> for Apollo, when x is between -32767 and 32768.
- SUBA.? #x,An translated to LEA (-x,An),An, when x is between -32767 and 32768.

25.7 Known Problems

Some known problems of this module at the moment:

- In some rare cases, mainly by stupid input sources, the optimizer might oscillate forever between two states. If this happens, assembly will terminate automatically after some time.
- When using FMOVE immediate addressing modes, but without specifying a size extension, constants between \$80000000 and \$ffffffff are stored with 32 bits, which leads to sign-extension problems when the instruction is really 64 or 96 bits.

25.8 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: illegal addressing mode
- 2003: invalid register list
- 2004: missing) in register indirect addressing mode
- 2005: address register required
- 2006: bad size extension
- 2007: displacement at bad position
- 2008: base or index register expected
- 2009: missing] in memory indirect addressing mode
- 2010: no extension allowed here
- 2011: illegal scale factor
- 2012: can't scale PC register
- 2013: index register expected
- 2014: too many] in memory indirect addressing mode
- 2015: missing outer displacement
- 2016: %c expected
- 2017: can't use PC register as index
- 2018: double registers in list
- 2019: data register required

- 2020: illegal bitfield width/offset
- 2021: constant integer expression required
- 2022: value from -64 to 63 required for k-factor
- 2023: need 32 bits to reference a program label
- 2024: option expected
- 2025: absolute value expected
- 2026: operand value out of range: %ld (valid: %ld..%ld)
- 2027: label in operand required
- 2028: using signed operand as unsigned: %ld (valid: %ld..%ld), %ld to fix
- 2029: branch destination out of range
- 2030: displacement out of range
- 2031: absolute displacement expected
- 2032: unknown option %c%c ignored
- 2033: absolute short address out of range
- 2034: 8-bit branch with zero displacement was converted to 16-bit
- 2035: illegal opcode extension
- 2036: extension for unsized instruction ignored
- 2037: immediate operand out of range
- 2038: immediate operand has illegal type or size
- 2039: data objects with %d bits size are not supported
- 2040: data out of range
- 2041: data has illegal type
- 2042: illegal combination of ColdFire addressing modes
- 2043: FP register required
- 2044: unknown cpu type
- 2045: register expected
- 2046: link.w changed to link.l
- 2047: branch out of range changed to jmp
- 2048: lea-displacement out of range, changed into move/add
- 2049: translated (A%d) into (0,A%d) for movep
- 2050: operand optimized: %s
- 2051: operand translated: %s
- 2051: instruction optimized: %s
- 2053: instruction translated: %s
- 2054: branch optimized into: b<cc>.%c
- 2055: branch translated into: b<cc>.%c
- 2056: basereg A%d already in use
- 2057: basereg A%d is already free
- 2058: short-branch to following instruction turned into a nop

- 2059: not a valid small data register
- 2060: small data mode is not enabled
- 2061: division by zero
- 2062: can't use B%d register as index
- 2063: register list on both sides
- 2064: "%s" directive was replaced by an instruction with the same name
- 2065: Addr.reg. operand at level #0 causes F-line exception
- 2066: Dr and Dq are identical, transforming DIVxL.L effectively into DIVx.L
- 2068: trailing garbage in operand
- 2067: 3Q value out of range: %ld (valid: -1, 1..%ld)
- 2069: encoding absolute displacement directly
- 2070: internal symbol %s has been modified
- 2071: instruction too large for bank prefix
- 2072: bad FPU id %d for selected cpu type
- 2073: absolute k-factor without '#'
- 2074: %d-bit access to absolute address

26 PowerPC cpu module

This chapter documents the Backend for the PowerPC microprocessor family.

26.1 Legal

This module is written in 2002-2016 by Frank Wille and is covered by the vasm copyright without modifications.

26.2 Additional options for this module

This module provides the following additional options:

- `-big` Select big-endian mode.
- `-little` Select little-endian mode.
- `-many` Allow both, 32- and 64-bit instructions.
- `-mavec, -maltivec`
 Generate code for the AltiVec unit.
- `-mcom` Allow only common PPC instructions.
- `-m601` Generate code for the PPC 601.
- `-mppc32, -mppc, -m603, -m604`
 Generate code for the 32-bit PowerPC 6xx family.
- `-mppc64, -m620`
 Generate code for the 64-bit PowerPC 600 family.
- `-m7400, -m7410, -m7455`
 Generate code for the 32-bit PowerPC 74xx (G4) family.
- `-m7450` Generate code for the 32-bit PowerPC 7450.
- `-m403, -m405`
 Generate code for the IBM/AMCC 32-bit embedded 40x family.
- `-m440, -m460`
 Generate code for the AMCC 32-bit embedded 440/460 family.
- `-m821, -m850, -m860`
 Generate code for the 32-bit MPC8xx PowerQUICC I family.
- `-mbooke` Generate code for the 32-bit Book-E architecture.
- `-me300` Generate code for the 32-bit e300 core (MPC51xx, MPC52xx, MPC83xx).
- `-me500` Generate code for the 32-bit e500 core (MPC85xx), including SPE, EFS and PMR.
- `-mpwr` Generate code for the POWER family.
- `-mpwrx, -mpwr2`
 Generate code for the POWER2 family.

-no-regnames

Don't predefine any register-name symbols.

-opt-branch

Enables translation of 16-bit branches into "B<!cc> \$+8 ; B label" sequences when destination is out of range.

-sd2reg=<n>

Sets the 2nd small data base register to Rn.

-sdreg=<n>

Sets small data base register to Rn.

The default setting is to generate code for a 32-bit PPC G2, G3, G4 CPU with AltiVec support.

26.3 General

This backend accepts PowerPC instructions as described in the instruction set manuals from IBM, Motorola, Freescale and AMCC.

The full instruction set of the following families is supported: POWER, POWER2, 40x, 44x, 46x, 60x, 620, 750, 74xx, 860, Book-E, e300 and e500.

The target address type is 32 or 64 bits, depending on the selected CPU model. Floating point constants in instructions and data are supported and encoded in IEEE format.

Default alignment for sections and instructions is 4 bytes. Data is aligned to its natural alignment by default.

26.4 Extensions

This backend provides the following specific extensions:

- When not disabled by the option **-no-regnames**, the registers r0 - r31, f0 - f31, v0 - v31, cr0 - cr7, vrsave, sp, rtoc, fp, fpSCR, xer, lr, ctr, and the symbols lt, gt, so and un will be predefined on startup and may be referenced by the program.

This backend extends the selected syntax module by the following directives:

.sdreg <n>

Sets the small data base register to Rn.

.sd2reg <n>

Sets the 2nd small data base register to Rn.

26.5 Optimizations

This backend performs the following optimizations:

- 16-bit branches, where the destination is out of range, are translated into B<!cc> \$+8 and a 26-bit unconditional branch.

26.6 Known Problems

Some known problems of this module at the moment:

- No real differentiation between 403, 750, 860 instructions at the moment.
- There may still be some unsupported PPC models.

26.7 Error Messages

This module has the following error messages:

- 2002: instruction not supported on selected architecture
- 2003: constant integer expression required
- 2004: trailing garbage in operand
- 2005: illegal operand type
- 2006: missing closing parenthesis in load/store addressing mode
- 2007: relocation does not allow hi/lo modifier
- 2008: multiple relocation attributes
- 2009: multiple hi/lo modifiers
- 2010: data size %d not supported
- 2011: data has illegal type
- 2012: relocation attribute not supported by operand
- 2013: operand out of range: %ld (allowed: %ld to %ld)
- 2014: not a valid register (0-31)
- 2015: missing base register in load/store addressing mode
- 2016: missing mandatory operand
- 2017: ignoring fake operand

27 c16x/st10 cpu module

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

Note that this module is not yet fully completed!

27.1 Legal

This module is written in 2002-2004 by Volker Barthelmann and is covered by the vasm copyright without modifications.

27.2 Additional options for this module

This module provides the following additional options:

-no-translations

Do not translate between jump instructions. If the offset of a **jmp** instruction is too large, it will not be translated to **jmps** but an error will be emitted.

Also, **jmpa** will not be optimized to **jmp**.

The pseudo-instruction **jmp** will still be translated.

-jmpa A **jmp** or **jmp** instruction that is translated due to its offset being larger than 8 bits will be translated to a **jmpa** rather than a **jmps**, if possible.

27.3 General

This backend accepts c16x/st10 instructions as described in the Infineon instruction set manuals.

The target address type is 32bit.

Default alignment for sections and instructions is 2 bytes.

27.4 Extensions

This backend provides the following specific extensions:

- There is a pseudo instruction **jmp** that will be translated either to a **jmp** or **jmpa** instruction, depending on the offset.
- The **sfr** pseudo opcode can be used to declare special function registers. It has two, three or four arguments. The first argument is the identifier to be declared as special function register. The second argument is either the 16bit sfr address or its 8bit base address (0xfe for normal sfrs and 0xf0 for extended special function registers). In the latter case, the third argument is the 8bit sfr number. If another argument is given, it specifies the bit-number in the sfr (i.e. the declaration declares a single bit).

Example:

```
.sfr    zeros,0xfe,0x8e
```

- **SEG** and **SOF** can be used to obtain the segment or segment offset of a full address.

Example:

```
mov r3,#SEG farfunc
```

27.5 Optimizations

This backend performs the following optimizations:

- `jmp` is translated to `jmp`, if possible. Also, if `-no-translations` was not specified, `jmp` and `jmpa` are translated.
- Relative jump instructions with an offset that does not fit into 8 bits are translated to a `jmp` instruction or an inverted jump around a `jmp` instruction.
- For instruction that have two forms `gpr,#IMM3/4` and `reg,#IMM16` the smaller form is used, if possible.

27.6 Known Problems

Some known problems of this module at the moment:

- Lots...

27.7 Error Messages

This module has the following error messages:

- 2001: illegal operand
- 2002: word register expected
- 2004: value does not find in %d bits
- 2005: data size not supported
- 2006: illegal use of SOF
- 2007: illegal use of SEG
- 2008: illegal use of DPP prefix

28 6502 cpu module

This chapter documents the backend for the MOS/Rockwell 6502 microprocessor family. It also supports the Rockwell/WDC 65C02, the 45GS02 from the MEGA65 project, the Hudson Soft HuC6280 and the WDC 65802/65816 instruction sets.

28.1 Legal

This module is written in 2002,2006,2008-2012,2014-2024 by Frank Wille and is covered by the vasm copyright without modifications.

28.2 Additional options for this module

This module provides the following additional options:

- 6280 Recognize all HuC6280 instructions. Includes `setdp $2000`, so zero-page addressing modes will be automatically used from `$2000` to `$20ff`.
- 802 Same as `-816`. There is no difference in the instruction set.
- 816 Enables the 8/16 bit instruction set for the WDC65816/65802 and additional directives to switch loading of the accumulator and/or the index register between 8 and 16 bits. The target address size becomes 24 bits.
- am Automatically mask values to match their data size or the size of immediate addressing, which effectively disables any range checks on immediate and data values in the assembler and linker.
- bbcade Swap meaning of `<` and `>` selectors for compatibility with the BBC ADE assembler.
- c02 Recognize all 65C02 instructions. This excludes DTV (`-dtv`) and illegal (`-illegal`) instructions.
- ce02 Enables the Commodore CSG65CE02 instruction set, which extends on the WDC02 instruction set.
- dpo Generate 8-bit offset instead of absolute relocations when accessing a zero- or direct-page symbol.
- dtv Recognize the three additional C64-DTV instructions.
- illegal Allow 'illegal' 6502 instructions to be recognized.
- mega65 Enables the 45GS02 instruction set for the MEGA65 computer.
- opt-branch Enables translation of `B<cc>` branches into sequences of `B!<cc> *+5 ; JMP label` when necessary. `BRA` (DTV, 65C02) is directly translated into a `JMP` when out of range. It also performs optimization of `JMP` to `BRA`, whenever possible.
- wdc02 Recognize all 65C02 instructions and the WDC65C02 extensions (`RMB`, `SMB`, `BBR`, `BBS`, `STP`, `WAI`).

28.3 General

This backend accepts 6502 family instructions as described in the instruction set reference manuals from MOS and Rockwell, which are valid for the following CPUs: 6502 - 6518, 6570, 6571, 6702, 7501, 8500, 8502.

Optionally accepts 65C02 family instructions as described in the instruction set reference manuals from Rockwell and WDC. Also supports the WDC extensions in the W65C02 and W65C134.

Optionally accepts 65CE02 family instructions as described in the instruction set reference manuals from Commodore Semiconductor Group.

Optionally accepts HuC6280 instructions as described in the instruction set reference manuals from Hudson Soft.

Optionally accepts 45GS02 instructions as defined by the Mega65 project.

Optionally accepts WDC65816 instructions as described in the Programming Manual by The Western Design Center.

The target address type is 16 bits, or 24 bits in WDC65816 mode.

Instructions consist of one up to three bytes for the standard 6502 family (up to 7 bytes for the 6280) and require no alignment. There is also no alignment requirement for sections and data.

All known mnemonics for illegal 6502 instructions are optionally recognized (e.g. `dcm` and `dcp` refer to the same instruction). Some illegal instructions (e.g. `$ab`) are known to show unpredictable behaviour, or do not always work the same on different CPUs.

Note that the WDC65816's `MVN` and `MVP` block move instructions require a full 24-bit address (or a label) for the source and destination, as documented in WDC's Programming Manual. To specify the bank only, you have to use immediate addressing syntax. Example: `mvn #$7f, ^label`.

28.4 Extensions

Note that some of these extensions have changed with version 1.0 of this backend. Bitstream-selectors (to get the low-byte, high-byte, etc. of an expression) are no longer available as unary operators, to avoid inconsistencies, and to better conform with classic assemblers and the official WDC syntax.

This backend provides the following specific extensions:

- Immediate addressing mode operands and data directives allow bitstream selector prefixes as the first character in the operand. `<` selects the least significant bits which fit into the current immediate or data field width (AKA low-byte for 8-bit data). `>` selects the same, but shifts the value right by 8 bits first (AKA high-byte for 8-bit data). `^` or ``` shifts the value by 16 bits first (AKA bank-byte for 8-bit data). Refer to chapter 6.3.3.4 (Byte Selection Operator) of the W65C816S datasheet. These selector prefixes are always valid, no matter if 6502 or 65816 mode is active. See also option `-bbcade`, which swaps the meaning of the `<` and `>` selectors.
- Other operands allow an addressing mode selector prefix as the first character in the operand field, which can be used as a hint for the assembler in case the best addressing

mode cannot be determined (e.g. externally defined symbols), or to enforce a specific addressing mode (Example: `lda >$12` to enforce a 16-bit instead of a zero-page address). `<` enforces 8-bit (direct/zero-page) addressing. `|` or `!` enforces 16-bit (absolute) addressing. `>` enforces 24-bit (long absolute) addressing in 65816 mode, 16-bit otherwise. Refer to chapter 6.3.3.5 of the W65C816S datasheet.

- The prefix character `?` can be used in immediate addressing modes and data directives to retrieve a symbol's memory/bank ID. Note, that this feature depends on a special relocation type, which is only supported by `vlink` and requires the `V0BJ` format.

This backend extends the selected syntax module by the following directives:

- a8** Declares that immediate instructions accessing the accumulator have 8 bits width (default, WDC65816 only).
- a16** Declares that immediate instructions accessing the accumulator have 16 bits width (WDC65816 only).
- as** Alias for **a8** (WDC65816 only).
- a1** Alias for **a16** (WDC65816 only).
- cpu <name>**
 Define the cpu model. Must be specified before any code is generated and has priority over cpu settings on the command line. Most common names are recognized, like: `65c02`, `wdc02`, `65816`, `HU6280`, `45gs02`, `6510`, etc.
- <symbol> ezp <expr>**
 Works exactly like the `equ` directive, but marks `<symbol>` as a zero/direct page symbol and use zero page addressing whenever `<symbol>` is used in a memory addressing mode.
- longa on|off**
 Turns 16-bit accumulator width on or off. WDC-style alias for **a16/a8** (WDC65816 only).
- longi on|off**
 Turns 16-bit index register width on or off. WDC-style alias for **x16/x8** (WDC65816 only).
- setdp <expr>**
 Set the current base address of the zero/direct page for optimizations from absolute to direct-page addressing modes. Can be set to any 16-bit address on 65816 (defaults to zero). Is preset to `$2000` for the HuC6280/PC-Engine.
- x8** Declares that immediate instructions accessing the index registers have 8 bits width (default, WDC65816 only).
- x16** Declares that immediate instructions accessing the index registers have 16 bits width (WDC65816 only).
- xs** Alias for **x8** (WDC65816 only).
- x1** Alias for **x16** (WDC65816 only).

zero Switch to a zero/direct page section called **zero** or **.zero**, which has the type **bss** with attributes "**aurwz**". Accesses to symbols from this section will default to zero page addressing mode.

zpage <symbol1> [, <symbol2> ...]

Mark symbols as zero/direct page and use zero page addressing for expressions based on this symbol, unless overridden by an addressing mode selector (like >).

All these directives are also available in the form starting with a dot (.).

28.5 Optimizations

This backend performs the following operand optimizations and translations:

- Absolute addressing modes are optimized to zero-page (or direct-page) addressing modes, whenever possible.
- An absolute addressing mode will be promoted to long addressing (24 bits) for the WDC65816, when needed.
- Conditional branches, where the destination is out of range, are translated into **B<!cc> *+5** and an absolute **JMP** instruction (**-opt-branch**).
- Some CPUs also allow optimization of **JMP** to **BRA**, when **-opt-branch** was given.

28.6 Known Problems

Some known problems of this module at the moment:

- None?

28.7 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: selector prefix ignored
- 2004: data size %d not supported
- 2005: relocation does not allow hi/lo modifier
- 2006: operand doesn't fit into %d bits
- 2007: branch destination out of range
- 2008: illegal bit number
- 2009: identifier expected
- 2010: bad operand
- 2011: zero/direct-page addressing not available
- 2012: operand not in zero/direct-page range
- 2013: absolute-long addressing not available
- 2014: cpu must be defined before any code is generated
- 2015: unknown cpu model: %s

29 SPC700 cpu module

This chapter documents the backend for the SONY SPC700 CPU.

29.1 Legal

This module is written in 2025 by Frank Wille and is covered by the vasm copyright without modifications.

29.2 Additional options for this module

This module provides the following additional options:

- am** Automatically mask values to match their data size or the size of immediate addressing, which effectively disables any range checks on immediate and data values in the assembler and linker.
- dpo** Generate 8-bit offset instead of absolute relocations when accessing a zero- or direct-page symbol.
- opt-branch** Enables translation of `B<cc>` branches into sequences of `B<!cc> *+5 ; JMP label` when necessary. `BRA` is directly translated into a `JMP` when out of range. It also performs optimization of `JMP` to `BRA`, whenever possible.

29.3 General

This backend accepts SPC700 instructions using the official names and syntax as provided by Sony.

The target address type is 16 bits.

Instructions consist of one up to three bytes and require no alignment. There is also no alignment requirement for sections and data.

29.4 Extensions

This backend provides the following specific extensions:

- Immediate addressing mode operands and data directives allow bitstream selector prefixes as the first character in the operand. `<` selects the least significant bits which fit into the current immediate or data field width (AKA low-byte for 8-bit data). `>` selects the same, but shifts the value right by 8 bits first (AKA high-byte for 8-bit data).
- The prefix character `?` can be used in immediate addressing modes and data directives to retrieve a symbol's memory/bank ID. Note, that this feature depends on a special relocation type, which is only supported by `vlink` and requires the `VOBJ` format.
- The prefix character `!` can be used to enforce 16-bit addressing modes. Otherwise the assembler tries to optimize operands to direct page (8-bit) addressing whenever possible.

This backend extends the selected syntax module by the following directives:

- <symbol> equd <expr>**
 Works exactly like the **equ** directive, but marks <symbol> as a direct page symbol and use direct page addressing whenever <symbol> is used in a memory addressing mode.
- p0**
 Informs the assembler about the current state of the P flag. P0 assumes the direct page to reside at \$0000 (default). This information is used to decide whether absolute address references can be optimized to direct page addressing modes.
- p1**
 Informs the assembler about the current state of the P flag. P1 assumes the direct page to reside at \$0100. This information is used to decide whether absolute address references can be optimized to direct page addressing modes.
- direct**
 Switch to a direct page section called **direct** or **.direct**, which has the type **bss** with attributes **"aurwz"**. Accesses to symbols from this section will default to direct page addressing mode.
- dpage <symbol1> [,<symbol2>...]**
 Mark symbols as direct page and use direct page addressing for expressions based on this symbol, unless overridden by the addressing mode selector **!**.

All these directives are also available in the form starting with a dot (.).

29.5 Optimizations

This backend performs the following operand optimizations and translations:

- Absolute addressing modes are optimized to direct-page addressing modes, whenever possible.
- Conditional branches, where the destination is out of range, are translated into **B<!cc> *+5** and an absolute **JMP** instruction (**-opt-branch**).
- **JMP** may be optimized to **BRA**, when **-opt-branch** was given.

29.6 Known Problems

Some known problems of this module at the moment:

- None?

29.7 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: selector prefix ignored
- 2004: data size %d not supported
- 2005: operand doesn't fit into %d bits
- 2006: branch destination out of range

- 2007: illegal bit number
- 2008: identifier expected
- 2009: operand not in direct-page range
- 2010: unsuitable addressing mode for retrieving memory id
- 2011: no memory id defined, assuming 0

30 ARM cpu module

This chapter documents the backend for the Advanced RISC Machine (ARM) microprocessor family.

30.1 Legal

This module is written in 2004,2006,2010-2015,2025 by Frank Wille and is covered by the vasm copyright without modifications.

30.2 Additional options for this module

This module provides the following additional options:

- `-a2` Generate code compatible with ARM V2 architecture.
- `-a3` Generate code compatible with ARM V3 architecture.
- `-a3m` Generate code compatible with ARM V3m architecture.
- `-a4` Generate code compatible with ARM V4 architecture.
- `-a4t` Generate code compatible with ARM V4t architecture.
- `-big` Output big-endian code and data.
- `-little` Output little-endian code and data (default).
- `-m2` Generate code for the ARM2 CPU.
- `-m250` Generate code for the ARM250 CPU.
- `-m3` Generate code for the ARM3 CPU.
- `-m6` Generate code for the ARM6 CPU.
- `-m600` Generate code for the ARM600 CPU.
- `-m610` Generate code for the ARM610 CPU.
- `-m7` Generate code for the ARM7 CPU.
- `-m710` Generate code for the ARM710 CPU.
- `-m7500` Generate code for the ARM7500 CPU.
- `-m7d` Generate code for the ARM7d CPU.
- `-m7di` Generate code for the ARM7di CPU.
- `-m7dm` Generate code for the ARM7dm CPU.
- `-m7dmi` Generate code for the ARM7dmi CPU.
- `-m7tdmi` Generate code for the ARM7tdmi CPU.
- `-m8` Generate code for the ARM8 CPU.
- `-m810` Generate code for the ARM810 CPU.
- `-m9` Generate code for the ARM9 CPU.

- m9** Generate code for the ARM9 CPU.
- m920** Generate code for the ARM920 CPU.
- m920t** Generate code for the ARM920t CPU.
- m9tdmi** Generate code for the ARM9tdmi CPU.
- msa1** Generate code for the SA1 CPU.
- mstrongarm**
 Generate code for the STRONGARM CPU.
- mstrongarm110**
 Generate code for the STRONGARM110 CPU.
- mstrongarm1100**
 Generate code for the STRONGARM1100 CPU.
- opt-adr** The ADR directive will be automatically converted into ADRL if required (which inserts an additional ADD/SUB to calculate an address).
- opt-ldrpc**
 The maximum range in which PC-relative symbols can be accessed through LDR and STR is extended from +/-4KB to +/-1MB (or +/-256 Bytes to +/-65536 Bytes when accessing half-words). This is done by automatically inserting an additional ADD or SUB instruction before the LDR/STR.
- thumb** Start assembling in Thumb mode.

30.3 General

This backend accepts ARM instructions as described in various ARM CPU data sheets. Additionally some architectures support a second, more dense, instruction set, called THUMB. There are special directives to switch between these two instruction sets.

The target address type is 32bit.

Default alignment for instructions is 4 bytes for ARM and 2 bytes for THUMB. Sections will be aligned to 4 bytes by default. Data is aligned to its natural alignment by default.

30.4 Extensions

This backend extends the selected syntax module by the following directives:

- .arm** Generate 32-bit ARM code.
- .ltorg** Dump the contents of the current literal pool at this address (aligned to 32 bits). Literal pool references from the following lines will go into a new pool.
- .thumb** Generate 16-bit THUMB code.

30.5 Literal Pool

The LDR instruction will load constants or label addresses from a literal pool, if prefixed by `=`. Example:

```
ldr    r0,=0x12345678
ldr    r0,label
```

The literal pool is managed automatically by this backend. Its position for literals from the current section can be defined by the `.ltorg` directive. Pool references from the code will always access the subsequent pool, at a higher address. Never the previous one. If there are still entries in the pool at the end of a section an automatic pool dump is enforced there.

Optimizations are considered, to avoid unnecessary pool references (see below). Duplicate entries in a single pool are avoided.

30.6 Optimizations

This backend performs the following optimizations and translations for the ARM instruction set:

- LDR/STR `Rd,symbol`, with a distance between symbol and PC larger than 4KB, is translated to `ADD/SUB Rd,PC,#offset&0xff000 + LDR/STR Rd,[Rd,#offset&0xff]`, when allowed by the option `-opt-ldrpc`.
- ADR `Rd,symbol` is translated to `ADD/SUB Rd,PC,#rotated_offset8`.
- ADRL `Rd,symbol` is translated to `ADD/SUB Rd,PC,#hi_rotated8 + ADD/SUB Rd,Rd,#lo_rotated8`. ADR will be automatically treated as ADRL when required and when allowed by the option `-opt-adr`.
- The immediate operand of ALU-instructions will be translated into the appropriate 8-bit-rotated value. When rotation alone doesn't succeed the backend will try it with inverted and negated values (inverting/negating the ALU-instruction too). Optionally you may specify the rotate constant yourself, as an additional operand.
- References to the literal pool are optimized to MOV (for constants) or ADR (for label addresses) whenever possible.

For the THUMB instruction set the following optimizations and translations are done:

- A conditional branch with a branch-destination being out of range is translated into `B<!cc> .+4 + B label`.
- The BL instruction is translated into two sub-instructions combining the high- and low 22 bit of the branch displacement.

30.7 Known Problems

Some known problems of this module at the moment:

- Literal pool for thumb-mode is not yet implemented.
- Only instruction sets up to ARM architecture V4t are supported.

30.8 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: label from current section required
- 2004: branch offset (%ld) is out of range
- 2005: PC-relative load/store (offset %ld) out of range
- 2006: cannot make rotated immediate from PC-relative offset (0x%lx)
- 2007: constant integer expression required
- 2008: constant (0x%lx) not suitable for 8-bit rotated immediate
- 2009: branch to an unaligned address (offset %ld)
- 2010: not a valid ARM register
- 2011: PC (r15) not allowed in this mode
- 2012: PC (r15) not allowed for offset register Rm
- 2013: PC (r15) not allowed with write-back
- 2014: register r%ld was used multiple times
- 2015: illegal immediate shift count (%ld)
- 2016: not a valid shift register
- 2017: 24-bit unsigned immediate expected
- 2018: data size %d not supported
- 2019: illegal addressing mode: %s
- 2020: signed/halfword ldr/str doesn't support shifts
- 2021: %d-bit immediate offset out of range (%ld)
- 2022: post-indexed addressing mode expected
- 2023: operation not allowed on external symbols
- 2024: ldc/stc offset has to be a multiple of 4
- 2025: illegal coprocessor operation mode or type: %ld\n
- 2026: %d-bit unsigned immediate offset out of range (%ld)
- 2027: offset has to be a multiple of %d
- 2028: instruction at unaligned address
- 2029: TSTP/TEQP/CMNP/CMPP deprecated on 32-bit architectures
- 2030: rotate constant must be an even number between 0 and 30: %ld
- 2031: %d-bit unsigned constant required: %ld
- 2032: literal pool has no references

31 80x86 cpu module

This chapter documents the Backend for the 80x86 microprocessor family.

31.1 Legal

This module is written in 2005-2006,2011,2015-2019,2024 by Frank Wille and is covered by the vasm copyright without modifications.

31.2 Additional options for this module

This module provides the following additional options:

-cpudebug=<n>	Enables debugging output.
-m8086	Generate code for the 8086 CPU.
-mi186	Generate code for the 80186 CPU.
-mi286	Generate code for the 80286 CPU.
-mi386	Generate code for the 80386 CPU.
-mi486	Generate code for the 80486 CPU.
-mi586	Generate code for the Pentium.
-mi686	Generate code for the PentiumPro.
-mpentium	Generate code for the Pentium.
-mpentiumpro	Generate code for the PentiumPro.
-mk6	Generate code for the AMD K6.
-mathlon	Generate code for the AMD Athlon.
-msledgehammer	Generate code for the Sledgehammer CPU.
-m64	Generate code for 64-bit architectures (x86_64).

31.3 General

This backend accepts 80x86 instructions as described in the Intel Architecture Software Developer's Manual.

The target address type is 32 bits. It is 64 bits when the x86_64 architecture was selected (-m64). Floating point constants in instructions and data are supported and encoded in IEEE format.

Instructions do not need any alignment. Data is aligned to its natural alignment by default. The backend uses AT&T-syntax! This means the left operands are always the source and the right operand is the destination. Register names have to be prefixed by a '%'.

The operation size is indicated by a 'b', 'w', 'l', etc. suffix directly appended to the mnemonic. The assembler can also determine the operation size from the size of the registers being used.

31.4 Extensions

Predefined register symbols in this backend:

- 8-bit registers: `al cl dl bl ah ch dh bh axl cxl dxl spl bpl sil dil r8b r9b r10b r11b r12b r13b r14b r15b`
- 16-bit registers: `ax cx dx bx sp bp si di r8w r9w r10w r11w r12w r13w r14w r15w`
- 32-bit registers: `eax ecx edx ebx esp ebp esi edi r8d r9d r10d r11d r12d r13d r14d r15d`
- 64-bit registers: `rax rcx rdx rbx rsp ebp rsi rdi r8 r9 r10 r11 r12 r13 r14 r15`
- segment registers: `es cs ss ds fs gs`
- control registers: `cr0 cr1 cr2 cr3 cr4 cr5 cr6 cr7 cr8 cr9 cr10 cr11 cr12 cr13 cr14 cr15`
- debug registers: `dr0 dr1 dr2 dr3 dr4 dr5 dr6 dr7 dr8 dr9 dr10 dr11 dr12 dr13 dr14 dr15`
- test registers: `tr0 tr1 tr2 tr3 tr4 tr5 tr6 tr7`
- MMX and SIMD registers: `mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7 xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7 xmm8 xmm9 xmm10 xmm11 xmm12 xmm13 xmm14 xmm15`
- FPU registers: `st st(0) st(1) st(2) st(3) st(4) st(5) st(6) st(7)`

This backend extends the selected syntax module by the following directives:

- `.code16` Sets the assembler to 16-bit addressing mode.
- `.code32` Sets the assembler to 32-bit addressing mode, which is the default.
- `.code64` Sets the assembler to 64-bit addressing mode.

31.5 Optimizations

This backend performs the following optimizations:

- Immediate operands are optimized to the smallest size which can still represent the absolute value.
- Displacement operands are optimized to the smallest size which can still represent the absolute value.
- Jump instructions are optimized to 8-bit displacements, when possible.

31.6 Known Problems

Some known problems of this module at the moment:

- 64-bit operations are incomplete and experimental.

31.7 Error Messages

This module has the following error messages:

- 2001: instruction not supported on selected architecture
- 2002: trailing garbage in operand
- 2003: same type of prefix used twice
- 2004: immediate operand illegal with absolute jump
- 2005: base register expected
- 2006: scale factor without index register
- 2007: missing ')' in baseindex addressing mode
- 2008: redundant %s prefix ignored
- 2009: unknown register specified
- 2010: using register %%%s instead of %%%s due to '%c' suffix
- 2011: %%%s not allowed with '%c' suffix
- 2012: illegal suffix '%c'
- 2013: instruction has no suffix and no register operands - size is unknown
- 2015: memory operand expected
- 2016: you cannot pop %%%s
- 2017: translating to %s %%%s, %%%s
- 2018: translating to %s %%%s
- 2019: absolute scale factor required
- 2020: illegal scale factor (valid: 1,2,4,8)
- 2021: data objects with %d bits size are not supported
- 2022: need at least %d bits for a relocatable symbol
- 2023: pc-relative jump destination out of range (%lld)
- 2024: instruction doesn't support these operand sizes
- 2025: cannot determine immediate operand size without a suffix
- 2026: displacement doesn't fit into %d bits

32 Z80 cpu module

This chapter documents the backend for the 8080/z80/gbz80/64180/RCMx000 microprocessor family.

32.1 Legal

This module is copyright in 2009 by Dominic Morris.

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.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND 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 AUTHOR 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.

32.2 Additional options for this module

This module provides the following additional options:

- 8080 Turns on 8080/8085 compatibility mode. Any use of z80 (or higher) opcodes will result in an error being generated.
- gbz80 Turns on gbz80 compatibility mode. Any use of non-supported opcodes will result in an error being generated.
- hd64180 Turns on 64180 mode supporting additional 64180 opcodes.
- intel-syntax
 Turns on the older Intel 8080 syntax mode. When this mode is activated, mnemonics and operand types from the Intel 8080 syntax instead of the Zilog Z80 syntax (such as `STA 1234h` instead of `ld (1234h),a`) will be valid. This option can be used in parallel with -8080 to use both sets of mnemonics, although this is discouraged, as two instructions (`jp` and `cp`) mean different things in each syntax. In this case, these instructions will be assembled as the Intel syntax, and a warning will be emitted.
- rcm2000

- rcm3000**
- rcm4000** Turns on Rabbit compatibility mode, generating the correct codes for moved opcodes and supporting the additional Rabbit instructions. In this mode, 8 bit access to the 16 bit index registers is not permitted.
- rcmemu** Turns on emulation of some instructions which aren't available on the Rabbit processors.
- swapixiy** Swaps the usage of ix and iy registers. This is useful for compiling generic code that uses an index register that is reserved on the target machine.
- z80asm** Switches on z80asm mode. This translates ASMPc to \$ and accepts some pseudo opcodes that z80asm supports. Most emulation of z80asm directives is provided by the oldsyntax syntax module.

32.3 General

This backend accepts z80 family instructions in standard Zilog syntax. Rabbit opcodes are accepted as defined in the publicly available reference material from Rabbit Semiconductor, with the exception that the `ljp` and `lcall` opcodes need to be supplied with a 24 bit number rather than an 8 bit `xpc` and a 16 bit address.

The target address type is 16 bit.

Instructions consist of one up to six bytes and require no alignment. There is also no alignment requirement for sections and data.

The current PC symbol is \$.

32.4 Extensions

This backend provides the following specific extensions:

- Certain Rabbit opcodes can be prefixed by the `altd` and/or the `ioi/ioe` modifier. For details of which instructions these are valid for please see the documentation from Rabbit.
- The parser understands a `lo/hi`-modifier to select low- or high-byte of a 16-bit word. The character `<` is used to select the low-byte and `>` for the high-byte. It has to be the first character before an expression.
- When applying the operation `/256`, `%256` or `&255` on a label, an appropriate `lo/hi`-byte relocation will automatically be generated.

32.5 Optimisations

This backend supports the emulation of certain z80 instructions on the Rabbit/gbz80 processor. These instructions are `rld`, `rrd`, `cpi`, `cpir`, `cpd` and `cpdr`. The link stage should provide routines with the opcode name prefixed with `rcmx_` (eg `rcmx_rld`) which implements the same functionality. Example implementations are available within the z88dk CVS tree.

Additionally, for the Rabbit targets the missing call `cc`, opcodes will be emulated.

32.6 Known Problems

Some known problems of this module at the moment:

- Not all RCM4000 opcodes are supported (`llcall`, `lljp` are not available).

32.7 Error Messages

This module has the following error messages:

- 2001: trailing garbage in operand
- 2002: Opcode not supported by %s (%s)
- 2003: Index registers not available on 8080
- 2004: out of range for 8 bit expression (%d)
- 2005: invalid bit number (%d) should be in range 0..7
- 2006: rst value out of range (%d/0x%02x)
- 2007: %s value out of range (%d)
- 2008: index offset should be a constant
- 2009: invalid branch type for jr
- 2010: Rabbit target doesn't support rst %d
- 2011: Rabbit target doesn't support 8 bit index registers
- 2012: z180 target doesn't support 8 bit index registers
- 2013: invalid branch type for jre
- 2014: Opcode not supported by %s (%s) but it can be emulated (-rcmemu)
- 2015: %s specifier is only valid for Rabbit processors
- 2016: Only one of ioi and ioe can be specified at a time
- 2017: %s specifier is not valid for the opcode %s
- 2018: %s specifier redundant for the opcode %s
- 2019: %s specifier has no effect on the opcode %s
- 2020: Operand value must evaluate to a constant for opcode %s
- 2021: Unhandled operand type wanted 0x%x got 0x%x
- 2022: Missed matched index registers on %s
- 2023: Only out (c),0 is supported for the opcode %s
- 2024: Operations between different index registers are forbidden
- 2025: Operations between ix/iy/hl are forbidden
- 2026: Double indirection forbidden
- 2027: Instruction %s will be assembled as '%s'
- 2028: index offset out of bounds (%d)

33 6800 cpu module

This chapter documents the backend for the Motorola 6800 microprocessor family.

33.1 Legal

This module is written in 2013-2016,2021 by Esben Norby and Frank Wille and is covered by the vasm copyright without modifications.

33.2 Additional options for this module

This module provides the following additional options:

- m6800 Generate code for the 6800 CPU (default setting).
- m6801 Generate code for the 6801 CPU.
- m68hc11 Generate code for the 68HC11 CPU.

33.3 General

This backend accepts 6800 family instructions for the following CPUs:

- 6800 code generation: 6800, 6802, 6808.
- 6801 code generation: 6801, 6803.
- 68HC11.

The 6804, 6805 and 68HC08 are not supported, they use a similar instruction set, but are not opcode compatible.

The target address type is 16 bit.

Instructions consist of one up to five bytes and require no alignment. There is also no alignment requirement for sections and data.

33.4 Extensions

This backend provides the following specific extensions:

- When an instruction supports direct and extended addressing mode the < character can be used to force direct mode and the > character forces extended mode. Otherwise the assembler selects the best mode automatically, which defaults to extended mode for external symbols.
- When applying the operation /256, %256 or &256 on a label, an appropriate lo/hi-byte relocation will automatically be generated.

33.5 Optimizations

None.

33.6 Known Problems

Some known problems of this module at the moment:

- None?

33.7 Error Messages

This module has the following error messages:

- 2001: trailing garbage in operand
- 2002: data size %d not supported
- 2003: operand doesn't fit into 8-bits
- 2004: branch destination out of range

34 6809/6309/68HC12 cpu module

This chapter documents the backend for the Motorola 6809, 68HC12 and Hitachi 6309.

34.1 Legal

This module is written in 2020-2025 by Frank Wille and is covered by the vasm copyright without modifications.

34.2 Additional options for this module

This module provides the following additional options:

- `-6809` Generate code for the 6809 CPU (default setting). Also works on the 6309, which is backwards compatible.
- `-6309` Generate code for the 6309 CPU.
- `-68hc12` Generate code for the 68HC12 CPU.
- `-konami2` Generate code for the original revision (052001) of Konami's 6809 based arcade CPU (Konami-2).
- `-konami2ext`
 Generate code for later revisions (052526, 053248) of Konami's 6809 based arcade CPU (Konami-2), which have extra instructions over the original revision.
- `-opt-branch`
 Translate short-branches to long and optimize long-branches to short when required/possible. Also tries to optimize `jmp` and `jsr` instructions into short-branches.
- `-opt-offset`
 Delete zero offsets in indexed addressing modes, when possible.
- `-opt-pc` Convert all extended addressing modes with local or external labels to indexed, PC-relative addressing. Also translates absolute `jmp/jsr` instructions into PC-relative `lbra/lbsr` (or better).
- `-turbo9` Generate code for the TURBO9 CPU.

34.3 General

This backend accepts 6809/6309 instructions as described in the Motorola 6809 and Hitachi 6309 Programmer's Reference (Copyright 2009 Darren Atkinson). Optionally supports the 68HC12 instruction set as documented in Motorola's CPU12 Reference Manual.

The target address type is 16 bit.

Instructions consist of one up to six bytes and require no alignment. There is also no alignment requirement for sections and data.

34.4 Extensions

The backend supports the unary operators `<` and `>` to either select the size of an addressing mode or the LSB/MSB of a 16-bit word.

- When direct or extended addressing modes are applicable, `<` enforces direct mode and `>` enforces extended mode.
- For indexed, PC-relative addressing modes `<` enforces an 8-bit offset, while `>` enforces a 16-bit offset.
- For all other indexed addressing modes with a constant offset `<` enforces an 8-bit offset and `>` enforces a 16-bit offset.
- For immediate addressing modes or data constants `<` selects the LSB of a word and `>` selects the MSB.
- When applying the operation `/256`, `%256` or `&256` on a label in immediate addressing modes or data constants, an appropriate lo/hi-byte relocation will automatically be generated.

In absence of `<` or `>` vasm selects the best addressing mode possible, i.e. the one which requires the least amount of memory when the symbol value is known, or the one which allows the largest symbol values, when it is unknown at assembly time.

This backend extends the selected syntax module by the following directives:

setdp `<expr>`

Set the current base address of the direct page. It is used to decide whether an extended addressing mode can be optimized to direct addressing. No effect for 68HC12.

direct `<symbol>`

Tell the assembler to use direct addressing for expressions based on this symbol.

34.5 Optimizations

This backend performs the following operand optimizations:

- Short-branches (**Bcc**) are optionally translated into long-branches (**LBcc**) when their destination is undefined, in a different section or out of range. Note that there is no **LBSR** on the HC12.
- Long-branches (**LBcc**) are optionally optimized into short-branches (**Bcc**) when possible.
- Optionally optimize **JMP** into **BRA** and **JSR** into **BSR** when possible (same section, distance representable in 8 bits).
- Optionally translate **JMP** into **LBRA** and **JSR** into **LBSR** (`-opt-pc`).
- Optionally translate all extended addressing, referring to local or external labels, to an indexed, PC-relative addressing mode (`-opt-pc`).
- Optionally optimize indexed addressing modes with a constant zero offset into a no-offset addressing mode, which executes faster. Not possible on the HC12.
- Always select the shortest indexed addressing mode for constant offsets, unless externally defined.

34.6 Known Problems

Some known problems of this module at the moment:

- HC12 PC-relative addressing modes in `MOVx` instructions may be wrong. Needs testing.

34.7 Error Messages

This module has the following error messages:

- 2001: trailing garbage in operand
- 2002: `] expected` for indirect addressing mode
- 2003: missing valid index register
- 2004: pc-relative offset out of range: `%ld`
- 2005: constant offset out of range: `%ld`
- 2006: bad auto decrement/increment value: `%d`
- 2007: indirect addressing not allowed
- 2008: auto increment/decrement not allowed
- 2009: short branch out of range: `%ld`
- 2010: long branch out of range: `%ld`
- 2011: decrement branch out of range: `%ld`
- 2012: data size `%d` not supported
- 2013: data expression doesn't fit into `%d` bits
- 2014: illegal bit number specification: `%d`
- 2015: omitted offset taken as 5-bit zero offset
- 2016: immediate expression doesn't fit into `%d` bits
- 2017: directive ignored as selected CPU has no DP register
- 2018: double size modifier ignored
- 2019: addressing mode not supported

35 Jaguar RISC cpu module

This chapter documents the backend for the Atari Jaguar GPU/DSP RISC processor.

35.1 Legal

This module is written in 2014-2017,2020,2021,2024,2025 by Frank Wille and is covered by the vasm copyright without modifications.

35.2 Additional options for this module

This module provides the following additional options:

- big** Output big-endian code and data (default).
- little** Output little-endian code and data.
- many** Generate code for GPU or DSP RISC. All instructions are accepted (default).
- mdsp**
- mjerry** Generate code for the DSP RISC (part of Jerry).
- mgpu**
- mtom** Generate code for the GPU RISC (part of Tom).
- no-opt** Disable all standard optimizations which do not cause any side effects.
- opt-jr=<n>**
 Enable automatic translation of JR instructions with an out-of-range destination into `MOVEI dest,Rn` followed by `JUMP cc,(Rn)`, where `Rn` is always the selected temporary register.

35.3 General

This backend accepts RISC instructions for the GPU or DSP in Atari's Jaguar custom chip set according to the "Jaguar Technical Reference Manual for Tom & Jerry", Revision 8. Documentation bugs were fixed by using various sources on the net.

The target address type is 32 bits.

Default alignment for instructions is 2 bytes. Data is aligned to its natural alignment by default.

35.4 Optimizations

This backend performs the following optimizations and translations for the GPU/DSP RISC instruction set:

- `load (Rn+0),Rm` is optimized to `load (Rn),Rm`.
- `store Rn,(Rm+0)` is optimized to `store Rn,(Rm)`.
- `movei #x,Rn` is optimized to `moveq #x,Rn` when `x` is a constant from 0 to 31.
- `jr cc,label` is translated to `movei label,Rn` plus `jump cc,(Rn)`, when enabled by option `-opt-jr=<n>`.

35.5 Extensions

This backend extends the selected syntax module by the following directives (note that a leading dot is optional):

- <symbol> ccdef <expression>**
 Allows defining a symbol for the condition codes used in `jump` and `jr` instructions. Must be constant number in the range of 0 to 31 or another condition code symbol.
- ccundef <symbol>**
 Undefine a condition code symbol previously defined via `ccdef`.
- dsp**
 Select DSP instruction set.
- <symbol> equr <Rn>**
 Define a new symbol named <symbol> and assign the address register `Rn` to it. <Rn> may also be another register symbol. Note that a register symbol must be defined before it can be used.
- equrundef <symbol>**
 Undefine a register symbol previously defined via `equr`.
- gpu**
 Select GPU instruction set.
- <symbol> regequ <Rn>**
 Equivalent to `equr`.
- regundef <symbol>**
 Undefine a register symbol previously defined via `regequ`.

All directives may be optionally preceded by a dot (.), for compatibility with various syntax modules.

35.6 Known Problems

Some known problems of this module at the moment:

- Encoding of `MOVEI` instruction in little-endian mode is unknown.
- The developer has to provide the necessary `NOP` instructions after jumps, or `OR` instructions to work around hardware bugs, her/himself.
- Some rare relocations are not perfectly supported. For example `LOAD Rn+x` or `ADDQ #x,Rn`, when external symbol `x` becomes 32.

35.7 Error Messages

This module has the following error messages:

- 2001: data size %d not supported
- 2002: value from %d to %d required
- 2003: register expected

36 PDP11 cpu module

This chapter documents the backend for the PDP-11 CPU architecture.

36.1 Legal

This module is written in 2020 by Frank Wille and is covered by the vasm copyright without modifications.

36.2 Additional options for this module

This module provides the following additional options:

- `-eis` Enables the Extended Instruction Set option (EIS).
- `-fis` Enables the Floating point Instruction Set option (FIS).
- `-msp` Enables additional memory space instructions.
- `-opt-branch`
 Enables optimization of `jmp` instructions to `br` when possible and translates `br` instructions to `jmp` when required. It will also translate conditional branches, where the destination is out of range, into a `jmp` instruction and a negated conditional branch over this `jmp`.

36.3 General

This backend accepts PDP-11 instructions as described in the PDP11/40 Processor Handbook, by Digital Equipment Corporation.

The target address type is 16 bit.

Instructions consist of two up to six bytes and required 16-bit alignment. Data, when not accessed as single bytes, also requires 16-bit alignment.

36.4 Known Problems

Some known problems of this module at the moment:

- Doesn't support all PDP-11 extensions.

36.5 Error Messages

This module has the following error messages:

- 2001: bad addressing mode
- 2002: bad register
- 2003: pc-relative destination out of range: %ld (valid: %d..%d)
- 2004: bad trap code %ld (valid: %d..%d)
- 2005: displacement out of range: %ld
- 2006: immediate value out of range: %ld
- 2007: absolute address out of range: %ld
- 2008: data size %d not supported
- 2009: data expression doesn't fit into %d bits

37 unSP cpu module

This chapter documents the backend for the unSP CPU architecture.

37.1 Legal

This module is written in 2021-2024 by Adrien Destugues and is covered by the vasm copyright without modifications.

37.2 General

Instructions consist of one 16-bit word, sometimes followed by a 16-bit immediate value or address.

Single byte memory accesses are not possible.

The address space is 22 bits, with a segment register which is used by adding a D: prefix to addresses (otherwise, only 16 bit addresses to the "zero page" are accessible). The segment register is adjusted automatically when a post or pre increment or decrement overflows the address register. There is also a segment register for the PC, but that does not require specific handling from the assembler at the moment (except for the presence of an LJMP instruction allowing to call code in another segment).

Conditional jump instructions in unSP are PC relative and allow to jump up to 63 instructions forward or backwards. When the code in a conditional branch is longer than this, a 2-instruction sequence is used: an opposite condition jump to PC+2, followed by a GOTO to the branch target. This is done automatically as needed by vasm if you use the "l" prefix to the jump instruction (for example LJAЕ instead of JAE). In this case, the long form instruction is used if necessary, and the short form is used if the target is close enough. Without the prefix, the instruction can only use the PC-relative mode and you get an assembler error if you try to jump too far.

37.3 Extensions

This backend provides the following specific extensions:

- The parser understands a lo/hi-modifier to select low- or high-byte of a 22-bit word. The character < is used to select the low-byte and > for the high-byte. It has to be the first character before an expression.
- When applying the operation /65536, %65536 or &65535 on a label, an appropriate lo/hi-byte relocation will automatically be generated.

37.4 Compatibility with other assemblers and disassemblers

This backend accepts instructions in a traditional format, reusing the mnemonics from naked_asm. The official unSP documentation uses an unusual syntax, which would probably require a custom syntax module.

The closest syntax to naked_asm is the oldstyle one, but there are some differences:

- Instructions must be prefixed by a tab or spaces. Everything on the first column is parsed as a label.

- Addresses can be represented with parentheses in addition to square brackets: LD R1, (1234) instead of LD R1, [1234]
- The push/pop instructions list of registers is indicated as two operands instead of a range: PUSH R1, R1, (SP) instead of PUSH R1-R1, [SP]
- Hexadecimal numbers can be prefixed with \$ instead of 0x

The backend internally outputs data in big endian format. This is not compatible with other tools, which use little endian. The endianness can be changed at the output stage when generating a raw binary, by using vasm standard -ole command-line switch.

37.5 Known Problems

Only version 1.0 and 1.1 of the ISA is supported (they are identical as far as instruction encoding is concerned, but a few instructions have slight behavior differences). Versions 1.2 and 1.3 are backwards compatible but introduce additional instructions:

- CPU control: SECBANK ON/OFF; FRACTION ON/OFF; IRQNEST ON/OFF
- Load and stores of DS and flag registers
- Bit operations: TSTB, SETB, CLRB, INVB (with registers or indirect addressing)
- Shift operations: ASR, ASROR, LSL, LSLOR, LSR, LSROR, ROL, ROR
- Multiplication and MAC: unsigned * unsigned
- Divisions: DIVQ, DIVS, EXP
- Jumps: GOTO and CALL with indirect address (GOTO MR, CALL MR)

Version 2.0 is not fully compatible and introduces even more instructions.

38 SWEET16 cpu module

This chapter documents the backend for the SWEET16 Pseudo CPU architecture.

38.1 Legal

This module is written in 2026 by Frank Wille and is covered by the vasm copyright without modifications.

38.2 General

This backend accepts SWEET16 instructions as defined by Steve Wozniak and used in the Apple II computer ROM. An interpreter for this pseudo CPU, with 16 16-bit registers, can also be ported to other 6502-based systems.

The target address type is 16 bit.

Instructions consist of one, two (branches) or three bytes (**SET** instruction) and require no alignment. Data requires no alignment either.

38.3 Known Problems

Some known problems of this module at the moment:

- None.

38.4 Error Messages

This module has the following error messages:

- 2001: trailing garbage in operand
- 2002: branch destination out of range
- 2003: data size %d not supported
- 2004: data expression doesn't fit into %d bits
- 2005: %d cannot be a register

39 HANS cpu module

This chapter documents the backend for the "Hans" processor from the "Hans" project.

39.1 Legal

This module is written in 2023-2024 by Yannik Stamm and is covered by the vasm copyright without modifications.

39.2 Additional options for this module

39.3 General

This backend is made as part of a hobby project "Hans" which includes a custom processor with a custom instruction set. The project is open source under: <https://github.com/Byter64/Hans> (most part is German).

The instructions as well as all data is exactly 32-Bit, which is equal to 1 Byte on our system. Therefore, the target address size is also 32-Bit.

39.4 Known Problems

Some known problems of this module at the moment:

- None.

39.5 Error Messages

This module has the following error messages:

- 2001: data size of "%d" bits is not supported
- 2002: value does not fit into immediate. Allowed range is [-32768:32767]. Value is %i
- 2003: destination address (immediate value) is to far away. Allowed distance/immediate value is [-32768:32767]. Actual distance/immediate value is %i
- 2004: congratulations. Your jump distance (%i) actually goes beyond the scope of [-33,554,432:33,554,431]. We, the creators of Hans, are so proud of you for writing 33 million instructions. You should maybe take a break :D
- 2005: immediate is %i but allowed range is %s for %s

40 Trillek TR3200 cpu module

This chapter documents the Backend for the TR3200 cpu.

40.1 Legal

This module is written in 2014 by Luis Panadero Guardeno and is covered by the vasm copyright without modifications.

40.2 General

This backend accepts TR3200 instructions as described in the TR3200 specification (<https://github.com/trillek-team/trillek-computer>)

The target address type is 32 bits.

Default alignment for sections is 4 bytes. Instructions alignment is 4 bytes. Data is aligned to its natural alignment by default, i.e. 2 byte wide data alignment is 2 bytes and 4 byte wide data alignment is 4 bytes.

The backend uses TR3200 syntax! This means the left operand is always the destination and the right operand is the source (except for single operand instructions). Register names have to be prefixed by a '%' (%bp, %r0, etc.) This means that it should accept WaveAsm assembly files if oldstyle syntax module is being used. The instructions are lower case, -dotdir option is being used and directives are not in the first column.

40.3 Extensions

Predefined register symbols in this backend:

- register by number: r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15
- special registers by name: bp sp y ia flags

40.4 Known Problems

Some known problems of this module at the moment:

- This module needs to be fully checked, but has been tested building a full program that could be found here : <https://github.com/Zardoz89/trillek-firmware>
- Instruction relocations are missing.

40.5 Error Messages

This module has the following error messages:

- 2001: illegal operand
- 2002: illegal qualifier <%s>
- 2003: data size not supported

40.6 Example

It follows a little example to illustrate TR3200 assembly using the oldstyle syntax module (option `-dotdir` required):

```

const    .equ 0xBEBACAFE          ; A constant
an_addr  .equ 0x100                ; Other constant

; ROM code
        .org 0x100000
        .text
_start  ; Label with or without a ending ":"
        mov %sp, 0x1000            ; Set the initial stack

        mov %r0, 0
        mov %r1, 0xA5
        mov %r2, 0
        storeb %r0, an_addr, %r1

        add %r0, %r2, %bp
        add %r0, %r2, 0
        add %r0, %r0, 10
        add %r0, %r0, 10h
        add %r0, %r0, 0x100010
        add %r0, %r0, (256 + 100) ; vasm parses math expressions

        mov %r2, 0
        mov %r3, 0x10200

        loadb %r6, 0x100200
        loadb %r1, %r2, 0x100200
        loadb %r1, %r2, %r3
        loadb %r4, var1

        push %r0
        .repeat 2                  ; directives to repeat stuff!
        push const
        .endrepeat

        .repeat 2
        pop %r5
        .endrepeat
        pop %r0

        rcall foo                  ; Relative call/jump!
        sleep

foo:    ; Subroutine

```

```
    ifneq %r5, 0
        mul %r5, %r5, 2
        sub %r5, %r5, 1

    ret

; ROM data
    .org 0x100500
var1  .db 0x20                ; A byte size variable
    .even                    ; Enforce to align to even address
var3  .dw 0x1020              ; A word size variable
var4  .dd 0x0A0B0C20          ; A double word size variable
str1  .ascii "Hello world!"   ; ASCII string with null termination
str2  .string "Hello world!"  ; ASCII string with null termination
    .fill 5, 0xFF             ; Fill 5 bytes with 0xFF
    .reserve 5                ; Reserves space for 5 byte
```


41 Interface

41.1 Introduction

This chapter is under construction!

This chapter describes some of the internals of **vasm** and tries to explain what has to be done to write a cpu module, a syntax module or an output module for **vasm**. However, if someone wants to write one, I suggest to contact me first, so that it can be integrated into the source tree.

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.

41.2 Building vasm

This section deals with the steps necessary to build the typical **vasm** executable from the sources.

41.2.1 Directory Structure

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

vasm/ The main directory containing the assembler sources.

vasm/Makefile
 The Makefile used to build **vasm**.

vasm/syntax/<syntax-module>/
 Directories for the syntax modules.

vasm/cpus/<cpu-module>/
 Directories for the cpu modules.

vasm/obj/
 Directory the object files will be stored in.

All compiling is done from the main directory and the executables will be placed there as well. The main assembler for a combination of **<cpu>** and **<syntax>** will be called **vasm<cpu>_<syntax>**. All output modules are usually integrated in every executable and can be selected at runtime. Otherwise you have to adapt the **OUTFMTS** definition in **make.rules** and select those you want.

41.2.2 Adapting the Makefile

Before building anything you have to insert correct values for your compiler and operating system in the **Makefile**.

TARGET Here you may define an extension which is appended to the executable's name. Useful, if you build various targets in the same directory.

TARGETEXTENSION

Defines the file name extension for executable files. Not needed for most operating systems. For Windows it would be `.exe`.

CC Here you have to insert a command that invokes an ANSI C compiler you want to use to build vasm. It must support the `-I` option in the same way like e.g. `vc` or `gcc`.

COPTS Here you will usually define an option like `-c` to instruct the compiler to generate an object file. Additional options, like the optimization level, should also be inserted here as well. Specifying the host OS helps to determine work-directories for DWARF and defines the appropriate internal symbol for the host's file system path style. The following are supported:

-DAMIGA AmigaOS (M68k or PPC), MorphOS, AROS. Defines the internal symbol `__AMIGAFS`.

-DATARI Atari TOS. Defines the internal symbol `__MSDOSFS`.

-DMSDOS CP/M, MS-DOS, Windows. Defines the internal symbol `__MSDOSFS`.

-DUNIX All kinds of Unix (Linux, BSD) including MacOSX and Atari-MiNT. Defines the internal symbol `__UNIXFS`.

-D_WIN32 Windows. Defines the internal symbol `__MSDOSFS`.

Building without specifying a host-OS is allowed. Then vasm defaults to Unix-style path handling and will not define a file system symbol for conditional assembly. Other options:

-DLOWMEM Builds for a host-OS with a low amount of memory. This will basically reduce all hash tables to minimal size.

CCOUT Here you define the option which is used to specify the name of an output file, which is usually `-o`.

LD Here you insert a command which starts the linker. This may be the the same as under **CC**.

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

LDOUT Here you define the option which is used by the linker to specify the output file name.

RM Specify a command to delete a file, e.g. `rm -f`.

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

```
TARGET = _os3
TARGETEXTENSION =
CC = vc +aos68k
CCOUT = -o
COPTS = -c -c99 -cpu=68020 -DAMIGA -O1
LD = $(CC)
```



```
LDOUT = $(CCOUT)
LDFLAGS = -lmieee
RM = delete force quiet
```

An example for a typical Unix-installation would be:

```
TARGET =
TARGETEXTENSION =
CC = gcc
CCOUT = -o
COPTS = -c -O2
LD = $(CC)
LDOUT = $(CCOUT)
LDFLAGS = -lm
RM = rm -f
```

Open/Net/Free/Any BSD systems will probably require an additional `-D_ANSI_SOURCE` in `COPTS`.

41.2.3 Building vasm

Note to users of BSD 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 CPU=<cpu> SYNTAX=<syntax>
```

For example:

```
make CPU=ppc SYNTAX=std
```

The following CPU modules can be selected:

- CPU=6502
- CPU=6800
- CPU=6809
- CPU=arm
- CPU=c16x
- CPU=hans
- CPU=jagrisc
- CPU=m68k
- CPU=pdp11
- CPU=ppc
- CPU=qnice
- CPU=spc700
- CPU=test
- CPU=tr3200
- CPU=unsp
- CPU=vidcore
- CPU=x86

- CPU=z80

The following syntax modules can be selected:

- SYNTAX=std
- SYNTAX=mot
- SYNTAX=madmac
- SYNTAX=oldstyle
- SYNTAX=test

For Windows and various Amiga targets there are already Makefiles included, which you may either copy on top of the default Makefile, or call it explicitly with `make`'s `-f` option:

```
make -f Makefile.OS4 CPU=ppc SYNTAX=std
```

41.3 vasm global variables

Important global variables, which may be read or modified by syntax-, cpu- or output-modules.

`source *cur_src;`

Pointer to the current source text instance (see structures below).

`char *defsectname;`

Name of a default section which vasm creates when a label or code occurs in the source without any preceding `section` or `org` directive. Assigning NULL means that the default is an absolute section and its base address is taken from `defsectorg`. These `defsect...` variables can be overridden by syntax- or output-modules.

`taddr defsectorg;`

Used when `defsectname==NULL`. Defines the base address of a default absolute `org` section.

`char *defsecttype;`

Attributes of the default section (see above). May be NULL to indicate that no default has been defined and vasm will show an error.

`char emptystr[];`

An empty string (zero length).

`int exec_out;`

Non-zero, when the output file is an executable and not an object file.

`char *filename;`

Defaults to the file part of the input source file name. Syntax modules may modify it with `setfilename(char *)` and output modules may use it for their own purpose.

`char *inname;`

Input source file name.

`int octetsperbyte;`

Number of 8-bit bytes used to represent a backend's target-byte. The macro `OCTETS(n)` may be used to calculate the number of 8-bit bytes for `n` target-bytes.

char *outname;
Output object file name.

int output_bitsperbyte;
May be assigned by an output module with 1 during its init function, to indicate that it supports target-bytes with **BITSPERBYTE**. Otherwise an output module is expected to support 8-bit bytes only.

41.4 General data structures

This section describes the fundamental data structures used in vasm which are usually necessary to understand for writing any kind of module (cpu, syntax or output). More detailed information is given in the respective sections on writing specific modules where necessary.

41.4.1 Source

A source structure represents a source text module, which can be either the main source text, an included file, a macro or a repetition. There is always a link to the parent source from where the current source context was included or called.

struct source *parent;
Pointer to the parent source context. Assembly continues there when the current source context ends.

int parent_line;
Line number in the parent source context, from where we were called. This information is needed, because line numbers are only reliable during parsing and later from the atoms. But an include directive doesn't create an atom.

struct source_file *srcfile;
The **source_file** structure has the unique file name, index and text-pointer for this source text instance. Used for debugging output, like DWARF.

char *name;
File name of the main source or include file, or macro name.

char *text;
Pointer to the source text start.

size_t size;
Size of the source text to assemble in bytes.

struct source *defsrc;
This is a NULL-pointer for real source text files. Otherwise it is a reference to the source which defines the current macro or repetition.

int defline;
Valid when **defsrc** is not NULL. Contains the starting line number of a macro or repetition in a source text file.

macro *macro;
Pointer to macro structure, when currently inside a macro (see also **num_params**).

unsigned long repeat;
 Number of repetitions of this source text. Usually this is 1, but for text blocks between a **rept** and **endr** (or similar) directive it allows any number of repetitions, which is decremented every time the end of the source text block is reached.

long reptn;
 Current value of the repetition counter symbol in this source instance. So it can be restored when reentering this instance.

char *reptcntname;
 Name of an optional iterator symbol (when **reptcntname** is not NULL), which is set to the current value of **reptn**.

char *irpname;
 Name of the iterator symbol in special repeat loops, which use a sequence of arbitrary values, being assigned to this symbol within the loop. Example: **irp** directive in **std-syntax**.

struct macarg *irpvals;
 A list of arbitrary values to iterate over in a loop. With each iteration the frontmost value is removed from the list until it is empty.

int cond_level;
 Current level of conditional nesting while entering this source text. It is automatically restored to the previous level when leaving the source prematurely through **end_source()**.

struct macarg *argnames;
 The current list of named macro arguments.

int num_params;
 Number of macro parameters passed at the invocation point from the parent source. For normal source files this entry will be -1. For macros 0 (no parameters) or higher.

char *param[MAXMACPARAMS];
 Pointers to the macro arguments.

int param_len[MAXMACPARAMS];
 Number of characters per macro argument.

int num_qual;
 (If **MAX_QUALIFIERS**!=0.) Number of qualifiers for a macro. when not passed on invocation these are the default qualifiers.

char *qual[MAX_QUALIFIERS];
 (If **MAX_QUALIFIERS**!=0.) Pointer to macro qualifiers.

int qual_len[MAX_QUALIFIERS];
 (If **MAX_QUALIFIERS**!=0.) Number of characters per macro qualifier.

unsigned long id;
 Every source has its unique id. Useful for macros supporting the special **\@** argument for creating unique labels.

char *srcptr;
 The current source text pointer, pointing to the beginning of the next line to assemble.

int line; Line number in the current source context. After parsing, the line number of the current atom is stored here.

size_t bufsize;
 Current size of the line buffer (**linebuf**). The size of the line buffer is extended automatically, when an overflow happens.

char *linebuf;
 A buffer for the current line being assembled in this source text. A child-source, like a macro, can refer to arguments from this buffer, so every source has got its own. When returning to the parent source, the linebuf is deallocated to save memory.

expr *cargexp;
 (If **CARGSYM** was defined.) Pointer to the current expression assigned to the CARG-symbol (used to select a macro argument) in this source instance. So it can be restored when reentering this instance.

41.4.2 Sections

One of the top level structures is a linked list of sections describing continuous blocks of memory. A section is specified by an object of type **section** with the following members that can be accessed by the modules:

struct section *next;
 A pointer to the next section in the list.

char *name;
 The name of the section.

char *attr;
 A string describing the section flags in ELF notation (see, for example, documentation of the **.section** directive in the standard syntax module).

atom *first;
atom *last;
 Pointers to the first and last atom of the section. See following sections for information on atoms.

taddr align;
 Alignment of the section in target-bytes.

uint32_t flags;
 Flags of the section. Currently available flags are:

HAS_SYMBOLS
 At least one symbol is defined in this section.

RESOLVE_WARN
 The current atom changed its size multiple times, so **atom_size()** is now called with this flag set in its section to make the back-

end (e.g. `instruction_size()`) aware of it and do less aggressive optimizations.

UNALLOCATED

Section is unallocated, which means it doesn't use any memory space in the output file. Such a section will be removed before creating the output file and all its labels converted into absolute expression symbols. Used for "offset" sections. Refer to `switch_offset_section()`.

LABELS_ARE_LOCAL

As long as this flag is set new labels in a section are defined as local labels, with the section name as global parent label.

ABSOLUTE Section is loaded at an absolute address in memory.

PREVABS Remembers state of the **ABSOLUTE** flag before entering relocated-org mode (**IN_RORG**). So it can be restored later.

IN_RORG Section has entered relocated-org mode, which also sets the **ABSOLUTE** flag. In this mode code is written into the current section, but relocated to an absolute address. No relocation table entries are generated.

NEAR_ADDRESSING

Section is marked as suitable for cpu-specific "near" addressing modes. For example, base-register relative or zero/direct-page. The cpu backend can use this information as an optimization hint when referencing symbols from this section.

FAR_ADDRESSING

Section requires cpu-specific "far" addressing modes. For example an addressing mode including the bank or "segment". The cpu backend may use this information to select appropriate addressing modes when referencing symbols from this section.

`taddr org;`

Start address of a section. Usually zero. Set to an absolute start address in **ABSOLUTE** mode.

`taddr pc;` Current address in this section. Can be used while traversing through the section. Has to be updated by a module using it. Is set to `org` at the beginning.

`unsigned long idx;`

A member usable by the output module for private purposes.

41.4.3 Symbols

Symbols are represented by a linked list of type `symbol` with the following members that can be accessed by the modules:

`int type;` Type of the symbol. Available are:

`#define LABSYM 1`

The symbol is a label defined at a specific location.

```
#define IMPORT 2
    The symbol is externally defined and its value is unknown.

#define EXPRESSION 3
    The symbol is defined using an expression (equate).

uint32_t flags;
    Flags of this symbol. Available are:

#define TYPE_UNKNOWN 0
    The symbol has no type information.

#define TYPE_OBJECT 1
    The symbol defines an object.

#define TYPE_FUNCTION 2
    The symbol defines a function.

#define TYPE_SECTION 3
    The symbol defines a section.

#define TYPE_FILE 4
    The symbol defines a file.

#define EXPORT (1<<3)
    The symbol is exported to other object files.

#define INEVAL (1<<4)
    Used internally.

#define COMMON (1<<5)
    The symbol is a common symbol and also has a size. It will be
    allocated by the linker.

#define WEAK (1<<6)
    The symbol is weak, which means the linker may overwrite it with
    any global definition of the same name. Weak symbols may also
    stay undefined, in which case the linker would assign them a value
    of zero.

#define LOCAL (1<<7)
    Only informational. A symbol can be explicitly declared as local
    by a syntax-module directive. Otherwise all symbols without the
    EXPORT flag are not considered for object linking.

#define VASMINTERN (1<<8)
    Vasm-internal symbol, which must not be exported into an object
    file.

#define PROTECTED (1<<9)
    Used internally to protect the current-PC symbol from deletion.

#define REFERENCED (1<<10)
    Symbol was referenced in the source and a relocation entry has
    been created.
```

```
#define ABSLABEL (1<<11)
    Label was defined inside an absolute section, or during a code block
    in relocated-org mode. Therefore it has an absolute address and will
    not generate a relocation entry when being referenced.

#define EQUATE (1<<12)
    Symbols flagged as EQUATE are constant expressions and their value
    must not be changed.

#define REGLIST (1<<13)
    Symbol is a register list definition.

#define USED (1<<14)
    Symbol appeared in an expression. Symbols which were only de-
    fined, (as label or equate) and otherwise never appear throughout
    the whole source, don't get this flag set.

#define NEAR (1<<15)
    Symbol may be referenced by "near" addressing mode. For exam-
    ple, base register relative. Used as an optimization hint to the cpu
    backend.

#define XDEF (1<<16)
    This symbol must become defined in the source. Which means its
    type must not remain IMPORT. Otherwise a warning is displayed.

#define XREF (1<<17)
    Symbol is externally defined and its type must never become some-
    thing else than IMPORT. Otherwise an error is displayed.

#define SYMINDIR (1<<18)
    For EXPRESSION symbols only: the expression's base-symbol repre-
    sents the referenced indirect symbol name.

#define RSRVD_S (1L<<24)
    The range from bit 24 to 27 (counted from the LSB) is reserved for
    use by the syntax module.

#define RSRVD_O (1L<<28)
    The range from bit 28 to 31 (counted from the LSB) is reserved for
    use by the output module.
```

The type-flags can be extracted using the **TYPE()** macro which expects a pointer to a symbol as argument.

```
char *name;
```

The name of the symbol.

```
expr *expr;
```

The expression in case of **EXPRESSION** symbols.

```
expr *size;
```

The size of the symbol in target-bytes, if specified. Common symbols always have a size.

`section *sec;`

The section a LABSYM symbol is defined in.

`taddr pc;` The address of a LABSYM symbol.

`taddr align;`

The alignment of the symbol in target-bytes.

`unsigned long idx;`

A member usable by the output module for private purposes.

The case-sensitivity of symbols defaults to case-sensitive, which can be changed by the command line option `-nocase`. Syntax- or cpu-modules may change it in their init-function, or even during parsing, by calling `set_nocase_symbols(int nc)` (where `nc=1` means case-insensitive, `nc=0` means case-sensitive and `nc=-1` resets to the state defined by the command line). But note, that you cannot expect to access symbols from a part of the input source with different case-sensitivity.

41.4.4 Register symbols

Optional register symbols are available when the backend defines `HAVE_REGSYMS` in `cpu.h` and initializes the system before first use, typically during `init_cpu()`.

For initialization you have to specify the hash table size and the case-sensitivity, using one of the following functions:

`int init_reg syms(size_t htsize,int no_case)`

Initializes the register symbol hash table with the given size and case-sensitivity (nonzero for case-insensitive). Returns non-zero on success.

`int init_reg syms_c(size_t htsize)`

Same as above, for case-sensitive symbols.

`int init_reg syms_nc(size_t htsize)`

Same as above, for case-insensitive symbols.

`int init_reg syms_sc(size_t htsize)`

Uses the symbol default case-sensitivity.

`void set_nocase_reg syms(int nc)`

Change case-sensitivity for register symbols to case-insensitive (`nc==1`), case-sensitive (`nc==0`) or using the symbol default (`nc==-1`). The same note as for `set_nocase_symbols()` (see above) applies.

A register symbol is defined by an object of type `regsym` with the following members that can be accessed by the modules:

`char *reg_name;`

Symbol name.

`int reg_type;`

Optional type of register.

`unsigned int reg_flags;`

Optional register symbol flags.

```
unsigned int reg_num;
    Register number or value.
```

Refer to `symbol.h` for functions to create and find register symbols.

41.4.5 Atoms

The contents of each section are a linked list built out of non-separable atoms. The general structure of an atom is:

```
struct atom {
    struct atom *next;
    int type;
    taddr align;
    taddr lastsize;
    unsigned changes;
    source *src;
    int line;
    listing *list;
    union {
        instruction *inst;
        dblock *db;
        symbol *label;
        sblock *sb;
        defblock *defb;
        void *opts;
        int srcline;
        char *ptext;
        printexpr *pexpr;
        expr *roffs;
        taddr *rorg;
        assertion *assert;
        aoutnlist *nlist;
    } content;
};
```

The members have the following meaning:

```
struct atom *next;
    Pointer to the following atom (NULL if last).
```

```
int type; The type of the atom. Can be one of
```

```
#define VASMDEBUG 0
    Used for internal debugging.
```

```
#define LABEL 1
    A label is defined here.
```

```
#define DATA 2
    A fixed number of target-bytes with constant data are put here.
```

#define INSTRUCTION 3

Generally refers to a machine instruction or pseudo/opcode. These atoms can change their size during optimization passes and will be translated to **DATA**-atoms later.

#define SPACE 4

Defines a block of data filled with one value of a given size (up to **MAXPADSIZE** 8-bit bytes). BSS sections usually contain only such atoms, but they are also sometimes useful as shorter versions of **DATA**-atoms in other sections.

#define DATADEF 5

Defines data of fixed size which can contain cpu specific operands and expressions. Usually generated by data in a source text, which are no machine instructions. Will be translated to **DATA**-atoms later.

#define LINE 6

A source text line number (usually from a higher level language) is bound to the atom's address. Useful for source level debugging in certain ABIs.

#define OPTS 7

A means to change assembler options at a specific source text line. For example optimization settings, or the cpu type to generate code for. The cpu backend has to define **HAVE_CPU_OPTS** and export the required functions if it wants to use this type of atom.

#define PRINTTEXT 8

A string is printed to stdout during the final assembler pass. A newline is automatically appended.

#define PRINTEXPR 9

Prints the value of an expression during the final assembler pass to stdout.

#define ROFFS 10

Set the program counter to an address relative to the section's start address. These atoms will be translated into **SPACE** atoms in the final pass.

#define RORG 11

Assemble this block under the given base address, while the code is still written into the original memory region.

#define RORGEND 12

Ends a **RORG** block and returns to the original addressing.

#define ASSERT 13

The assertion expression is checked in the final pass and an error message is generated (using the expression string and an optional message out of this atom) when it evaluates to 0.

```
#define NLIST 14
```

Defines a stab-entry for the a.out object file format. nlist-style stabs can also occur embedded in other object file formats, like ELF.

```
taddr align;
```

The alignment of this atom. Address must be dividable by `align`.

```
taddr lastsize;
```

The size of this atom in the last resolver pass. When the size has changed in the current pass, the assembler will request another resolver run through the section.

```
unsigned changes;
```

Number of changes in the size of this atom since pass number `FASTOPTPHASE`. An increasing number usually indicates a problem in the cpu backend's optimizer and will be flagged by setting `RESOLVE_WARN` in the Section flags, as soon as `changes` exceeds `MAXSIZECHANGES`. So the backend can choose not to optimize this atom as aggressive as before.

```
source *src;
```

Pointer to the source text object to which this atom belongs.

```
int line;
```

The source line number that created this atom.

```
listing *list;
```

Pointer to the listing file object to which this atom belongs.

```
instruction *inst;
```

(In union `content`.) Pointer to an instruction structure in the case of an `INSTRUCTION`-atom. Contains the following elements:

```
int code;
```

The cpu specific code of this instruction.

```
char *qualifiers[MAX_QUALIFIERS];
```

(If `MAX_QUALIFIERS!=0`.) Pointer to the qualifiers of this instruction.

```
operand *op[MAX_OPERANDS];
```

(If `MAX_OPERANDS!=0`.) The cpu-specific operands of this instruction.

```
instruction_ext ext;
```

(If the cpu backend defines `HAVE_INSTRUCTION_EXTENSION`.) A cpu-specific typedef. Typically used to store appropriate opcodes, allowed addressing modes, supported cpu derivatives etc.

```
dblock *db;
```

(In union `content`.) Pointer to a `dblock` structure in the case of a `DATA`-atom. Contains the following elements:

```
taddr size;
```

The number of target-bytes stored in this atom.

```
uint8_t *data;
```

A pointer to the constant data. Consider using `writebyte()` or `setval()` to write target-bytes which are different from 8 bits. The

internal ordering of target-bytes on an 8-bit host is big-endian, and a target-byte will always allocate the next minimal number of 8-bit host-bytes. The macro `OCTETS(n)` may be used to calculate the number of host-bytes required to represent `n` target-bytes.

`rlist *relocs;`

A pointer to relocation information for the data.

`symbol *label;`

(In union `content`.) Pointer to a symbol structure in the case of a `LABEL`-atom.

`sblock *sb;`

(In union `content`.) Pointer to a sblock structure in the case of a `SPACE`-atom. Contains the following elements:

`size_t space;`

The number of space-elements (see below) to generate here.

`expr *space_exp;`

The above size as an expression, which will be evaluated during assembly and copied to `space` in the final pass.

`size_t size;`

The size of each space-element and of the fill-pattern in target-bytes.

`uint8_t fill[MAXPADSIZE];`

The fill pattern, up to `MAXPADSIZE` 8-bit bytes.

`expr *fill_exp;`

Optional. Evaluated and copied to `fill` in the final pass, when not null.

`rlist *relocs;`

A pointer to relocation information for the space.

`taddr maxalignbytes;`

An optional number of maximum padding bytes to fulfil the atom's alignment requirement. Zero means there is no restriction.

`uint32_t flags;`

`SPC_UNINITIALIZED`

This space is completely uninitialized. May be used as a hint by output modules.

`SPC_DATABSS`

The output module should not allocate any file space for this atom, when possible (example: `DataBss` section, as supported by the "hunkexe" output file format). It is not needed to set this flag when the output section is BSS.

`defblock *defb;`

(In union `content`.) Pointer to a defblock structure in the case of a `DATADef`-atom. Contains the following elements:

```

taddr bitsize;
    The size of the definition in bits.

operand *op;
    Pointer to a cpu-specific operand structure.

void *opts;
    (In union content.) Points to a cpu-backend specific options object in the case
    of a OPTS-atom.

int srcline;
    (In union content.) Line number for source level debugging in the case of a
    LINE-atom.

char *ptext;
    (In union content.) A string to print to stdout in case of a PRINTTEXT-atom.

printexpr *pexpr;
    (In union content.) Pointer to a printexpr structure in the case of a PRINTEXPR-
    atom. Contains the following elements:

    expr *print_exp;
        Pointer to an expression to evaluate and print.

    short type;
        Format type of the printed value. We can print as hexadecimal
        (PEXP_HEX), signed decimal (PEXP_SDEC), unsigned decimal (PEXP_
        UDEC), binary (PEXP_BIN) OR ASCII (PEXP_ASC).

    short size;
        Size (precision) of the printed value in bits. Excessive bits will be
        masked out, and sign-extended when requested.

expr *roffs;
    (In union content.) The expression holds the relative section offset to align to
    in case of a ROFFS-atom.

taddr *rorg;
    (In union content.) Assemble the code under the base address in rorg in case
    of a RORG-atom.

assertion *assert;
    (In union content.) Pointer to an assertion structure in the case of an ASSERT-
    atom. Contains the following elements:

    expr *assert_exp;
        Pointer to an expression which should evaluate to non-zero.

    char *exprstr;
        Pointer to the expression as text (to be used in the output).

    char *msgstr;
        Pointer to the message, which would be printed when assert_exp
        evaluates to zero.

```

```

aoutnlist *nlist;
    (In union content.) Pointer to an nlist structure, describing an aout stab entry,
    in case of an NLIST-atom. Contains the following elements:

    char *name;
        Name of the stab symbol.

    int type;
        Symbol type. Refer to stabs.h for definitions.

    int other;
        Defines the nature of the symbol (function, object, etc.).

    int desc;
        Debugger information.

    expr *value;
        Symbol's value.

```

41.4.6 Relocations

`DATA` and `SPACE` atoms can have a relocation list attached that describes how this data must be modified when linking/relocating. They always refer to the data in this atom only.

There are a number of predefined standard relocations and it is possible to add other cpu-specific relocations. Note however, that it is always preferable to use standard relocations, if possible. Chances that an output module supports a certain relocation are much higher if it is a standard relocation.

A relocation list uses this structure:

```

typedef struct rlist {
    struct rlist *next;
    void *reloc;
    int type;
} rlist;

```

`type` identifies the relocation type. All the standard relocations have type numbers between `FIRST_STANDARD_RELOC` and `LAST_STANDARD_RELOC`. Consider `reloc.h` to see which standard relocations are available.

Standard types may be combined with the modifier flags `REL_MOD_S` for signed and `REL_MOD_U` for unsigned relocations. The default, when both flags are missing, is to check the value being inserted into the given relocation field against both ranges, signed and unsigned. Setting both flags together is illegal!

Typical signed relocations are PC-relative. An example for an unsigned relocation would be a zero- or direct-page addressing mode on the 6502/6809 families. If you are unsure, just don't set any of these flags.

To get access to the real standard relocation type, without any flags, you have to use the `STD_REL_TYPE(type)` macro.

The detailed information can be accessed via the pointer `reloc`. It will point to a structure that depends on the relocation type, so a module must only use it if it knows the relocation type.

All standard relocations point to a type `nreloc` structure with the following members:

size_t byteoffset;

Offset in target-bytes, from the start of the current **DATA** atom, to the beginning of the relocation field. This may also be the address which is used as a basis for PC-relative relocations. Or a common basis for multiple separated relocation fields, which will be translated into a single relocation type by the output module.

size_t bitoffset;

Offset in bits to the beginning of the relocation field. Adds to **byteoffset*BITSPERBYTE**. Bits are counted in a bit-stream from lower to higher address bytes. But note, that within a little-endian byte bits are counted from the LSB to the MSB, while they are counted from the MSB to the LSB for big-endian targets.

int size; The size of the relocation field in bits.

taddr mask;

The mask defines which portion of the relocated value is set by this relocation field.

taddr addend;

Value to be added to the symbol value.

symbol *sym;

The symbol referred by this relocation

To describe the meaning of these entries, we will define the steps that shall be executed when performing a relocation:

1. Extract **size** number of bits from the **DATA** atom, starting with bit number **byteoffset*BITSPERBYTE+bitoffset**. We start counting bits from the lowest to the highest numbered byte in memory. Within a big-endian byte we count from the MSB to the LSB. Within a little-endian byte we count from the LSB to the MSB.
2. Determine the relocation value of the symbol. For a simple absolute relocation, this will be the value of the symbol **sym** plus the **addend**. For other relocation types, more complex calculations will be needed. For example, in a program-counter relative relocation, the value will be obtained by subtracting the address of the data atom plus **byteoffset** from the value of **sym** plus **addend**.
3. Calculate the bit-wise AND of the value obtained in the step above and the **mask** value.
4. Normalize, i.e. shift the value above right as many bit positions as there are low order zero bits in **mask**.
5. Add this value to the value extracted in step 1.
6. Insert the low order **size** bits of this value into the **DATA** atom starting with bit **byteoffset*BITSPERBYTE+bitoffset**.

41.4.7 CPU-specific Relocations

Whenever a CPU module requires a relocation type which cannot be expressed by any standard type you have the option to define your own, beginning with **FIRST_CPU_RELOC** (defined in **reloc.h**). The last CPU-specific relocation has to be defined in your backend's

`cpu.h` by `LAST_CPU_RELOC`, which is also a hint for the assembler's core routines that CPU-specific relocations do exist.

Example:

```
/* PPC specific relocations */
#define REL_PPCEABI_SDA2 (FIRST_CPU_RELOC)
#define REL_PPCEABI_SDA21 (FIRST_CPU_RELOC+1)
#define REL_PPCEABI_SDAI16 (FIRST_CPU_RELOC+2)
#define REL_PPCEABI_SDA2I16 (FIRST_CPU_RELOC+3)
#define REL_MORPHOS_DREL (FIRST_CPU_RELOC+4)
#define REL_AMIGAOS_BREL (FIRST_CPU_RELOC+5)
#define LAST_CPU_RELOC REL_AMIGAOS_BREL
```

Also the CPU module must implement and export the following functions:

```
size_t cpu_reloc_size(rlist *)
    Return the size of your cpu-specific relocation in 8-bit host-bytes for the given
    type. If this type uses the standard nreloc structure, just return zero.

void cpu_reloc_print(FILE *, rlist *)
    Print the relocation name and its parameters to the given file. Usually it has the
    form: rname(startbyte, startbit-endbit, mask, addend, symbol). When using
    nreloc you may call print_nreloc() (refer to reloc.h) to simplify output.

void cpu_reloc_write(FILE *, rlist *)
    If this relocation type uses a cpu-specific structure, write it to the object file's
    relocation table in VOBJ format.
```

Note, that support for cpu-specific relocations has to be added to other tools reading and writing the `VOBJ` format, like `vlink` and `vobjdump`, as well. Otherwise they will be ignored (but do not cause any failure).

Also, if you want cpu-specific relocations to be recognized in other output modules, you have to handle these types there appropriately.

41.4.8 Errors

Each module can provide a list of possible error messages contained e.g. in `syntax_errors.h` or `cpu_errors.h`. They are a comma-separated list of a `printf`-format string and error flags. Allowed flags are `WARNING`, `ERROR`, `FATAL`, `MESSAGE` and `NOLINE`. They can be combined using `or` (`|`). `NOLINE` has to be set for error messages during initialization or while writing the output, when no source context is available. Errors cause the assembler to return false. `FATAL` causes the assembler to terminate immediately.

The errors can be emitted using the function `syntax_error(int n, ...)`, `cpu_error(int n, ...)` or `output_error(int n, ...)`. The first argument is the number of the error message (starting from zero). Additional arguments must be passed according to the format string of the corresponding error message.

41.5 Support Functions

Useful support functions for writing CPU-, Syntax- and Output-modules.

41.5.1 Memory Functions

`void *mymalloc(size_t sz)`

Allocate memory. See also `mycalloc()` and `myrealloc()`.

`void myfree(void *p)`

Free an allocated memory block with one of the functions above.

`uint64_t readval(int be, void *src, size_t size)`

Reads an unsigned value with `size` target-bytes and byte-ordering `be` from the `src` pointer (0 is little-, 1 is big-endian).

`void *setval(int be, void *dest, size_t size, uint64_t val)`

Stores the value `val` to `dest`, which has a size of `size` target-bytes, using a byte-ordering of `be` (0 is little-, 1 is big-endian).

`uint64_t readbits(int be, void *p, unsigned bsize, unsigned offset, unsigned size)`

Reads an unsigned value from the bitfield with `bsize` bits at `p`. `size` bits at bit-offset `offset` will be extracted. Use endianness `be` while reading the bitfield.

`void setbits(int be, void *p, unsigned bsize, unsigned offset, unsigned size, uint64_t d)`

Writes the value `d` with `size` bits into a bitfield at `p` with `bsize` bits, starting at bit-offset `offset`, following the rules of the given endianness `be`.

`utaddr readbyte(void *src)`

Reads an unsigned target-byte from `src`.

`void writebyte(void *dest, utaddr val)`

Stores the value `val` into a target-byte at `dest`.

`OCTETS(n)`

This macro expands to the number of 8-bit host-bytes required for `n` target-bytes.

41.5.2 Expressions

`expr *parse_expr(char **pp)`

Parses any expression starting at `*pp`. Updates `*pp` to point at the next character after the expression.

`expr *parse_expr_tmplab(char **pp)`

Same as `parse_expr()`, but defines a temporary label for the PC symbol, so it does not change its value when this expression is evaluated at other section locations.

`expr *parse_expr_huge(char **pp)`

Parses a constant expression (no labels allowed) with 128 bit integers.

`expr *parse_expr_float(char **pp)`

Parses a floating point expression (no labels allowed), if the backend supports floating point constants (defines `FLOAT_PARSER`).

```

taddr parse_constexpr(char **pp)
    Parses an expression which is immediately evaluated to a constant value. Gen-
    erates an error message and returns zero when it depends on non-constant,
    128-bit (huge) or floating point values.

void free_expr(expr *tree)
    Free an expression.

int type_of_expr(expr *tree)
    Returns the type of an expression, which may be either NUM, HUG (128 bit
    constant) or FLT (floating point). This will tell you which evaluation function
    to use (see below).

void simplify_expr(expr *tree)
    Try to evaluate the expression as far as possible. Subexpressions only containing
    constants or absolute symbols are simplified.

int eval_expr(expr *tree, taddr *result, section *sec, taddr pc)
    Evaluate an expression using the pc in section sec and store the result in
    *result. The return value is non-zero when the result is constant (i.e. only
    depends on constants or absolute symbols).

int eval_expr_huge(expr *tree, thuge *result)
    Evaluate a constant 128-bit integer expression, which is written to *result.
    The return value becomes zero when there were problems (like unsupported
    operations).

int eval_expr_float(expr *tree, tfloat *result)
    Evaluate a constant floating point expression and write the result to *result.
    The return value becomes zero when there were problems (like unsupported
    operations).

int find_base(expr *p, symbol **base, section *sec, taddr pc)
    Tests, if an expression is based only on one non-absolute symbol plus constants
    or minus label. Writes either that symbol-pointer to *base or NULL. The return
    value defines the type of base-relative operation, which is BASE_OK (normal base
    plus constant), BASE_PCREL (base minus label is pc-relative) or BASE_ILLEGAL
    (illegal arithmetic operation with base symbol).

expr *number_expr(taddr val)
    Create a constant expression from val.

expr *huge_expr(thuge val)
    Create a constant 128-bit expression from val.

expr *float_expr(tfloat val)
    Create a constant floating point expression from val.

```

Syntax modules may overwrite the default operator tokens from `expr.h`. Every token macro follows the syntax `T_NAME(s)`, where `char *s` points to the next token from the stream. The macro has to return the token length in characters on match or zero otherwise. Note, that the result of logical operations is transformed by the `BOOLEAN(x)` macro, which may be

overwritten by a backend, for example to define True as -1 instead of the default 1. The following token macros are known (default token follows the macro name):

```
T_LOR(s) ||
    Logical Or.

T_LAND(s) &&
    Logical And.

T_EQ(s) ==
    Equality.

T_NEQ(s) !=
    Not equal.

T_LEQ(s) <=
    Lesser or equal.

T_GEQ(s) >=
    Greater or equal.

T_LT(s) < Lesser than.

T_GT(s) > Greater than.

T_ADD(s) +
    Addition.

T_SUB(s) -
    Subtraction.

T_MUL(s) *
    Multiplication.

T_DIV(s) /
    Division.

T_MOD(s) %
    Modulo.

T_BOR(s) |
    Bitwise Or.

T_XOR(s) ^
    Bitwise Exclusive Or.

T_BORN(s) 0
    Bitwise Or-Not, i.e. |~. Undefined by default.

T_BAND(s) &
    Bitwise And.

T_LSH(s) <<
    Shift Left.

T_RSH(s) >>
    Shift Right. May be signed or unsigned. A backend may set the global variable
    unsigned_shift to non-zero. Defaults to signed.
```

`T_PLUS(s) +`
Unary Plus operator. Is ignored.

`T_MINUS(s) -`
Unary Negation operator.

`T_NOT(s) !`
Unary logical Not.

`T_CPL(s) ~`
Unary bitwise Complementation.

Expression evaluation priority for all these tokens cannot be modified, and is defined from highest to lowest:

1. `T_PLUS`, `T_MINUS`, `T_NOT`, `T_CPL`
2. `T_LSH`, `T_RSH`
3. `T_BAND`
4. `T_XOR`, `T_BORN`
5. `T_BOR`
6. `T_MUL`, `T_DIV`, `T_MOD`
7. `T_ADD`, `T_SUB`
8. `T_LEQ`, `T_GEQ`, `T_LT`, `T_GT`
9. `T_EQ`, `T_NEQ`
10. `T LAND`
11. `T_LOR`

41.5.3 Symbols

`symbol *new_abs(const char *name, expr *tree)`
Create a new absolute symbol (type `EXPRESSION`).

`symbol *new_equate(const char *name, expr *tree)`
Same as `new_abs()` but set the `EQUATE` flag to indicate that the symbol must not change its value (unlike a `.set` symbol).

`symbol *new_import(const char *name)`
Add an externally defined symbol name.

`symbol *new_labsym(section *sec, const char *name)`
Create a label symbol (type `LABSYM`) for the current PC in the given section. Uses the current/default section when `sec` is `NULL`.

`symbol *new_tmplabel(section *sec)`
Create a temporary label symbol with a unique name for the given section.

`symbol *internal_abs(const char *name)`
Create an internal absolute symbol (type `EXPRESSION`), which gets the flag `VASMINTERN`, so it is never written into object files.

```
regsym *new_regsym(int redef, const char *name, int type, unsigned int
flags, unsigned int num)
```

(If HAVE_REGS YMS is set.) Create a new CPU register symbol with name `name` and number `num`. The `type` and `flags` can be used by the backend as needed (for example integer and floating point register type). A non-zero `redef` allows redefinition of a register symbol without error message.

```
regsym *find_regsym(const char *name, int len)
```

(If HAVE_REGS YMS is set.) Find register symbol with `name` and `len` characters.

41.5.4 Macros, Structures and Repetitions

```
macro *new_macro(char *name, struct namelen *maclist, struct namelen
*endmlist, char *args)
```

Make the parser read a new macro with the given `name`, which has `namelen` characters. `maclist` is a list of `struct namelen` structures, which define all macro directive names currently known to the syntax module and `endmlist` defines the directives which terminate a macro definition. The parser will use these lists to parse and store the body of the macro definition. `args` may optionally point to a string of named macro arguments. It is parsed by using the macros `SKIP_MACRO_ARGNAME`, `MACROS_ARG_OPTS` and `MACRO_ARG_SEP` from `syntax.h`.

```
macro *find_macro(char *name, int name_len)
```

Find the macro using the name at `name` with `name_len` characters.

```
int execute_macro(char *name, int name_len, char **q, int *q_len, int nq, char *p)
```

Execute the macro with the given name (which is `name_len` characters long). If `nq` is greater than zero, `q` points to a list of qualifiers (with their length in characters defined by `q_len`), provided the cpu-backend supports qualifiers and they are not disabled for macros (`NO_MACROS_QUALIFIERS`). Missing qualifiers are set to the CPU's default (e.g. `.w` for m68k). Finally, `p` points to a string of parameters passed into the macro. An optional `EXEC_MACRO()` macro is called just before execution, when defined by the syntax module.

```
int leave_macro(void)
```

Terminate execution of the current macro (e.g. by a `mexit` directive).

```
int undef_macro(char *name)
```

Undefine this macro. It will be unknown for the remaining source text.

```
int new_structure(char *name)
```

Start a structure definition using an offset-section with called `name`.

```
int end_structure(section **prev)
```

End a structure definition and return the previously active section in `prev`. The syntax module may then do some finalizing tasks and switch back to this section.

```
section *find_structure(char *name, int name_len)
```

Find and return the offset-section for the given structure name. The functionality for "executing" the structure is in the syntax module.

```
void new_repeat(int rcnt, char *name, char *vals, struct namelen
*reptlist, struct namelen *endrlist)
```

Make the parser repeat the source text between the following line and any end-repeat directive defined by `endrlist` `rcnt` times. Valid repeat directives are defined by `reptlist` for nesting. If `name` is not NULL then it defines an additional symbol name to be set to the current iteration counter. If `rcnt` is set to `REPT_IRP` then `vals` points to a string of comma separated values, which are assigned one by one for every iteration to the symbol with `name` (prefixed by a backslash). If `rcnt` is set to `REPT_IRPC` then `vals` points to a string of comma separated characters instead.

```
void set_nocase_macros(int nc)
```

Change case-sensitivity for macro and structure names to case-insensitive (`nc==1`), case-sensitive (`nc==0`) or using the symbol default (`nc==-1`).

41.5.5 Hash Tables

Modules may create hash tables for their own purpose with any size or case-sensitivity. Hash table entries have the type `hashdata`, which is a union with currently at least the following choices:

```
typedef union hashdata {
    void *ptr;
    unsigned idx;
} hashdata;
```

```
hashtable *new_hashtable(size_t sz, int no_case)
```

Create a new hash table with `sz` entries. The `no_case` parameter defines whether names will be handled case-sensitive (`no_case==0`) or case-insensitive (`no_case==1`).

```
hashtable *new_hashtable_c(size_t sz)
```

Create a new hash table with `sz` case-sensitive entries.

```
hashtable *new_hashtable_nc(size_t sz)
```

Create a new hash table with `sz` case-insensitive entries.

```
hashtable *new_hashtable_sc(size_t sz)
```

Create a new hash table with `sz` entries using the symbol default case-sensitivity.

```
void add_hashentry(hashtable *ht, const char *name, hashdata d)
```

Make a hash table entry with `d` for the given `name`.

```
void rem_hashentry(hashtable *ht, const char *name)
```

Remove the entry for `name` from the given hash table.

```
int find_name(hashtable *ht, const char *name, hashdata *d)
```

Find `name` in the given hash table and return its `hashdata` in `d`. The return code is zero when `name` doesn't exist.

```
int find_namelen(hashtable *ht, const char *name, int len, hashdata *d)
```

Find an entry which matches the string at `name` with `len` characters in the given hash table and return its `hashdata` in `d`. The return code is zero when `name` doesn't exist.

41.5.6 Atoms

`instruction *new_inst(const char *inst, int len, int op_cnt, char **op, int *op_len)`
 Traverses the backend's mnemonic table for instructions called `inst` and tries to parse `op_cnt` operands using the cpu module's `parse_operand()` function. Returns an instruction pointer when the requirements from the mnemonic table were met or NULL.

`dblock *new_dblock(void)`
 Allocate a new `dblock` structure for storing constant data.

`sblock *new_sblock(expr *space, size_t size, expr *fill)`
 Allocate a new `sblock` structure for storing `space` elements of `size` target-bytes, filled with `fill`, or zeros when `fill` is NULL.

`atom *new_inst_atom(instruction *p)`
 Allocate a new INSTRUCTION atom from `p`.

`atom *new_data_atom(dblock *p, taddr align)`
 Allocate a new DATA atom with alignment `align` atom from `p`.

`atom *new_label_atom(symbol *p)`
 Allocate a new LABEL atom from symbol `p`.

`atom *new_space_atom(expr *space, size_t size, expr *fill)`
 Allocate a new SPACE atom with `space` elements of `size` target-bytes, filled with `fill`, or zeros when `fill` is NULL.

`atom *new_datadef_atom(size_t bitsize, operand *op)`
 Allocate a new DATADEF atom from operand `op` with `bitsize` bits.

`atom *new_atom(int type, taddr align)`
 Allocate a new atom with given type and alignment.

`void add_atom(section *sec, atom *a)`
 Adds a new atom to the end of the specified section. If `sec` is NULL then the current section is used. If there is no current section yet, then a default section is created.

41.6 Syntax modules

A new syntax module must have its own subdirectory under `vasm/syntax`. At least the files `syntax.h`, `syntax.c` and `syntax_errors.h` must be written.

41.6.1 The file `syntax.h`

`#define ISIDSTART(x)/ISIDCHAR(x)`
 These macros should return non-zero if and only if the argument is a valid character to start an identifier or a valid character inside an identifier, respectively. `ISIDCHAR` must be a superset of `ISIDSTART`.

`#define ISBADID(p, l)`
 Even with `ISIDSTART` and `ISIDCHAR` checked, there may be combinations of characters which do not form a valid initializer (for example, a single character).

This macro returns non-zero, when this is the case. First argument is a pointer to the new identifier and second is its length.

#define ISEOL(x)

This macro returns true when the string pointing at **x** is either a comment character or end-of-line.

#define CHKIDEND(s,e) chkidend((s),(e))

Defines an optional function to be called at the end of the identifier recognition process. It allows you to adjust the length of the identifier by returning a modified **e**. Default is to return **e**. The function is defined as **char *chkidend(char *startpos, char *endpos)**.

#define BOOLEAN(x) -(x)

Defines the result of boolean operations. Usually this is **(x)**, as in C, or **-(x)** to return -1 for True.

#define NARGSYM "NARG"

Defines the name of an optional symbol which contains the number of arguments in a macro.

#define CARGSYM "CARG"

Defines the name of an optional symbol which can be used to select a specific macro argument with **\.**, **\+** and **\-**.

#define REPTNSYM "REPTN"

Defines the name of an optional symbol containing the counter of the current repeat iteration.

#define EXPSKIP(s) exp_skip(s)

Defines an optional replacement for **skip()** to be used in **expr.c**, to skip blanks in an expression. Useful to forbid blanks in an expression and to ignore the rest of the line (e.g. to treat the rest as comment). The function is defined as **char *exp_skip(char *stream)**.

#define IGNORE_FIRST_EXTRA_OP 1

Should be defined as non-zero (true) if the syntax module wants to ignore the operand field on instructions without an operand. Useful, when everything following the operand should be regarded as comment, without requiring a comment character.

#define MAXMACPARAMS 35

Optionally defines the maximum number of macro arguments, if you need more than the default number of 9.

#define SKIP_MACRO_ARGNAME(p) skip_identifier(p)

An optional function to skip a named macro argument in the macro definition. Argument is the current source stream pointer. The default is to skip an identifier.

#define MACRO_ARG_OPTS(m,n,a,p) NULL

An optional function to parse and skip options, default values and qualifiers for each macro argument. Returns **NULL** when no argument options have been found. Arguments are:

`struct macro *m;` Pointer to the macro structure being currently defined.

`int n;` Argument index, starting with zero.

`char *a;` Name of this argument.

`char *p;` Current source stream pointer. An updated pointer will be returned.

Defaults to unused.

`#define MACRO_ARG_SEP(p) (*p==',' ? skip(p+1) : NULL)`
 An optional function to skip a separator between the macro argument names in the macro definition. Returns NULL when no valid separator is found. Argument is the current source stream pointer. Defaults to using comma as the only valid separator.

`#define MACRO_PARAM_SEP(p) (*p==',' ? skip(p+1) : NULL)`
 An optional function to skip a separator between the macro parameters in a macro call. Returns NULL when no valid separator is found. Argument is the current source stream pointer. Defaults to using comma as the only valid separator.

`#define EXEC_MACRO(s)`
 An optional function to be called just before a macro starts execution. Parameters and qualifiers are already parsed. Argument is the **source** pointer of the new macro. Defaults to unused.

`#define CHAR_CONST_TRANSFORM(c,q) (c)`
 Define this if you want to transform character constant `c` in expressions according to the current state of the module and/or the character constant prefix `q`. Example: Set bit 7 on characters whenever the constant was prefixed by a `"` instead of `'`.

41.6.2 The file `syntax.c`

A syntax module has to provide the following elements (all other functions should be **static** to prevent name clashes):

`const char *syntax_copyright;`
 A string that will be emitted as part of the copyright message.

`hashtable *dirhash;`
 A pointer to the hash table with all directives.

`char commentchar;`
 A character used to introduce a comment until the end of the line.

`int dotdirs;`
 Define `dotdirs` as non-zero, when the syntax module works with directives starting with a dot (`.`).

`int init_syntax(void);`
 Will be called during startup, after argument parsing. Must return zero if initializations failed, non-zero otherwise.

```
int syntax_args(char *);
```

This function will be called with the command line arguments (unless they were already recognized by other modules). If an argument was recognized, return non-zero.

```
int syntax_defsect(void);
```

Lets the syntax module define a default section, which is used when no section was created by any **section** or **org** directive, before the first code or data is defined. May set **defsectname**, **defsecttype** and **defsectorg** accordingly and return with non-zero. Or return with zero and accept the defaults, which are: **defsectname**=".text" and **defsecttype**="acrx".

```
char *skip(char *);
```

A function to skip whitespace etc.

```
void eol(char *);
```

This function should check that the argument points to the end of a line (only comments or whitespace following). If not, an error or warning message should be omitted.

```
char *const_prefix(char *,int *);
```

Check if the first argument points to the start of a constant. If yes return a pointer to the real start of the number (i.e. skip a prefix that may indicate the base) and write the base of the number through the pointer passed as second argument. Return zero if it does not point to a number.

```
char *const_suffix(char *,char *);
```

First argument points to the start of the constant (including prefix) and the second argument to first character after the constant (excluding suffix). Checks for a constant-suffix and skips it. Return pointer to the first character after that constant. Example: constants with a 'h' suffix to indicate a hexadecimal base.

```
void parse(void);
```

This is the main parsing function. It has to read source text lines via the **read_next_line()** function, parse them and create sections, atoms and symbols. Pseudo directives are usually handled by the syntax module. Instructions can be parsed by the cpu module using **parse_instruction()**.

```
char *parse_macro_arg(struct macro *,char *,struct namelen *,struct namelen *);
```

Called to parse a macro parameter by using the source stream pointer in the second argument. The start pointer and length of a single passed parameter is written to the first **struct namelen**, while the optionally selected named macro argument is passed in the second **struct namelen**. When the **len** field of the second **namelen** is zero, then the argument is selected by position instead by name. Returns the updated source stream pointer after successful parsing.

```
int expand_macro(source *,char **,char *,int);
```

Expand parameters and special commands inside a macro source. The second argument is a pointer to the current source stream pointer, which is updated

on any successful expansion. The function will return the number of characters written to the destination buffer (third argument) in this case. Returning -1 means: no expansion took place. The last argument defines the space in characters which is left in the destination buffer.

```
char *get_local_label(char **);
```

Gets a pointer to the current source pointer. Has to check if a valid local label is found at this point. If yes return a pointer to the vasm-internal symbol name representing the local label and update the current source pointer to point behind the label.

Have a look at the support functions provided by the frontend to help.

Syntax modules may support additional features, which can be enabled or disabled by a pre-processor define. Like allowing the # character for introducing comments in the std-syntax module, when the CPU's operand parser doesn't need it (Example: PPC or x86). Defines for these optional features follow the general syntax [module type]_[module name]_[feature name].

```
#define SYNTAX_STD_COMMENTCHAR_HASH
```

41.7 CPU modules

A new cpu module must have its own subdirectory under `vasm/cpus`. At least the files `cpu.h`, `cpu.c` and `cpu_errors.h` must be written.

41.7.1 The file `cpu.h`

A cpu module has to provide the following elements (all other functions should be `static` to prevent name clashes) in `cpu.h`:

```
#define LITTLEENDIAN 1
```

```
#define BIGENDIAN 0
```

Define these according to the target endianness. For CPUs which support big- and little-endian, you may assign a global variable here. So be aware of it, and never use `#if BIGENDIAN`, but always `if(BIGENDIAN)` in your code.

```
#define VASM_CPU_<cpu> 1
```

Defines a cpu-specific macro. May be used to perform special handling in syntax- or output-modules.

```
#define BITSPERBYTE 8
```

The number of bits per byte of the target cpu. Usually 8. We require that vasm is running on a host architecture and file system which uses 8-bit bytes. When writing output for a backend with `BITSPERBYTE > 8` the vasm-internal ordering of 8-bit host-bytes within a target-byte is big-endian.

```
#define MAX_OPERANDS 3
```

Maximum number of operands of one instruction.

```
#define MAX_QUALIFIERS 0
```

Maximum number of mnemonic-qualifiers per mnemonic.

```
#define NO_MACRO_QUALIFIERS
    Define this, when qualifiers shouldn't be allowed for macros. For some archi-
    tectures, like ARM, macro qualifiers make no sense.

typedef int32_t taddr;
    Data type to represent a target-address. Preferably use the types from
    stdint.h. Does not necessarily have to match the cpu's address bus size
    (refer to bytespertaddr), but choose it according to the largest data you will
    be able to do calculations with. For example, you may want to allow 32-bit
    data definitions for an 8-bit cpu.

typedef uint32_t utaddr;
    Unsigned data type to represent a target-address.

#define INST_ALIGN 2
    Minimum instruction alignment.

#define DATA_ALIGN(n) ...
    Default alignment for n-bit data. Can also be a function.

#define DATA_OPERAND(n) ...
    Operand class for n-bit data definitions. Can also be a function. Negative
    values denote a floating point data definition of -n bits.

typedef ... operand;
    Structure to store an operand for a machine instruction or a data constant.
    Stores, for example, addressing modes and expressions.

typedef ... mnemonic_extension;
    Mnemonic extension for the cpu's instruction table. Often used for the actual
    opcode or cpu-model flags.
```

Optional features, which can be enabled by defining the following macros:

```
#define FLOAT_PARSER 1
    Enables the floating point parser and floating point evaluation in the
    expression module. With this option the backend has to be prepared that
    expressions may contain floating point constants, which can be checked
    by testing the result of type_of_expr(expression) for FLT. Then use
    eval_expr_float(expression,&float_val) to retrieve the floating point
    value with type tfloat. It is up to the backend to convert the host's floating
    point format, which should be IEEE, into the backend's native format. The
    vasm frontend only supports IEEE to IEEE conversion via conv2ieee32()
    and conv2ieee64().

#define HAVE_INSTRUCTION_EXTENSION 1
    If cpu-specific data should be added to all instruction atoms.

typedef ... instruction_ext;
    Type for the above extension.

#define HAVE_CPU_SECT_EXTENSION 1
    If cpu-specific data should be added to all sections.
```

```
typedef ... section_extc;
    Type for the above extension.

#define HAVE_CPU_CLEANUP_PARSE 1
    When defined, vasm calls the function cpu_cleanup_parse(section *), with a
    pointer to the first section, directly after parsing has finished and all atoms have
    been created. You can use it to modify the sections before assembly begins.

#define CLEAR_OPERANDS_ON_START 1
    Backend requires zeroed operand structures when calling parse_operand() for
    the first time. Might be useful to parse operands only once. Defaults to unde-
    fined.

#define CLEAR_OPERANDS_ON_MNEMO 1
    Backend requires zeroed operand structures when calling parse_operand() for
    any new mnemonic. Useful to parse the same operand multiple times on the
    current mnemonic, but reset everything for the next mnemonic. Defaults to
    undefined.

START_PARENTH(x)
    Valid opening parenthesis for instruction operands. Defaults to '('.

END_PARENTH(x)
    Valid closing parenthesis for instruction operands. Defaults to ')'.

#define MNEMONIC_VALID(idx)
    An optional function with the arguments (int idx). Returns true when the
    mnemonic with index idx is valid for the current state of the backend (e.g. it
    is available for the selected cpu model).

#define MNEMOHTABSIZE 0x4000
    You can optionally overwrite the default hash table size defined in vasm.h. May
    be necessary for larger mnemonic tables. Run vasm with option -debug to print
    the number of collisions in the hash tables.

#define OPERAND_OPTIONAL(p,t)
    When defined, this is a function with the arguments (operand *op,int type),
    which returns true when the given operand type (type) is optional. The func-
    tion is only called for missing operands and should also initialize op with default
    values (e.g. 0).
```

Implementing additional target-specific unary operations is done by defining the following optional macros:

```
#define EXT_UNARY_NAME(s)
    Should return True when the string in s points to an operation name we want
    to handle.

#define EXT_UNARY_TYPE(s)
    Returns the operation type code for the string in s. Note that the last valid
    standard operation is defined as LAST_EXP_TYPE, so the target-specific types
    will start with LAST_EXP_TYPE+1.
```

```
#define EXT_UNARY_EVAL(t,v,r,c)
```

Defines a function with the arguments (`int t`, `taddr v`, `taddr *r`, `int c`) to handle the operation type `t` returning an `int` to indicate whether this type has been handled or not. Your operation will be applied on the value `v` and the result is stored in `*r`. The flag `c` is passed as 1 when the value is constant (no relocatable addresses involved).

```
#define EXT_FIND_BASE(b,e,s,p)
```

Defines a function with the arguments (`symbol **b`, `expr *e`, `section *s`, `taddr p`) to save a pointer to the base symbol of expression `e` into the symbol pointer, pointed to by `b`. The type of this base is given by an `int` return code. Further on, `e->type` has to be checked to be one of the operations to handle. The section pointer `s` and the current pc `p` are needed to call the standard `find_base()` function.

41.7.2 The file `cpu.c`

A `cpu` module has to provide the following elements (all other functions and data should be `static` to prevent name clashes) in `cpu.c`:

```
int bytespertaddr;
```

The number of bytes per target address. Note, that this really defines the size of a backend's address pointer in target-bytes and might differ from the actual size of `taddr` (see above).

```
mnemonic mnemonics[];
```

The mnemonic table keeps a list of mnemonic names and operand types the assembler will match against using `parse_operand()`. It may also include a target specific `mnemonic_extension`.

```
const char *cpu_copyright;
```

A string that will be emitted as part of the copyright message.

```
const char *cpuname;
```

A string describing the target `cpu`.

```
int init_cpu(void);
```

Will be called during startup, after argument parsing. Must return zero if initializations failed, non-zero otherwise.

```
int cpu_args(char *);
```

This function will be called with the command line arguments (unless they were already recognized by other modules). If an argument was recognized, return non-zero.

```
char *parse_cpu_special(char *);
```

This function will be called with a source line as argument and allows the `cpu` module to handle `cpu`-specific directives etc. Functions like `eol()` and `skip()` from the `syntax`-module should be used to keep the `syntax` consistent.

```
operand *new_operand();
```

Allocate and initialize a new operand structure.

```
int parse_operand(char *text,int len,operand *op,int requires);
```

Parses the source text for an instruction's operand at `text` with length `len` to fill the target specific operand structure pointed to by `op`, whenever the operand's type matches the type from `requires`. Return with one of the following codes:

PO_NOMATCH The source did no match the operand type given in `requires`.

PO_CORRUPT The source was definitely identified as garbage, making it useless to try matching it against any other operand types.

PO_MATCH The parsed source matches the operand type in `requires`. As soon as all the instruction's operands have been matched, the instruction is successfully recognized.

PO_SKIP Works like **PO_MATCH**, but skips the next operand from the mnemonic table. For example, because it was already handled together with the current operand.

PO_COMB_OPT Works like **PO_MATCH**, but requests parsing of the next argument from the source text, if any, with a pointer to the same `operand` structure as before. This makes it possible to merge multiple operands into a single operand structure.

PO_COMB_REQ Like **PO_COMB_OPT**, requests parsing of the next argument with a pointer to the same `operand` structure. But this time the additional argument is mandatory.

PO_NEXT Source did not match the given operand type in `requires`. Request parsing the same chunk of source text again, but using the following operand type. Can be used to break a **PO_COMB_OPT** or **PO_COMB_REQ** attempt and continue normally.

```
size_t instruction_size(instruction *ip, section *sec, taddr pc);
```

Returns the size of the instruction `ip` in target-bytes, which, in the final pass, must be identical to the number of bytes written by `eval_instruction()` (see below).

```
dblock *eval_instruction(instruction *ip, section *sec, taddr pc);
```

Converts the instruction `ip` into a DATA atom, including relocations when necessary.

```
dblock *eval_data(operand *op, taddr bitsize, section *sec, taddr pc);
```

Converts a data operand into a DATA atom, including relocations.

```
void init_instruction_ext(instruction_ext *);
```

(If **HAVE_INSTRUCTION_EXTENSION** is set.) Initialize an instruction extension.

```
char *parse_instruction(char *,int *,char **,int *,int *);
```

(If **MAX_QUALIFIERS** is greater than 0.) Parses instruction and saves extension locations.


```
int set_default_qualifiers(char **,int *);
    (If MAX_QUALIFIERS is greater than 0.) Saves pointers and lengths of default
    qualifiers for the selected CPU and returns the number of default qualifiers.
    Example: for a M680x0 CPU this would be a single qualifier, called "w". Used
    by execute_macro().
```

```
cpu_opts_init(section *);
    (If HAVE_CPU_OPTS is set.) Gives the cpu module the chance to write out OPTS
    atoms with initial settings before the first atom for a section is generated.
```

```
cpu_opts(void *);
    (If HAVE_CPU_OPTS is set.) Apply option modifications from an OPTS atom. For
    example: change cpu type or optimization flags.
```

```
print_cpu_opts(FILE *,void *);
    (If HAVE_CPU_OPTS is set.) Called from print_atom() to print an OPTS atom's
    contents.
```

```
cpu_init_section(section *);
    (if HAVE_CPU_SECT_EXTENSION is set.) Gives you the possibility to initialize
    cpu-specific elements for any new sections. Refer to section_extc typedef.
```

41.8 Output modules

Output modules can be chosen at runtime rather than compile time. Therefore, several output modules are linked into one vasm executable and their structure differs somewhat from syntax and cpu modules.

Usually, an output module for some object format `fmt` should be contained in a file `output_<fmt>.c` (it may use/include other files if necessary). To automatically include this format in the build process, the `OUTFMTS` definition in `make.rules` has to be extended. The module should be added to the `OBJS` variable at the start of `make.rules`. Also, a dependency line should be added (see the existing output modules).

An output module must only export a single function which will return pointers to necessary data/functions. This function should have the following prototype:

```
int init_output_<fmt>(
    char **copyright,
    void (**write_object)(FILE *,section *,symbol *),
    int (**output_args)(char *)
);
```

In case of an error, zero must be returned. Otherwise, It should perform all necessary initializations, return non-zero and return the following output parameters via the pointers passed as arguments:

copyright

A pointer to the copyright string.

write_object

A pointer to a function emitting the output. It will be called after the assembler has completed and will receive pointers to the output file, to the first section

of the section list and to the first symbol in the symbol list. See the section on general data structures for further details.

`output_args`

A pointer to a function checking arguments. It will be called with all command line arguments (unless already handled by other modules). If the output module recognizes an appropriate option, it has to handle it and return non-zero. If it is not an option relevant to this output module, zero must be returned.

At last, a call to the `init_output_<fmt>()` has to be added in the `init_output()` function in `vasm.c` (should be self-explanatory). Besides assigning the above mentioned function pointers, this function can be used to redefine the assembler's behaviour. For example you may optionally set the following global variables:

`asciiout = 1;`

Set when the output module likes the output file to be opened in text-mode instead of binary-mode.

`unnamed_sections = 1;`

Set when the output module cannot handle section names. Usually such an output module differentiates sections by their type only: text, data or bss.

`secname_attr = 1;`

Set when the section attributes are used to differentiate between two sections with the same name.

`output_bitsperbyte = 1;`

Set when the output module supports target-bytes with `BITSPERBYTE`. Otherwise it is expected that all output modules do work at least with 8-bit target-bytes.

`output_indirect = 1;`

Set when the output module supports indirect symbols. They can be recognized as a symbol of type `EXPRESSION` with the flag `SYMINDIR`. The expression's base-symbol provides the referenced indirect symbol.

Writing a section's contents is typically done by traversing over all the section's atoms, establish alignment and write the contents of a `DATA` or `SPACE` atom using `fwdblock()` or `fwsblock()` into the output file.

```
section *s;
atom *p;

for (p=s->first,pc=(unsigned long long)s->org; p; p=p->next) {
    npc = fwpcalign(f,p,s,pc);

    if (p->type == DATA)
        fwdblock(f,p->content.db);
    else if (p->type == SPACE)
        fwsblock(f,p->content.sb);

    pc = npc + atom_size(p,s,npc);
}
```

```
}

```

Useful support functions for output modules, when writing data into the output file:

```
void fw8(FILE *f,uint8_t x)
    Write 8 bits of data.

void fw16(FILE *f,uint16_t x,int be)
    Write 16 bits of data with endianness be (0 is little, 1 is big).

void fw24(FILE *f,uint32_t x,int be)
    Write 24 bits of data with endianness be (0 is little, 1 is big).

void fw32(FILE *f,uint32_t x,int be)
    Write 32 bits of data with endianness be (0 is little, 1 is big).

void fwdata(FILE *f,const void *buf,size_t n)
    Write n 8-bit bytes of data.

void fwspace(FILE *f,size_t n)
    Write n zeroed 8-bit bytes.

void fwbytes(FILE *f,void *buf,size_t n)
    Write n target-bytes (BITSPERBYTE).

void fwdblock(FILE *f,dblock *db)
    Write the target-bytes within a dblock.

void fwsblock(FILE *f,sblock *sb)
    Write the target-bytes within a sblock.

void fwalign(FILE *f,taddr n,taddr align)
    Write as many zero target-bytes as required to align address n to align bytes.

int fwpattern(FILE *f,taddr n,uint8_t *pat,int patlen)
    Write n target-bytes, which are initialized with pattern pat. The patlen is
    given in target-bytes as well. Note, that the pattern output may be preceded
    by a number of zero-bytes when n is not a multiple of patlen. The function
    returns non-zero if that happened.

taddr fwpcalign(FILE *f,atom *a,section *sec,taddr pc)
    Write as many target-bytes as required to achieve proper alignment for atom a.
    This space will either be filled by a SPACE atom's fill-pattern, or otherwise by
    the section's default pattern (section.pad). The newly aligned pc is returned.
```

Some remarks:

- Note, that the output of atoms and target-bytes, using the functions from above, is done by writing 8-bit bytes to the host's file system. The order of 8-bit bytes within target-bytes which are greater than 8 bits can be selected by the options **-obe** for big-endian and **-ole** for little-endian. The target-bytes are automatically written in that selected order, when using the **fw*** functions from above. Otherwise you can check the global variable **output_bytes_le**, which will be zero for big-endian and non-zero for little-endian target-byte output. The default is vasm's internal target-byte endianness, which is big-endian.

- Some output modules cannot handle all supported CPUs. Nevertheless, they have to be written in a way that they can be compiled. If code references CPU-specifics, they have to be enclosed in `#ifdef VASM_CPU_MYCPU ... #endif` or similar.
Also, if the selected CPU is not supported, the init function should fail.
- Error/warning messages can be emitted with the `output_error` function. As all output modules are linked together, they have a common list of error messages in the file `output_errors.h`. If a new message is needed, this file has to be extended (see the section on general data structures for details). When the cause for an error relates to an `atom` you may also use the `output_atom_error` function instead, which additionally prints the atom's source text line. In `output_errors.h` use the `NOLINE` flag when no atom is available.
- `vasm` has a mechanism to specify rather complex relocations in a standard way (see the section on general data structures). They can be extended with CPU-specific relocations, but usually CPU modules will try to create standard relocations (sometimes several standard relocations can be used to implement a CPU-specific relocation). An output module should try to find appropriate relocations supported by the object format. The goal is to avoid special CPU-specific relocations as much as possible.

Volker Barthelmann vb@compilers.de